Benemérita Universidad Autónoma de Puebla.

Facultad de Ciencias de la Computación.

Aplicación móvil con Android para el seguimiento de toma de medicamentos usando Kotlin.

Tesis presentada para obtener el grado de:

Licenciatura en Ingeniería en Ciencias de la Computación.

Presenta:

Carlos Eduardo Corona Hernández

Asesor:

M.C. Pedro Bello López.

PUEBLA, PUEBLA **OCTUBRE 2019**

Agradecimientos

A mi asesor de Tesis M.C. Pedro Bello López, por haberme brinda su tiempo, atención y guía en la realización de este proyecto, así como por sus años compartiendo sus conocimientos a generaciones de estudiantes.

A Mi Padre Florentino Corona Díaz y Mi Madre Florencia Hernández Martínez por todo su amor, esfuerzo y sacrificio para hacer posible la realización de mi carrera universitaria en esta Máxima casa de Estudios y hacer posible que esté donde me encuentro hoy.

Contenido

Tabla de ilustraciones:

Introducción:

El presente trabajo de tesis tiene como principal objetivo la creación de una aplicación para el sistema operativo Android, aplicando los conocimientos, metodologías y procesos aprendidos a lo largo de mi estancia en la Facultad de Ciencias de la Computación de la Benemérita Universidad Autónoma de Puebla. En este documento se presentará el proceso de planeación y desarrollo que dará como resultado una aplicación móvil estable y funcional al alcance de cualquier persona con un dispositivo Android. Además, se utilizarán los lineamientos sugeridos por Google para el diseño y mantenimiento de aplicaciones Android para brindar un producto de calidad, mantenible y escalable.

La aplicación que se desarrolló utilizando el lenguaje de programación Kotlin, el cual fue creado por Google y la empresa rusa JetBrains que busca sustituir a Java como Lenguaje estándar para la creación de aplicaciones en el sistema operativo Android.

Actualmente el desarrollo y mantenimiento correcto de una aplicación móvil es un proceso complejo y delicado, pues hoy en día el nivel de madures que han alcanzado los sistemas operativos móviles está al nivel de los sistemas operativos de escritorio e incluso superándolos en algunos aspectos por ejemplo la velocidad de adopción de las actualizaciones cada vez que una nueva versión de un sistema operativo móvil es lanzada.

La idea detrás de esta aplicación es ofrecer una solución a aquellas personas que padezcan alguna afección medica que requiera la toma constante de uno o más medicamentos ya sea en un periodo de tiempo especificado o de forma indefinida por enfermedades crónicas, por ejemplo, la hipertensión. Y es que la toma de varios medicamentos al mismo tiempo puede ser un proceso complicado sobre todo cuando estos tienen que ser tomados en intervalos de tiempo precisos pues esto eleva la eficiencia del tratamiento.

Los resultados de este trabajo de tesis serán los siguientes:

- Un proyecto creado y configurado de Android Studio que contendrá el código fuente de la aplicación.
- Un sistema de control de versiones basado en Git para el respaldo y mantenimiento de nuestro código fuente.
- Toda la documentación relacionada directamente con el proyecto, como diagramas, arquitectura, elementos gráficos (imágenes, iconos, vectores), claves de licencias, etc.…
- Un paquete de instalación .apk correctamente ensamblado y firmado para su distribución.

Capítulo 1. Conceptos generales

En este capítulo explicaremos la historia y estado del campo del arte del ecosistema de los dispositivos móviles. Abarcaremos el tema desde dos perspectivas: primero hablaremos de los dispositivos móviles en sí, tanto del hardware como del software, pero de este último nos centraremos en los sistemas operativos móviles. En la segunda parte hablaremos de las aplicaciones móviles, la historia, concepto y su impacto más allá de un smartphone.

1.1 Los dispositivos móviles

Un Smartphone es un dispositivo de comunicación móvil, evolución del teléfono celular, pues además de contar con la capacidad de realizar y recibir llamadas telefónicas y SMS cuentan con un enorme poder de procesamiento de computo, permitiendo realizar funciones complejas, como el manejo de pantallas de alta resolución, toma de fotografías y vídeos de gran calidad, renderizado de gráficos 3D, etc.… además cada año suelen añadirse sensores cada vez más complejos y precisos como sensores biométricos, giroscopio, acelerómetro, brújula, barómetro, etc.… Esto hace de estos pequeños aparatos en poderosas computadoras de bolsillo que solo eran posibles en la ciencia ficción hace tan solo unos años.

Pero para poder manejar todo este hardware es necesario contar con un Software avanzado, ahí es donde entran los sistemas operativos móviles, los cuales son como su nombre indica Sistemas operativos diseñados especialmente para ejecutarse en arquitecturas de bajo consumo energético (Como la ARM y la PowerPC), además de brindar los mecanismos necesarios para que programadores externos puedan tener acceso a todas las funcionalidades del Hardware, servicios del fabricante, conectividad, etc.…

Los dispositivos móviles están evolucionando a pasos agigantados, pero muchas de las características que hoy damos por sentado no hubieran sido posibles de no ser por Apple, cuando en el año 2007 presentó el primer iPhone, un dispositivo que marcó un antes y un después en cuanto a las características que debía tener un Smartphone y un sistema operativo móvil.

Figura 1 "Form-factor de los Smartphones antes y después del iPhone" Figura descargada de http://icosmogeek.com/apple-wants-to-kill-these-samsung-products-in-patent-case/.

El iPhone impuso que los dispositivos móviles debían tener hardware potente que pudieran recibir constantes actualizaciones de software tanto del sistema operativo como de los programas de

terceros que pudieran ser instalados en dicho sistema operativo. Antes de eso, los smartphones existentes disponían capacidades dispersas, así tuvieran el mismo sistema operativo la experiencia de usuario y funcionalidades no eran homogéneas, así que muchas veces era necesario cambiar de dispositivo año con año si es que se quería disfrutar de las ultimas funcionalidades, así estas no requirieran de hardware nuevo, pero sí de software más actualizado.

Apple había comenzado una nueva guerra comercial [1], pues a partir del lanzamiento del iPhone y de iOS (en ese momento llamado iPhone OS) distintas empresas de tecnología comenzaron una carrera para lanzar o actualizar sus propios sistemas operativos y dispositivos móviles buscando convertirse en el fabricante insignia, es decir aquel que impusiera tendencias y guiara la evolución de este ecosistema.

Figura 2: Distintos sistemas operativos móviles. Figura descargada de https://www.unlockwindows.com/hot-mobile-operating-systems/.

Muchos sistemas operativos móviles veteranos comenzaron a tener problemas para seguir siendo una opción de compra a considerar para los usuarios pues no lograron lanzar nuevas funcionalidades con la sufriente rapidez para superar a Apple, mientras que otros, como BlackBerry continuó con su estrategia de lanzar dispositivos empresariales con teclados físicos y sin interfaces táctiles. Otras empresas como Nokia trataron de tomar sus sistemas operativos existentes y lanzaron dispositivos móviles con características similares a las del iPhone, pero no tuvieron éxito pues la experiencia de usuario era mala al no estar diseñados para trabajar con interfaces táctiles.

Pero también comenzaban a nacer nuevos sistemas operativos móviles, los cuales trataban de aprender de los errores de otros y de las experiencias del pasado para ser un rival de iOS, uno de estos sistemas era Android.

Android había comenzado en 2007 [2] como un sistema operativo para cámaras compactas creado por una compañía del mismo nombre, y que se basaba en el Kernel de Linux. Fue comprado por Google ese mismo año y continuó su desarrollo como un rival para el iPhone OS (posteriormente conocido como iOS). Android tomó muchas de las características de iOS, como las interfaces táctiles y la tienda de aplicaciones. El principal cambio respecto al sistema de Apple es que Android era abierto. Cualquier compañía o persona podía descargar el código fuente e instalarlo en cualquier plataforma de Hardware que cumpliera los requerimientos mínimos. Esto fue lo que impulsó a

Android a ser actualmente el sistema operativo móvil más usado [3], pues cada empresa puede lanzar tantos dispositivos como quiera, certificándolos ante Google para tener acceso a sus servicios (Maps, Gmail, YouTube, Play Services, etc.…) aunque esto no es siempre necesario (por este motivo es que hemos podido ver productos que usan Android distintos a Smartphones y tablets como Televisiones, lavadoras, Automóviles y muchos más).

Figura 3 Evolución del Look and Feel de Android.

Google ha habilitado el acceso a muchos de los componentes del núcleo de Android de forma nativa, esto permite que el usuario final pueda personalizarlo en gran detalle y del lado del programador es posible crear aplicaciones y dispositivos externos que se conectan al smartphone sin mucha dificultad para añadir capacidades extra.

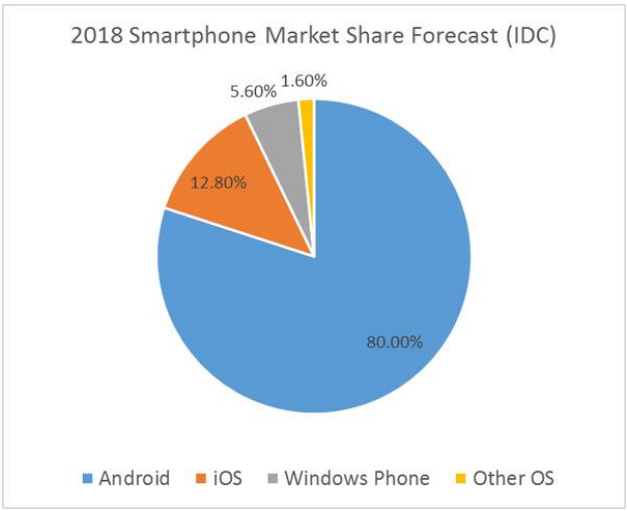

Figura 4 Distribución del mercado de los sistemas operativos móviles en 2018 Figura descargada de: https://www.intertech.com/Blog/ios-versus-android-matter/).

Para mala fortuna de los competidores de Android e iOS los usuarios habían descubierto que lo que Apple y Google podían ofrecer era mucho mejor que lo planteado hasta ese momento, la cuota de mercado de los antiguos sistemas operativos móviles fue disminuyendo poco a poco, a tal grado

que las empresas dejaron de crear actualizaciones y también dejaron de lanzar nuevos dispositivos que hacían uso de ellos, muchas de estas empresas quebraron, algunas se restructuraron y otras decidiendo migrar sus productos y servicios a Android.

La popularidad de Android ha sido tal que actualmente existen más de 24000 modelos distintos de dispositivos certificados y más de 1300 fabricantes en todo el mundo [4]

El avance tecnológico ha permitido que el abaratamiento de los dispositivos Android sea una realidad, ahora es posible poseer un smartphone con Android por una fracción de lo que costaban hace unos 10 años, es más, existen dispositivos tan económicos como $30 USD con las capacidades "básicas" que todo smartphone debe tener para brindar una experiencia de usuario completa y estable.

Pero quizás uno de los cambios más radícales que ha tenido Android desde su lanzamiento es que el pasado año del 2017 Google anunció el soporte oficial para Kotlin, un lenguaje de programación creado inicialmente en 2012 por la compañía JetBrains, una empresa muy importante para Google y el desarrollo Android, pues han tenido colaboraciones desde hace varios años, siendo quizás Android Studio, el IDE oficial para crear aplicaciones, su aportación más notoria.

Kotlin nació por la necesidad de Google de tener su propio lenguaje de programación, del cual poseyera todos los derechos, patentes y, sobre todo, conocimiento sobre su arquitectura.

Desde el lanzamiento de Android, el principal lenguaje de programación para crear aplicaciones era Java, esto principalmente porque Android utiliza versiones especializadas de la Máquina Virtual de Java según la arquitectura del procesador (Dalvik o Art), siendo cada aplicación abierta una instancia de dicha máquina virtual.

En 2012, Oracle creador y dueño de derechos de Java demandó a Google alegando que había infringido patentes respecto al uso del lenguaje Java y la arquitectura de la JVM Dalvik.

Google y JetBrains comenzaron a trabajar en Kotlin [6] a modo de prevención ante el caso de Oracle prohibiera el uso de Java para el desarrollo de apps Android. Pero no solo por cuestiones legales se comenzó a crear Kotlin como nuevo lenguaje de programación, si no por todos los problemas que se habían presentado en los años de existencia de Android y el desarrollo de aplicaciones. Y es que, aunque el lenguaje era prácticamente el mismo, la implementación de Google siempre iba atrás de la de Oracle, por lo que características nuevas tardaban más tiempo en ser adaptadas en Android, haciendo el desarrollo de proyectos mucho más lento pues un programador experimentado en Java tenía que comprobar que características eran compatibles y cuáles no.

Además, se comenzó a evidenciar otras carencias de Java respecto a otros lenguajes de programación, por ejemplo, la carencia de lambdas y la cantidad de "verbose" que se podía ahorrar usando Kotlin. Por verbose se puede definir a la cantidad de código escrito necesario para realizar una acción concreta [7].

Kotlin es dependiente de las librerías "core" de Java por tanto es interoperable, es decir es posible mezclar código creado en Java y Kotlin en un mismo archivo y compilar sin problemas.

La idea de Google es remplazar poco a poco el desarrollo de aplicaciones Android usando Java por Kotlin, buscando llegar a un punto donde el desarrollo con Java sea prácticamente nulo, aunque no incompatible, esto para seguir brindando soporte a grandes aplicaciones que hayan sido escritas en Java y no hayan emigrado aún a Kotlin [8]. Este movimiento sería similar a lo que Apple realizó cuando anuncio el lenguaje Swift en 2014 para sustituir paulatinamente a Objetive-c.

El soporte de Kotlin se hizo oficial con la salida de Android Studio 3.0 en el pasado mes de octubre del 2017 [9] y Google comenzó a alentar a los desarrolladores a crear y migrar las aplicaciones a Kotlin publicando en sitios web destacados de la propia empresa para fomentar su descarga e implementación del nuevo lenguaje.

Con esto también se crean nuevas áreas laborales para los estudiantes de carreras afines a la computación, pues las empresas comenzarán a necesitar programadores especializados en Kotlin a medida que pase el tiempo y Java comience a ser desplazado.

El salto es relativamente sencillo pues Google ha creado y actualizado una gran cantidad de su documentación para incluir temas de cómo crear componentes Android usando Kotlin, además de que el sitio web del proyecto ofrece una documentación completa sobre el lenguaje con un enfoque más general a distintos paradigmas como la programación orientada a objetos y la programación funcional para crear software de escritorio y aplicaciones web.

Los Smartphone se han convertido en un producto de consumo prácticamente indispensable para un gran porcentaje de la población mundial independientemente de su estatus socioeconómico, y no es de extrañar, ya que se han convertido en el principal medio de acceso a internet y de generación/consumo de contenido multimedia de mucha gente.

Según el último estudio del diario mexicano “El Economista” [10] en México, existen cerca de 96 millones de líneas telefonía celular activas que corresponden a Smartphones, de este porcentaje el 92.1% instaló aplicaciones móviles en su dispositivo, es decir se tienen más de 88 millones de posibles usuarios solo en México de cualquier posible aplicación móvil exitosa que llegue a desarrollarse para estos aparatos.

1.2 Aplicaciones móviles hoy en día

Uno de los mayores requerimientos de cualquier sistema informático es la capacidad de añadir funcionalidades extra después de su lanzamiento original sin necesidad de crear una nueva versión completa de dicho sistema.

Los primeros smartphones y teléfonos celulares en general hacían uso de microcontroladores para funcionar correctamente, estos primitivos dispositivos ofrecían características “fijas” es decir que las funcionalidades eran diseñadas meramente por el fabricante y que no se podían añadir posteriormente nuevas. El usuario estaba limitado a las aplicaciones que el fabricante añadía, por lo tanto, sí el usuario quería disfrutar de alguna nueva funcionalidad tenía que comprar un nuevo dispositivo. En la mayoría de las veces la funcionalidad deseada no era lo suficientemente significativa como para realizar gastos de I+D en su implementación, por lo que usualmente los fabricantes esperaban un cierto periodo de tiempo en lo que la tecnología avanzaba y los estudios de mercado arrojaban resultados de las necesidades de los usuarios para lanzar nuevos dispositivos con lotes de nuevas características.

Pero en la mayoría de los casos, muchas de estas funcionalidades eran especificas según el mercado, es decir lo que sería útil en país pudiera no serlo en otro, por lo que era necesario lanzar varias versiones de estos dispositivos lo que resultaba en gastos de operación extra para los fabricantes.

Los primeros sistemas operativos móviles ofrecieron una solución a este problema: proporcionar la arquitectura necesaria para ofrecer API's, de esta manera los fabricantes podrían ofrecer todas las funcionalidades base, pero además podrían crear aplicaciones rápidamente según los requerimientos del cliente sin tener que volver a lanzar el hardware.

Las primeras aplicaciones móviles eran por lo general calculadoras, agendas, videojuegos 2d y tonos de llamada dado el poco poder que podían ofrecer los dispositivos móviles en ese momento. Con la llegada de los primeros dispositivos con sistema operativo "completo" se presentaron nuevos retos, como la capacidad de añadir funcionalidades sin depender directamente del fabricante, por ejemplo, aplicaciones que eran requeridas por los operadores de telefonía móvil para ofrecer sus servicios de comunicación.

A raíz de esto Oracle propuso una implementación de Java dentro de los microcontroladores en los dispositivos móviles, se trata de Java Micro Edition y Java 2 Micro Edition [5], la cual estaba diseñada para brindar una interfaz de programación (API's) a los elementos del software y hardware por medio del lenguaje de programación java.

Figura 5 Programación para J2ME Figura descargada de https://plugins.jetbrains.com/plugin/1006-intellime-j2me-plugin.

J2ME proporciona características estándar para dispositivos de 160KB a 512KB de memoria interna entre ROM y RAM. Con esto los programadores podían crear una aplicación móvil que pudiera correr en distintos dispositivos con el mismo código fuente.

Sí bien J2ME permitía crear aplicaciones para dispositivos móviles estas eran muy limitadas por la falta de recursos, por lo que usualmente era usado para la programación de Juegos 2D.

Otros sistemas operativos ofrecieron otras soluciones para la creación de Software adicional como SDK's nativos, los cuales permitían el acceso a la mayoría de las capacidades de los dispositivos por lo que los programadores podían crear aplicaciones más complejas, pues podían utilizar la mayoría de los recursos del dispositivo, tanto de hardware como el software.

Pero otro problema que existía en ese momento era la distribución del Software, ya que para instalar una aplicación era necesario realizar procesos complejos usando PC's de escritorio, los cuales usualmente requerían conocimientos especializados, ya que cualquier falló durante la instalación de una aplicación podía dañar el dispositivo incluso de forma irreparable. Además, que

para tener acceso a software externo uno tenía que recurrir a CD's y las páginas web de los programadores para conseguir los archivos necesarios para la instalación, básicamente de la misma manera a la que se hacía en las computadoras de escritorio.

Apple tomó la de idea de ofrecer a los programadores la capacidad de tener acceso a las API's completas del Sistema de la misma manera que lo es en el software de aplicación en sistemas operativos de escritorio, pero además decidió crear un centro de distribución virtual, donde los programadores pudieran publicar las aplicaciones y juegos creados. Por otro lado, los usuarios finales pueden entrar a estos centros de distribución desde el dispositivo para instalar aplicaciones sin necesidad de un equipo de escritorio con software adicional, pero sobre todo sin engorrosos procesos técnicos.

Estos centros de distribución recibieron el nombre de "Tienda de aplicaciones", esto debido a que Apple bautizó así a su servicio: "App Store".

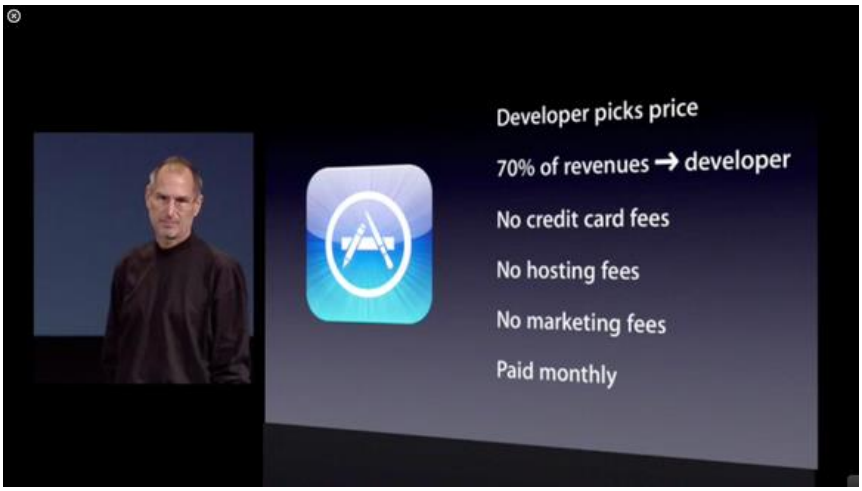

Figura 6 Presentación de la App Store de Apple Figura descargada de https://www.youtube.com/watch?v=xo9cKe_Fch8.

La App Store planteó un cambio radical en cuanto al tipo de software que se consumía en los dispositivos móviles, que poco a poco comenzaban a ganar mucho más poder de procesamiento y funcionalidades.

Rápidamente los otros sistemas operativos comenzaron a lanzar sus propias tiendas con distintos nombres, Marketplace para Windows Phone, Android Market (Conocido actualmente como Google Play Store) para Android y algunas otras como Amazon App Store, Apptoide, etc. las cuales perduran hasta hoy en día y siguen siendo las que más actividad generan después de la App Store.

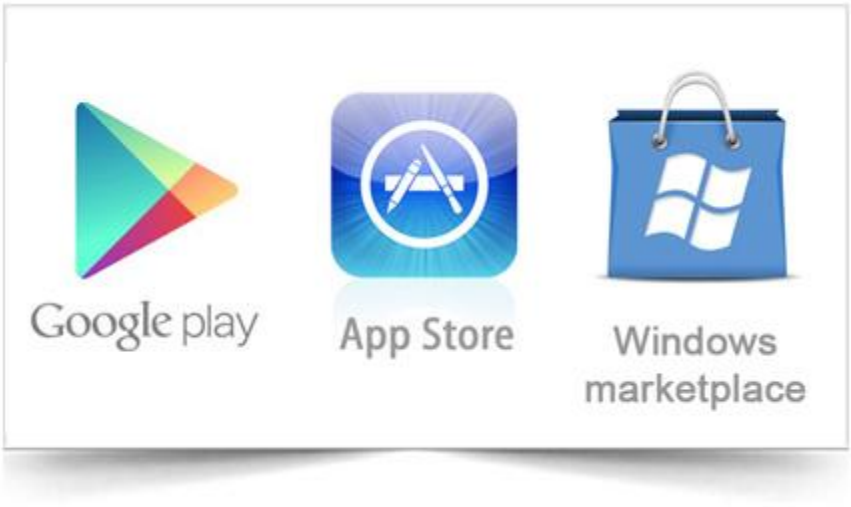

Figura 7 Google Play Store, Apps Store y Windows Marketplace, las principales tiendas de aplicaciones.

El impacto del concepto de las tiendas de aplicaciones fue brutal, pues plantearon nuevos modelos de negocios para las empresas desarrolladores de software, pero también para programadores independientes [9].

Ya que era posible vender aplicaciones y juegos, no solo de forma directa en una sola adquisición, sino que ofrecieron la capacidad de proveer servicios y contenido por suscripción o elementos consumibles (cómo vidas en videojuegos), además de que empresas de publicidad comenzaron a ofrecer la posibilidad de generar ingresos mostrando anuncios dentro de las aplicaciones según los criterios de comportamiento de un usuario.

En poco tiempo comenzaron a aparecer empresas que crearon una gran cantidad de aplicaciones nuevas, mientras que otras buscaron brindar sus productos y servicios a través de los dispositivos móviles. Además de que gracias a las tiendas de aplicaciones los programadores podían llegar a usuarios en distintos países que hablaran otros idiomas, algo que era muy difícil y costoso de alcanzar económicamente para la mayoría [11].

Figura 8 Algunas de las aplicaciones móviles más famosas, Figura descargada de: https://www.businessinsider.com/100-best-apps-for-iphone-and-android-2013-9.

Capítulo 2. Metodología del desarrollo

Es este capítulo explicaremos la metodología que se usará para el desarrollo de la aplicación, así como todos los puntos que debemos cumplir para considerar que la metodología se está aplicando correctamente.

2.1 El desarrollo ágil de software

El desarrollo ágil es una metodología de desarrollo de software que su principal característica es la filosofía de que los requerimientos van evolucionando, al mismo tiempo que el proyecto se va desarrollando, adaptándose a los problemas y riesgos que se van presentando, así como los posibles cambios que vaya solicitado el dueño del producto final. Por lo general estos dos puntos son autocontrolados de forma individual por cada uno de los miembros del equipo, pero cuando uno de estos supera sus capacidades individuales, estos deben ser solucionados de forma grupal, buscando un mejoramiento continuo del proyecto. Una de sus características más importantes del desarrollo ágil de software es entregar un producto funcional en un lapso establecido.

El concepto de desarrollo ágil nace como tal en el año 2001, cuando distintos ingenieros de software de renombre entre ellos Howard G. "Ward" Cunningham (creador de la primera Wiki y del concepto en sí) y Kent Beck (Quien participó en el desarrollo de Small Talk uno de los primeros lenguajes de programación orientado a objetos y creador de los frameworks xUnit para pruebas unitarias) se reunieron para discutir sus puntos de vista sobre las distintas metodologías de desarrollo de software, combinando características de metodologías como Programación Extrema (XP), Scrum, Proceso Unificado (UP), entre algunas otras, lo que dio como resultado la creación del “Manifiesto Ágil”, un documento que engloba todos los puntos que cualquier metodología debe cumplir para ser considerada como desarrollo ágil.

El “Manifiesto para desarrollo ágil de software” se rige por 12 principios los cuales son [14]:

1. La satisfacción del cliente debe ser asegurada enviando resultados de forma continua.
2. Los cambios en las reglas y requerimientos siempre son bienvenidos incluso cuando ocurren en el desarrollo tardío.
3. La entrega de software funcional debe ser frecuente (Semanal en lugar de Mensual).
4. Debe haber cooperación cercana y diarias entre los miembros del equipo de desarrollo y la gente de negocios.
5. Los proyectos son construidos alrededor de individuos motivados, que sean gente de confianza.
6. La conversación cara a cara es la mejor forma de comunicación para cualquier asunto relacionado al proyecto.
7. El software funcional es la principal forma medición de progreso.
8. Debe haber desarrollo sostenible, capaz de mantenerse estable y de forma constante.
9. Atención continua hacia la excelencia técnica y buen diseño.
10. La simplicidad es esencial.
11. Las mejores arquitecturas, requerimientos y diseños emergen desde equipo auto administrados.
12. Regularmente el equipo aprende de sí mismo a ser más efectivo conforme la experiencia ganada.

Además, se han establecido ciertas características adicionales para poder considerar un proyecto como desarrollo ágil, que no son verdades universales, sino que dependen del proyecto a desarrollar, el tamaño de los equipos de programadores y de la infraestructura con la que se cuente. Estos puntos extra se pueden definir como [17]:

- Las herramientas y procesos son importantes, pero es más importante tener gente competente trabajando de forma conjunta para desarrollar el proyecto.
- La buena documentación es útil para ayudar tanto a programadores y clientes a entender como el software está siendo construido y como usarlo, pero la finalidad es crear buen código, no documentación.
- Un contrato especificando las reglas de negocio son importantes pero la retroalimentación constante de lo que el cliente quiere siempre será mejor pues permitirá saber si el trabajo que se tiene es lo que realmente el cliente busca.
- Un plan de desarrollo es importante, pero debe estar diseñado para adaptase a los cambios tecnológicos que pueden presentarse, por ejemplo, la obsolescencia en el uso de ciertas tecnologías por otras nuevas, el caso más claro sería cuando las aplicaciones Android debían adaptarse a procesadores de 64 bits arm en lugar de 32 bits que fueron con lo que originalmente se anunció el sistema operativo.

2.2 Aplicación del modelo Extreme programming (XP)

La metodología de desarrollo elegida es Extreme programming (XP) pues es una metodología ágil que se adapta a desarrollos de software que son propensos a que los requerimientos cambien de forma constante [18], por ejemplo, cuando aún no se tiene una idea firme de qué y cómo debería ser un sistema o cuando las tecnologías cambian rápidamente cada cierto tiempo (un ejemplo claro de esto sería cada vez que es lanzada una nueva versión de un sistema operativo móvil).

La metodología Extreme programming (XP) puede se recomienda para desarrollos de software donde:

- Los requerimientos de software que cambian de forma constante.
- Las tecnologías utilizadas cambian en periodos relativamente cortos de tiempo.
- Los equipos de desarrollo son pequeños (de 2 a 12 programadores), pero incluso es aplicable para casos donde hay un solo programador.
- La tecnología utilizada permite la realización de pruebas unitarias de forma rápida y fácil.
- Sea posible trabajar de forma cercana con el cliente.

El siguiente diagrama muestra el flujo que se sigue cuando se trabaja con XP cómo metodología.

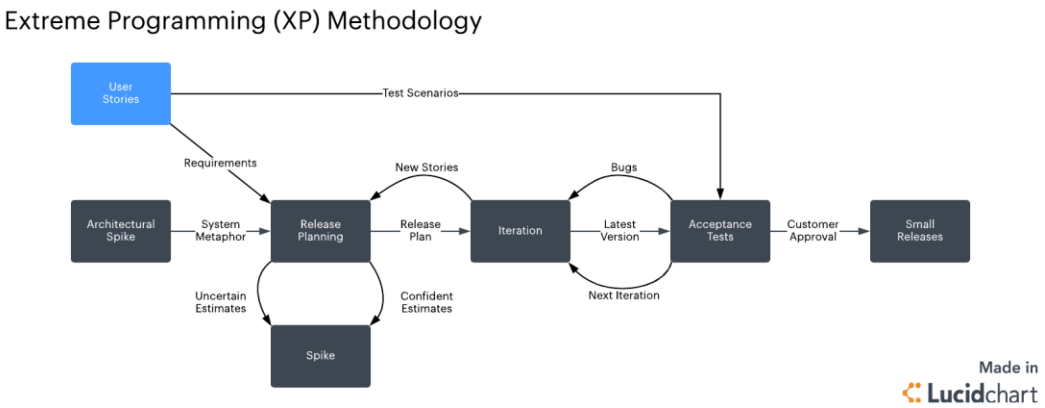

Figura 9 Flujo de trabajo de la metodología Extreme Programming (XP) Figura descargada de: https://www.lucidchart.com/blog/what-is-extreme-programming.

Uno de los principales objetivos de la metodología XP es entregar el software que es necesario cuando es necesario. Es decir, crear prototipos iterativos donde en cada iteración se vayan refinando hasta llegar a lo que el cliente desea, pero esto brinda una ventaja enorme respecto a otras metodologías y es que sí un requerimiento o tecnología cambia es posible actualizar o desechar el prototipo o funcionalidad sin afectar al software ya creado o entregado.

Los pasos para aplicar XP a un desarrollo son:

Planeación

Consiste en definir las historias de usuario, las cuales definen las funcionalidades que el cliente quisiera ver. Las historias no tienen que ser muy detalladas solo lo suficiente para cómo para crear prototipos o maquetas de la interfaz que ayuden al equipo de determinar que funcionalidades deben ser implementadas primero. Las planeaciones deben ser pensadas para entregables a corto plazo, ya que de caso contrario sí la planeación contempla la entrega de todo el sistema y se diera el caso de que un requerimiento cambiara o no fuera tecnológicamente posible, gran parte de la planeación ya no sería válida para el proyecto, se tendría que hacer una nueva y por tanto el desarrollo sufriría retraso.

Administración

El administrador del proyecto debe establecer todo para que la aplicación de la metodología tenga éxito. Es necesario que todos los miembros del equipo trabajen colaborativamente y que exista una comunicación efectiva para evitar problemas en el desarrollo derivados a la falta de estas cualidades.

El administrador es el encargado de definir la duración de las iteraciones, la velocidad de desarrollo (la cantidad de trabajo que debe ser hecho antes de que un entregable sea considerado terminado) y de redefinir la prioridad de las tareas para evitar cuellos de botella.

Diseño

La metodología Extreme Programming propone que los diseños tanto de código como de interfaces graficas deben centrarse en la simplicidad, pues un diseño simple pero funcional toma menos tiempo de implementar, es más fácil de refactorizar y previene que se realice trabajo en vano en caso de que una característica sea desechada.

Es recomendable realizar una investigación rápida sobre las posibles soluciones a un problema y elegir aquella que del mejor retorno de la inversión en el menor tiempo posible.

Codificación
Se recomienda elegir una convención de nombres para las clases, métodos, variables y paquetes que permitan diferenciarlos rápidamente, pero sobre todo para mantener un código legible y fácil de mantener.

De igual forma se recomienda utilizar un sistema de control de versiones, realizando "commits" cada cierta cantidad de horas o cuando una funcionalidad ha sido realizada y esta se considera estable.

Testing.
El equipo deberá realizar pruebas funcionales (ya sea manuales o automatizadas) así como unitarias de forma constante para detectar bugs rápidamente y solucionarlos antes de integrar o lanzar una nueva funcionalidad.

Por tanto, podemos decir que usar XP permite optimizar el tiempo de desarrollo de un sistema de software, pero sin perder la calidad de los entregables, reduce el riesgo de trabajar de más y se adapta perfectamente a equipos de programación pequeños.

Capítulo 3. Arquitectura

En este capítulo documentaremos el proceso de la definición y modelado de la arquitectura para la aplicación. Es decir, diseñaremos y planearemos las partes que compondrán la aplicación, así como la interacción que tendrá el usuario final. Con esto podremos comenzar a codificar teniendo un camino más claro de que es a lo que queremos llegar y evitaremos la posibilidad de crear componentes que se desechen dado que no son requeridos. También se especificará los mecanismos que utilizaremos para tener un código mantenible y escalable, pero sobre todo nos permitirá dar un paso atrás en caso de que cometamos un error crítico, estamos hablando del control de versiones.

3.1 Descripción de la aplicación

La aplicación será una herramienta pensada para usuarios que están bajo algún tratamiento médico que requiera la toma de medicación constante o recurrente y que tengan dificultades para llevar una administración manual de que medicamento tomar y cuando hacerlo, así también permitirá llevar registros de los medicamentos tomados con anterioridad.

Teniendo en cuenta que es muy probable que los usuarios que registren más medicamentos son personas con enfermedades crónicas o recurrentes es probable que también tengan que visitar distintos médicos o establecimientos relacionados a temas de salud (laboratorios y farmacias, por ejemplo), la aplicación deberá brindar un tipo de agenda y directorio para tener los datos de contacto de dichos lugares o médicos en una forma ordenada, concentrada y de fácil acceso. La aplicación deberá de notificar de alguna manera al usuario de que ha ocurrido o está por ocurrir un evento de importancia: La hora de toma de un medicamento o una consulta médica, por ejemplo.

De la descripción anterior podemos definir las siguientes reglas de negocio:

- El usuario puede registrar medicamentos por nombre comercial, nombre genérico, gramaje/presentación, forma farmacéutica (pastillas, jarabes, capsulas, etc.…) y opcionalmente una fotografía de la caja o medicamento físico, por último, se podrá especificar un color distintivo para fácil identificación por el usuario en otras partes de la aplicación.
- El usuario puede crear “tratamientos” lo cual es tomar un medicamento previamente registrado y crear el periodo de tiempo e intervalos en los que el medicamento será consumido, los tratamientos pueden estar activos, pausados o terminados.
- El usuario puede especificar sí desea recibir una notificación, una alarma o ninguna de estas cuando llegue la hora en la que se deba tomar un medicamento.
- El usuario puede visualizar los medicamentos que pueden ser tomados en el transcurso del día actual.
- El usuario puede elegir que acción tomar cuando se presenta la notificación de toma de medicamento: posponerlo, tomó el medicamento o descartar la toma.
- La aplicación contará con un directorio médico, contando con los campos: Titulo (Dr., Dra., Lic., etc.), nombre, especialidad e icono. Pudiendo añadir al menos una ficha de contacto con los datos: Titulo (consultorio particular, hospital, clínica, etc.) dirección, teléfono, celular, email y sitio web.

- Se incluirá un apartado al directorio para registrar establecimientos que puedan ser útiles para el usuario como farmacias, laboratorios de análisis clínicos, laboratorios de imagen, etc., con los campos de nombre establecimiento, dirección, teléfono, email y sitio web.
- El usuario podrá consultar las fichas de los médicos y establecimientos de forma individual, cada campo presentado realizará una acción según el contexto (enviar un email, abrir la aplicación de teléfono para realizar una llamada a dicho establecimiento, abrir la aplicación de mapas indicando la dirección, etc.)
- Se podrán crear recordatorios a citas o eventos médicos (consulta médica, toma de muestras, etc.), cada cita incluirá título, médico o lugar, una nota para indicaciones, fecha y hora, tipo de recordatorio (notificación o alarma), un color distintivo y la dirección/ubicación de la cita.
- Los datos guardados se mostrarán en listas según el tipo de "objeto" creado, pudiendo realizar búsquedas en estas listas según los parámetros del usuario.
- Hacer un toque en un elemento de las listas anteriores mostrarán los detalles individuales del elemento seleccionado.
- Se tendrá un apartado de configuraciones que permitan establecer opciones globales y por default en distintos apartados y procesos de la aplicación.
- La aplicación deberá tener soporte para distintos usuarios y brindar mecanismos de seguridad para evitar que un usuario modifique los datos creados por otro usuario.

Con estas reglas de negocio establecidas podemos comenzar a modelar los comportamientos esperados al usar la aplicación, la base de datos y los prototipos de la interfaz gráfica. Dada la metodología de desarrollo elegida puede que las funcionalidades cambien en el proceso de codificación con el paso del tiempo, esto no debe ser un problema siembre y cuando se cumplan con los requerimientos especificados en las reglas de negocio.

3.2 Diagramas de secuencia

Utilizaremos los diagramas de secuencia para mostrar la interacción entre sí de los distintos objetos o módulos de nuestra aplicación a través del tiempo [13]. Especificando cada uno de los eventos o mensajes que permiten una interacción de un objeto a otro. Nuestros diagramas de secuencia serán modelados desde el punto de vista del usuario final tratando de reducir lo más posible el aspecto técnico, esto para hacer más comprensible los pasos que se siguen al usar nuestra aplicación.

Primera Ejecución:

La primera ejecución de la aplicación es un proceso importante y único que ocurre cuando la aplicación es instalada en su versión final en un dispositivo de un usuario desde la Google Play Store, pues establece el entorno interno que la aplicación requiere para funcionar correctamente, este proceso es semi automático, pues requiere interacción por parte del usuario.

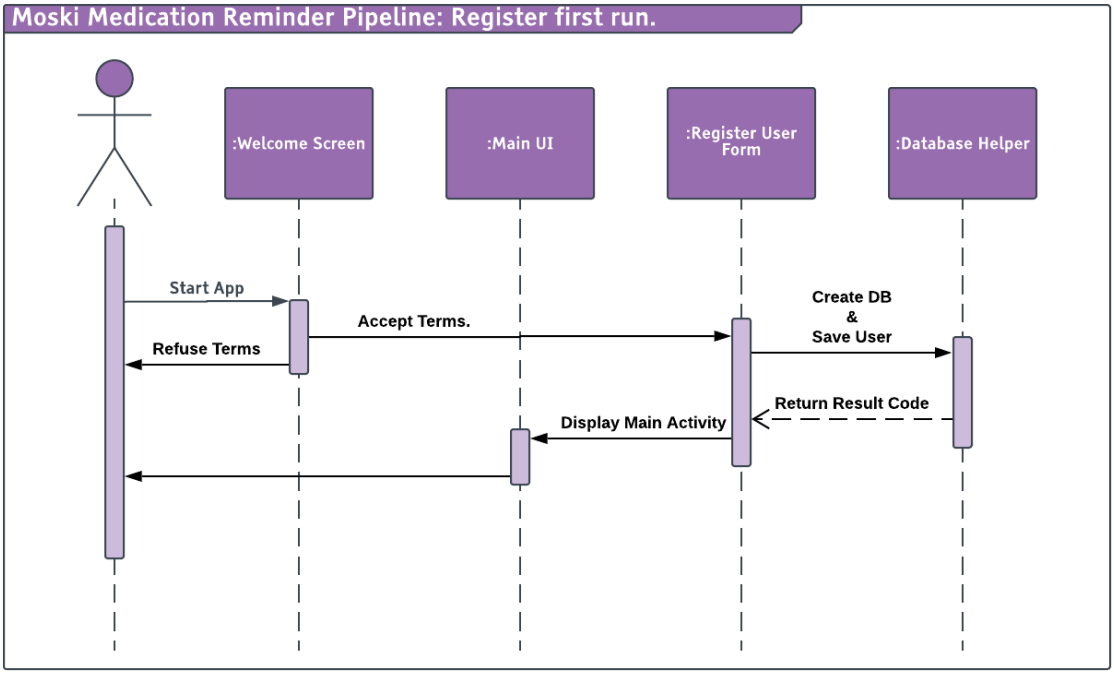

Figura 10 Diagrama de secuencia de la primera ejecución.

El proceso es el siguiente: El usuario abre la aplicación e inmediatamente ve una pantalla de bienvenida, la cual mostrará un acceso a los términos y condiciones de uso de la aplicación, las cuales deben ser aceptadas para continuar. Este es un apartado muy importante, pues Google requiere que toda aplicación que esté publicada en la Play Store muestre los términos y condiciones explícitamente al usuario, en caso contrario la aplicación es rechazada o puede ser dada de baja.

Posterior a esto se mostrará el formulario para la creación del primer usuario dentro de la base de datos y el usuario "activo" dentro de la aplicación, es decir que todos los datos que se muestren dentro de la aplicación serán de este usuario hasta que se cree otro usuario y se establezca como activo este último.

Una vez completado el proceso se navegará a la actividad principal y se mostrará un mensaje según sea el resultado. En caso de que haya habido un problema se seguirá en el formulario de registro y se notificará el fallo.

Registro de elementos.

Una vez creado el entorno inicial debemos especificar el proceso para el registro de los elementos (Medicamentos, Citas Médicas, Médicos, etc.…) en la base de datos. Cabe aclarar que este proceso debemos tomarlo como “general”, pero los elementos especializados que tengan muchas dependencias o procesos más específicos serán detallados más adelante.

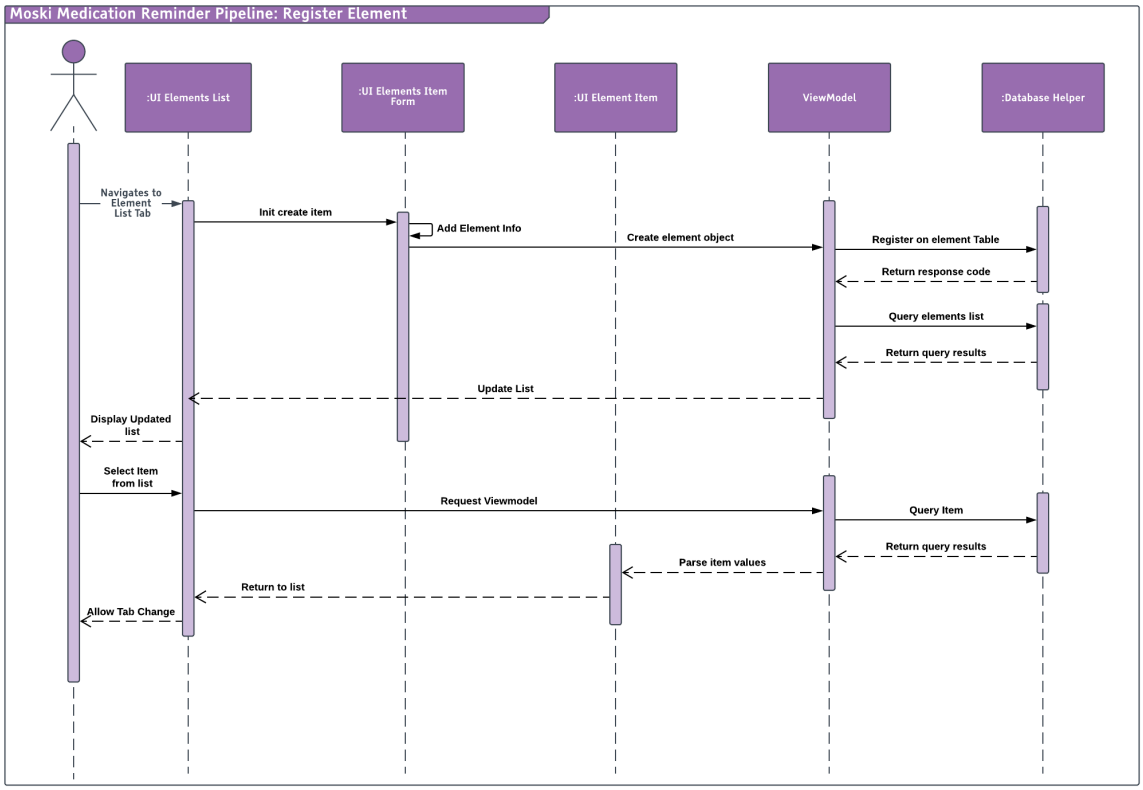

Figura 11 Diagrama de secuencia para el registro de elementos.

El usuario activo navega hasta la sección que representa el listado o colección de un elemento, posteriormente inicializa el formulario para la creación de ese elemento. Una vez llenado el formulario se procede a crear las estructuras para el “View Model” (La cual es una clase especial para brindar una interfaz para compartir y consumir datos de forma encapsulada), para posteriormente registrarlos en la base de datos. En este último paso se registran los eventos que el sistema operativo puede requerir para el uso en otras funciones dentro de la aplicación.

En todos los pasos se espera un código de respuestas que nos informará si el resultado del proceso fue correcto u ocurrió un falló. Una vez que se ha guardado el elemento la aplicación regresa a la lista de Elementos actualizada incluyendo al nuevo objeto.

El usuario puede hacer un toque en cualquiera de estos elementos y se realizará una consulta a la base de datos para construir la actividad del elemento individual y mostrarla al usuario con los respectivos datos. El usuario puede dejar esta actividad y regresar a la lista de elementos.

Cambiar de Usuario Activo.

La aplicación deberá brindar soporte para su uso por varias personas desde una misma instalación en un Smartphone o Tablet, pero es necesario tener separados los elementos que cada uno de estos usuarios generan para prevenir que se confundan al saber que elemento corresponde a que usuario, por ejemplo ¿qué pasa si dos usuarios registran el mismo medicamento, pero bajo diferente tratamiento?

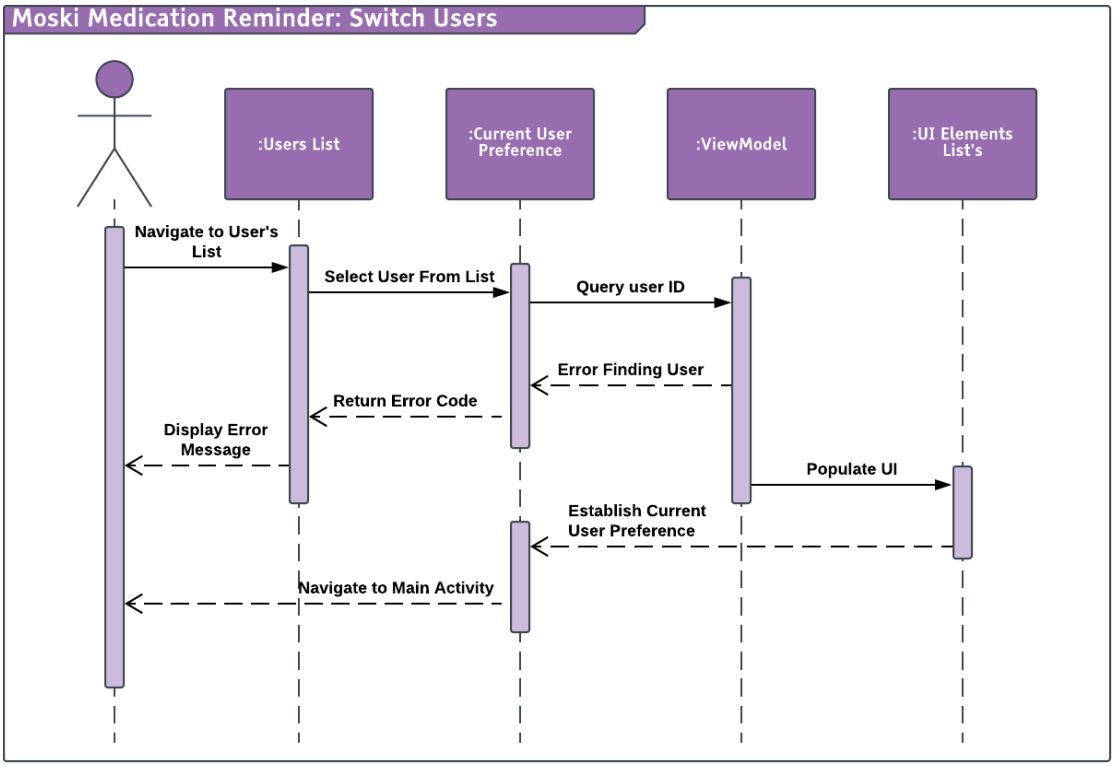

Figura 12 Diagrama de secuencia para cambiar de usuario dentro de la aplicación.

El proceso consiste en que el usuario activo navegue hasta la lista de usuarios registrados, el usuario toca en cualquiera de los elementos de esta lista, se solicita el ID del nuevo usuario activo. En caso de haber un error se retorna un código de estado para construir un mensaje para que se notifique al usuario de lo ocurrido.

En caso contrario se comienzan a realizar consultas para poblar las listas y todos los otros elementos de la interfaz gráfica del actual usuario activo. Posteriormente se establece una “Preferencia” que es una de configuración del SDK de Android para guardar información de forma persistente y que es accesible desde cualquier parte dentro de la aplicación, este caso para establecer el ID del usuario actual y usarlo desde las actividades de registro de nuevos elementos y vincularlos a dicho usuario. Finalmente, la aplicación retorna a la pantalla principal con todos los cambios realizados.

Registro de médicos.

El registro de los médicos dentro de la aplicación es un proceso un poco distinto y es que el usuario puede añadir distintas “fichas de contacto” los cuales son lugares donde un médico ofrece sus servicios como un hospital, clínica o consultorio particular, cada uno con sus respectivos datos como dirección y número de teléfono.

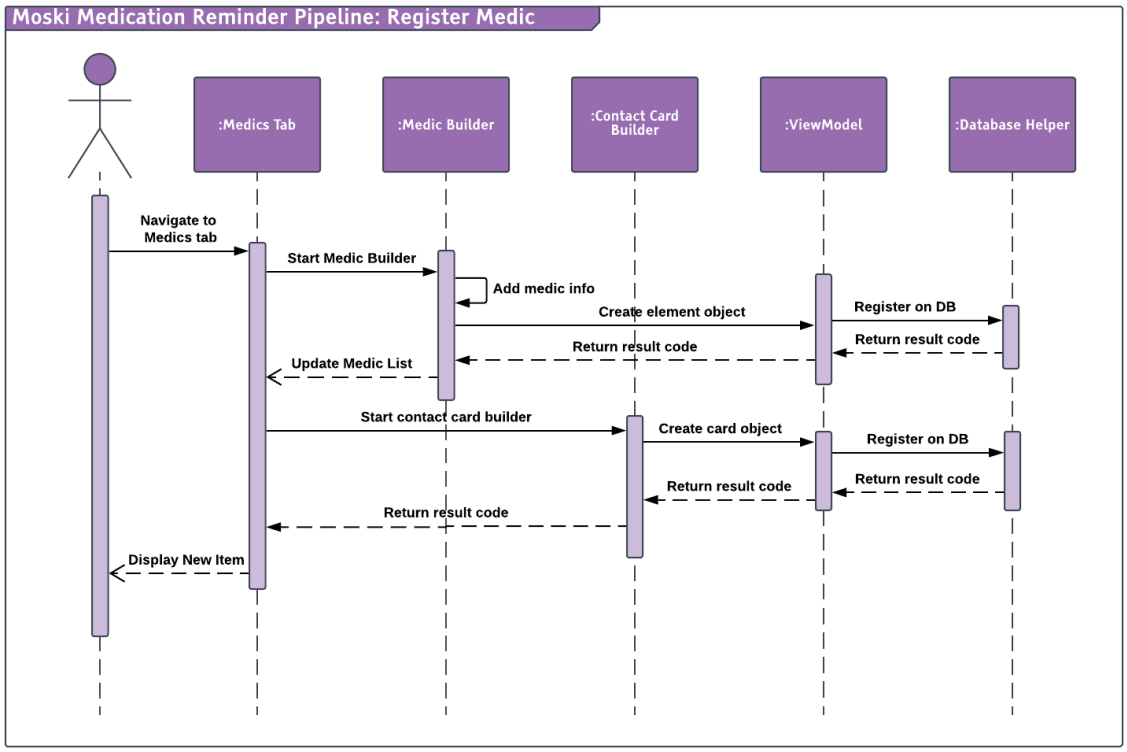

Figura 13 Diagrama de secuencia para registro de un médico en la aplicación.

El usuario navega hasta la pestaña de la lista de médicos posteriormente inicializa el formulario de registro, añadirá la información de médico como su nombre y especialidad, se registra en la base de datos y posteriormente agrega una ficha de contacto a la cual añade información de contacto (titulo, dirección, número de teléfono, celular, etc. el usuario puede añadir tantas fichas de contacto como desea registrando una ficha de contacto a la vez en la base de datos. En ambos pasos se retorna un código de respuesta. Sí el proceso no tuvo mayores complicaciones se retorna a la lista de médicos con los datos actualizados.

Registro de citas médicas.

El usuario puede registrar distintas citas, ya sean a médicos, establecimientos o ingresando información nueva usando en el formulario, la cual tendrá la opción de establecer alarmas o notificaciones para dicho recordatorio.

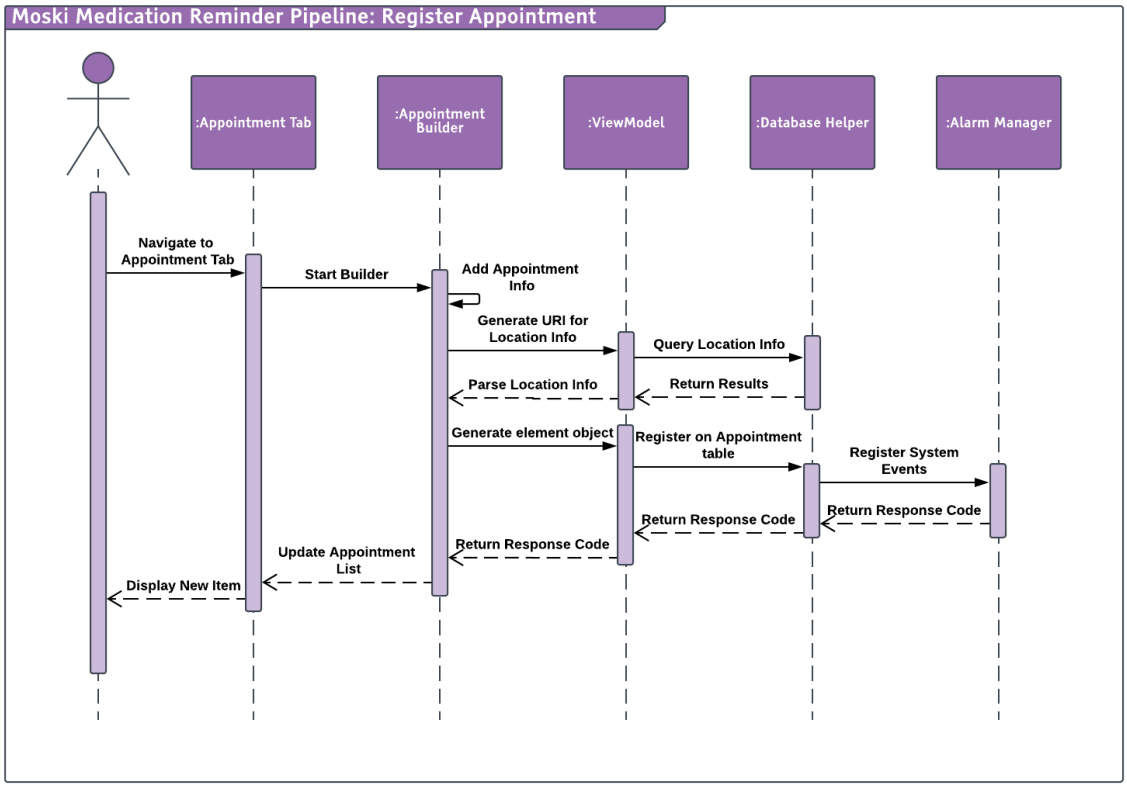

Figura 14 Diagrama de secuencia para el registro de una nueva cita.

El usuario navega hasta la lista de citas e inicializa el formulario, añade información y usando el ViewModel recupera la información de ubicación desde los médicos o establecimientos registrados, dando la opción de ingresar una dirección y ubicación nueva.

Se genera el objecto de esta información y se almacena en la base de datos, entre los datos de registro se incluye fecha y hora, la cual es retomada de la base de datos y usada para generar los eventos del sistema para crear las notificaciones cuando llegue dicho momento. Se retornan códigos de estado en cada paso del proceso para saber los resultados de cada uno de estos. Al finalizar si todo salido bien se retorna a la lista de citas con los datos actualizados.

Registro de tratamientos.

Por último, explicaremos el proceso para el registro de tratamientos médicos, la piedra angular de nuestra aplicación y quizás el proceso más complejo pues requieres de dos elementos distintos: Medicamentos y Tomas.

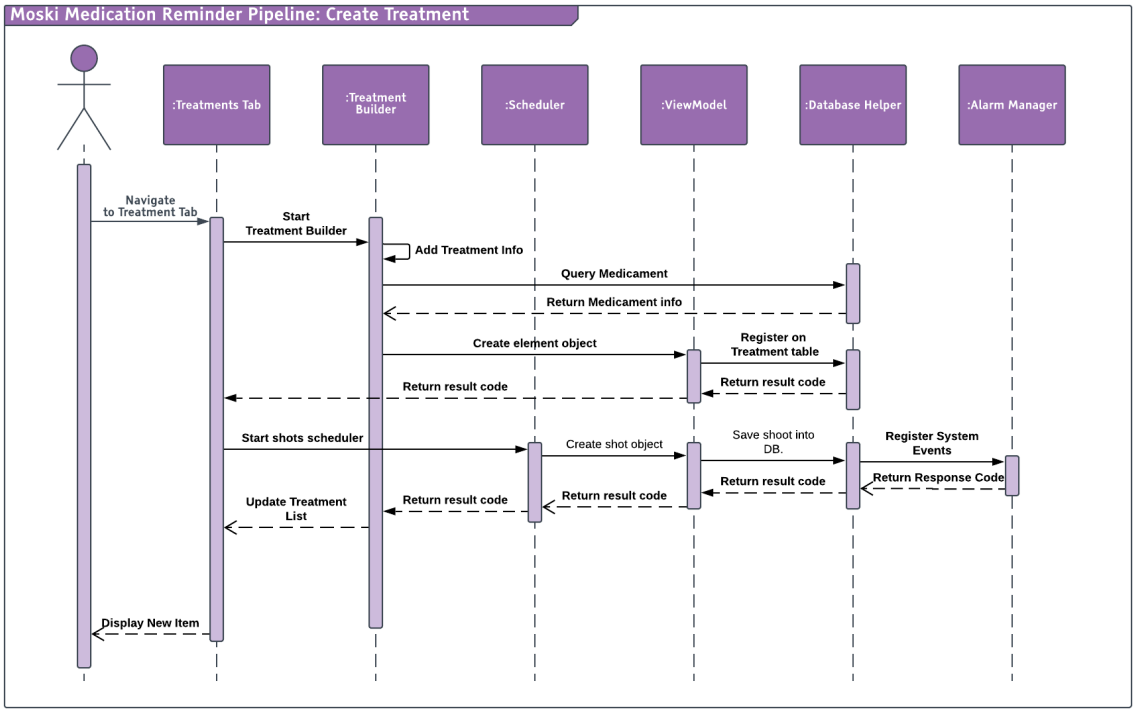

Figura 15 Diagrama de secuencia para el registro de un tratamiento médico.

El usuario navega hasta la pestaña de tratamientos e inicializa el formulario de creación de tratamientos, comienza a añadir información sobre el nuevo tratamiento cómo fecha de inicio, tipo de tratamiento y medicamento solo por mencionar algunos campos y lo almacena en la base de datos, posteriormente el usuario puede comenzar a crear “tomas” que serán las horas del día en los medicamentos deben ser tomados, pudiendo usar una notificación o alarma para establecer los recordatorios.

Con los diagramas de secuencia ya tenemos mucho más clara los elementos y secciones con los que funcionará la aplicación por lo que ya podemos comenzar a crear el primer prototipo del modelo de la base de datos, así como las maquetas y secuencias de navegación que seguirá un usuario normal.

3.3 Modelado de la base de datos

Con las reglas de negocio y los diagramas de secuencia podemos comenzar a diseñar nuestra base de datos. La base de datos nos permitirá definir cuáles serán las secciones que tendrá nuestra aplicación, pues usualmente cada tabla suele representar una sección en concreto dentro de la aplicación.

La base de datos también puede ser utilizada para diseñar las interfaces de usuario, ya que muchas veces una tabla o entidad en la DB suele ser una plantilla exacta de la información que se le mostrará al usuario y viceversa, un formulario mostrado al usuario puede llenar en su totalidad una tupla dentro de una tabla en la BD.

A continuación, mostraremos la versión preliminar de nuestro diagrama entidad relación de la base de datos de la aplicación. El modelo está hecho usando MySQL Workbench que como su nombre indica está enfocado a MySQL, Android utiliza el motor de bases de datos SQLite, el cual tiene una sintaxis similar, pero los principales cambios son los tipos primitivos que maneja [12]. De cualquier forma, el diagrama será usado para codificar la base de datos en una estructura que SQLite y Android puedan operar como es que se espera.

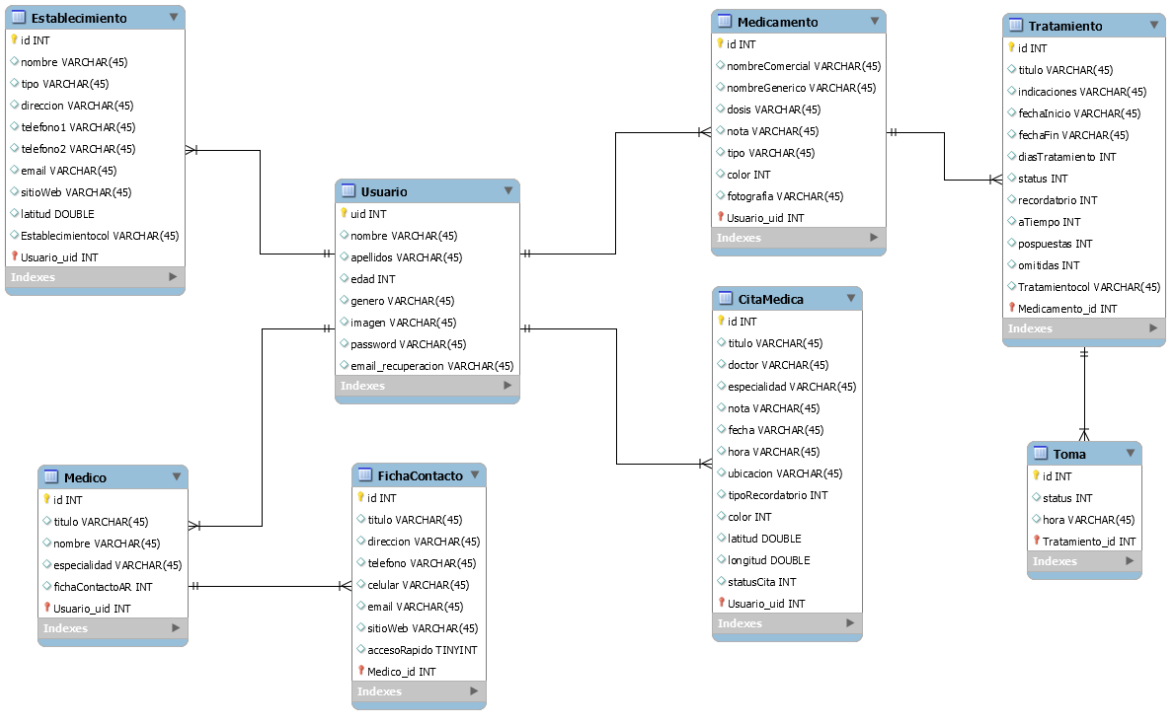

Figura 16 Esquema de la base de datos.

Las tablas y atributos son:

- **Usuario:**
  Representa a un usuario único dentro de la aplicación, cada usuario tendrá su propio "espacio" siendo independiente, el id del usuario será usado como llave foránea en otras tablas y que por tanto permitirán identificar los medicamentos, tratamientos, médicos y todo otro elemento individual que ese usuario haya registrado. La tabla "usuario" contendrá los atributos: nombre, apellidos, edad, género, password, imagen y un email de

recuperación, los primeros dos campos serán siempre obligatorios y el género tendrá siempre un valor dado el widget que usaremos para llenarlo, pero la edad será opcional y el campo password en caso de ser llenado será usado para limitar el acceso a la aplicación, sirviendo como mecanismo de seguridad para cambiar de perfil, en caso de utilizar password es necesario incluir un email de recuperación al cual se enviará un mensaje con el password sí es que llegará a ser olvidado. Finalmente, el usuario puede añadir una fotografía a su perfil para fácil identificación.

- **Medico:**
  Esta tabla representa a un médico, conteniendo los atributos: nombre, titulo, especialidad, icono.
- **FichaContacto:**
  Cada registro en la tabla FichaContacto estará ligada a un médico, pues representará una dirección física donde dicho medico ofrece consultas o alguno de sus servicios. Los campos serán: titulo, dirección, teléfono, celular, email, sitio web y acceso rápido. Los campos serán usados para brindar una acción contextual es decir sí el usuario toca el sitio web se abrirá el navegador o la aplicación de teléfono si se toca el número de teléfono.
- **Citas:**
  La tabla citas contendrá los campos título, dirección, fecha-hora, nota, latitud, longitud y color. Será usada para agendar citas con médicos, análisis clínicos u otro establecimiento, pudiendo disparar una notificación al usuario.
- **Establecimiento:**
  Esta tabla será usada para almacenar datos de contacto de establecimientos que puedan ser útiles para el usuario: farmacias, laboratorios, etc. Contendrá los campos nombre, tipo, dirección, teléfono 1, teléfono 2, email, sitio web, latitud y longitud.
- **Medicamento:**
  El medicamento será uno de los elementos más importantes de la aplicación, pues es que muchos de los otros registros giraran en torno a lo que se inserte en un registro de esta tabla. Los campos serán: nombre comercial, nombre genérico, dosis, indicaciones/nota, fotografía, icono, color y tipo (tabletas, jarabes, inyecciones, etc.).
- **Tratamiento**:
  Esta tabla llevará un registro de los tratamientos registrados, especificando el medicamento que se administrará, título (el motivo o causa), la fecha de inicio, la fecha de fin, los días, el estatus (activos, pausado, suspendido, terminado), el tipo de notificación, indicaciones y el tipo de continuidad del tratamiento (temporal o continuo).
- **Toma**:
  Esta tabla representará la toma de un medicamento a causa de un tratamiento previamente existente, estará vinculado a un usuario y un medicamento, contando con los atributos de: estatus (tomado, aún no tomado, pospuesto y saltado) y el atributo hora-fecha que deberá siempre estar dentro del intervalo establecido por su respectivo tratamiento.

Este diagrama cubre con nuestros requerimientos iniciales, pero es probable que llegue a cambiar para cuando comencemos con el desarrollo de la aplicación, pero notificaremos estos cambios en su momento sí es que llegan a ocurrir.

La base se encuentra normalizada hasta la tercera forma normal (3FN), ya que cumple con las características:

- Las entidades se encuentran en la segunda forma normal (2NF).
- Ningún atributo no-primario de la tabla es dependiente transitivamente de una clave primaria, un atributo no-primario es un atributo que no pertenece a ninguna clave candidata.

Gracias a la aplicación de la 3FN eliminamos las anomalías en la captura de datos, reducimos la necesidad de reestructurar la base de datos en cuanto esta comienza a ser utilizadas, hace el modelo de datos más informativo y permite realizar consultas más eficientes.

3.4 El modelo MVVM y sus beneficios

Durante los primeros años de Android, el patrón de arquitectura de software estándar para el desarrollo de aplicaciones móviles era el MVC (Model-View-Controller o Modelo-Vista-Controlador) [20] pues se adaptaba perfectamente a la manera en la que Android está diseñado:

Vista: Las vistas, en el caso de Android serían todo lo relacionado a la interfaz de usuario que serían los archivos XML que contienen los layouts y controles (botones, etiquetas, spinners, etc.)

Controlador: La actividad que está unida a una vista para darle contexto y responder a los eventos del usuario (toques en pantalla) así como para actualizar los elementos de la interfaz.

Modelo: Las clases que implementan la lógica de la aplicación y que puede esta fuera de la actividad, por ejemplo, los sistemas de bases de datos, proveedores de contenido, escuchas de eventos del sistema, etc...

Durante mucho tiempo la gran mayoría de aplicaciones constaban de una o pocas actividades que realizaban una sola función en específico y que se actualizaban con una acción del usuario. Esto limitaba mucho las funcionalidades que una aplicación podía tener, pues la constante interacción con el usuario para poder actualizar las vistas hacía que estas aplicaciones no fueran tan versátiles para consultar datos en tiempo real.

Las aplicaciones poco a poco comenzaron a realizar funciones más complejas que requerían interfaces graficas complejas y responsivas, así también el hardware evolucionaba para permitir esta mayor complejidad.

Pero con cada actualización del sistema operativo Android se han ido añadiendo características que permiten una manipulación más versátil de los datos contenidos en la aplicación, sistemas de notificaciones push, consumo de servicios web, interfaces graficas dinámicas, etc. pero pronto Google y muchos de los programadores Android experimentados se daban cuenta que el patrón MVC comenzaba a complicar más el desarrollo de aplicaciones que facilitarlo, pues se tenían pocas clases con una gran cantidad de líneas de código que eran difíciles de mantener y muchas veces se tenían que crear varios archivos XML que representaban la misma interfaz gráfica, pero para distintos tamaños, orientaciones y resoluciones de pantalla.

Google propone como solución la implementación oficial del patrón MVVM (Model View View-Model o en español Modelo-Vista-Modelo de Vista) [19], en el cual la interfaz gráfica (La vista)

interactúa directamente con el usuario recibiendo las entradas y actualizándose cuando es necesario haciendo uso del View Model, el cual es un componente intermedio entre la vista y el modelo, pues contiene la lógica para mostrar los datos en la vista a modo de observador observando de manera bidireccional donde se cambian los datos y notificándolo a su contraparte (Sí la vista cambia se actualiza el modelo y viceversa). El modelo sigue siendo la lógica de la aplicación de igual manera que en el MVC.

Entre las ventajas que tiene el MVVM sobre el MVC están:

- El nivel de interactividad es superior, por tanto, las aplicaciones pueden cambiar de forma dinámica sin necesidad de tener interacción por parte del usuario.
- Al añadir un ViewModel se añade una capa adicional que expone únicamente los datos para actualizar la UI, por lo que se añade más seguridad para el usuario.
- Permite generalizar el código, el modelo puede ser código fuente independiente de la plataforma y es posible crear ViewModels para distintos objetivos: Apps móviles, páginas web, software de escritorio, etc.… Separando la lógica de negocio y datos de la capa de presentación. Por tanto, se tienen más componentes, pero son mucho más independientes que en el MVC, podríamos cambiar la vista sin afectar la lógica y viceversa.

Google añadió una capa adicional al MVVM cuando es aplicado a Android y es el Repositorio, la cual es una capa que permite decidir de donde serán consumidos los datos y lógica, es decir es posible tener varios modelos y elegir bajo ciertas condiciones cual se va a utilizar. El diagrama inferior muestra el uso del patrón MVVM en Android y de la aplicación de la capa repositorio.

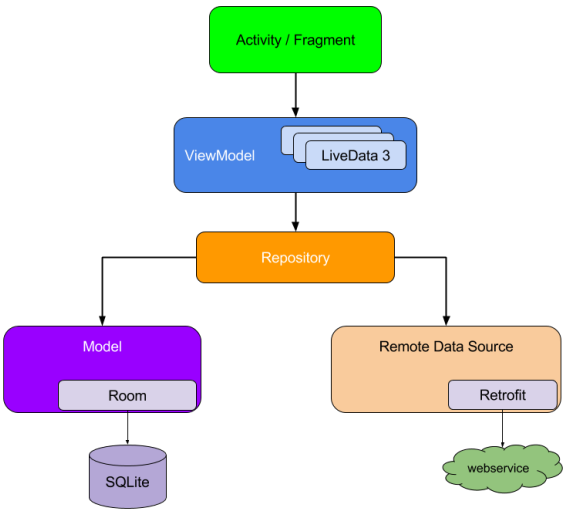

Figura 17 El patrón MVVM en Android Figura descargada de: https://developer.android.com/jetpack/docs/guide.

Podemos ejemplificar mejor su uso con caso real: Una aplicación móvil puede tener más una fuente de datos, digamos que cada vez que la app es abierta se descargan los últimos datos desde un servidor web con una consulta REST, estos datos son guardados en una base de datos local, para que en caso de que el usuario vuelva abrir la aplicación pero no cuente con conexión a internet estos son cargados desde la base de datos, sí la conexión a internet regresa entonces se vuelve a cambiar el repositorio a la fuente remota.

La aplicación de MVVM en una aplicación Android es bastante sencillo pues Google libero en mayo del 2018 Android Jetpack, un conjunto de librerías y guías de diseño que buscan acelerar el desarrollo de aplicaciones permitiendo la fácil aplicación del patrón MVVM, así como de reducir la fragmentación de sistema operativo, es decir codifica una sola vez y despliega en la mayor cantidad de versiones de Android posibles. Utilizaremos estas librerías en el desarrollo de la aplicación para facilitar su creación y reducir el tiempo de trabajo.

3.5 "Story boards" y maquetas de interfaz de usuario

Una de las principales ventajas de Android es que es posible generar interfaces graficas complejas en poco tiempo y esfuerzo, esto nos permite poder crear "maquetas", es decir solo el front-end sin comportamiento lógico que mostrarán el aspecto que puede tener la aplicación y dejándonos decidir cambios antes de comenzar a trabajar en la lógica de cada una de las pantallas o actividades.

Las maquetas de las pantallas nos permitirán crear "Story boards", es decir podremos definir y mostrar el camino que tendrá que realizar un usuario hasta llegar a cierto punto de la aplicación y el camino de regreso usando el botón back del sistema.

Comenzaremos creando un diagrama de estados, el cual usaremos como guía inicial para definir la navegación que el usuario puede seguir para utilizar las secciones core de la aplicación, de momento no representaremos la actividad para seleccionar una ubicación en un mapa, tomar una fotografía, configuración de la aplicación y la pantalla de bienvenida de la primera ejecución, pues de momento consideraremos estas funciones como parte de otras actividades sobre todo las de registro.

Aclaramos siguiendo nuestro diagrama, el usuario puede pasar de un estado a otro (cada estado representa una actividad dentro de la aplicación) por medio de toques dentro de la interfaz gráfica y puede regresar en el estado anterior usando el botón back del dispositivo o el que se incluirá en la barra de navegación superior de cada actividad.

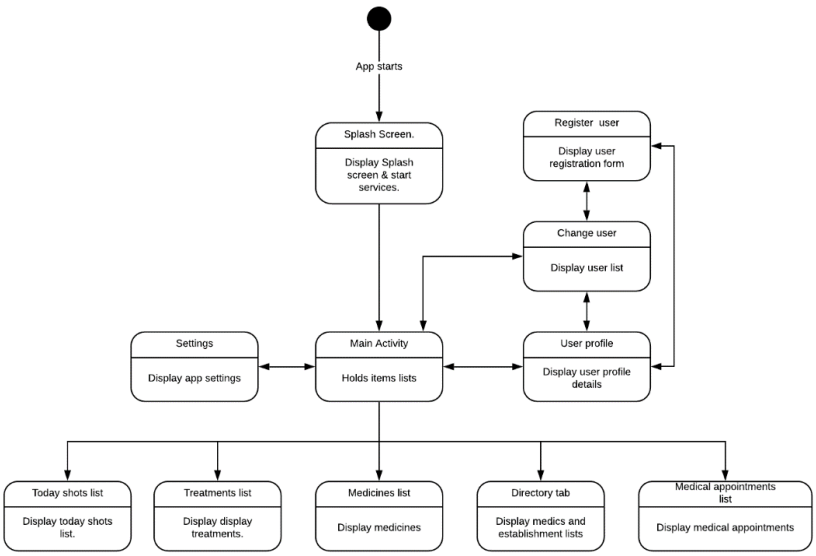

Figura 18 Diagrama de estados/actividades de la aplicación.

En la Figura 18 observamos que la aplicación comienza con una pantalla de presentación o splash screen, la cual es mostrada mientras la aplicación carga en el dispositivo, inmediatamente después se presenta la actividad principal, está contiene acceso a todas las listas de elementos, tomas, tratamientos, medicamentos, directorio médico, directorio de establecimientos y citas médicas, pero también podrá acceder a las configuraciones de la aplicación, el listado de usuarios y a los detalles de perfil del usuario activo actual. Desde el listado de usuarios será posible cambiar de usuario activo o de registrar uno nuevo.

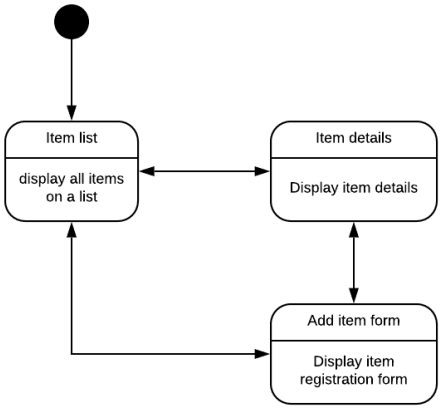

Figura 19 Diagrama de estados para un elemento.

La Figura 19 muestra el diagrama de estados para un elemento genérico (tratamiento, medico, medicamento o cualquier otro), donde buscamos que los procesos de registro y visualización sean los mismos para todos: El usuario navega a una lista de elementos del mismo tipo, pudiendo registrar uno nuevo usando un formulario o visualizar alguno ya existente tocando su elemento de la lista, pero será posible editar dicho elemento desde su vista individual reutilizando el formulario de registro.

Con estos diagramas podemos comenzar a crear los primeros prototipos de la interfaz gráfica teniendo una idea mucho más clara de lo que queremos lograr y descubrir sí es que el modo correcto y en caso de que no corregirlo antes de tener mucho más código fuente con dependencias funcionales.

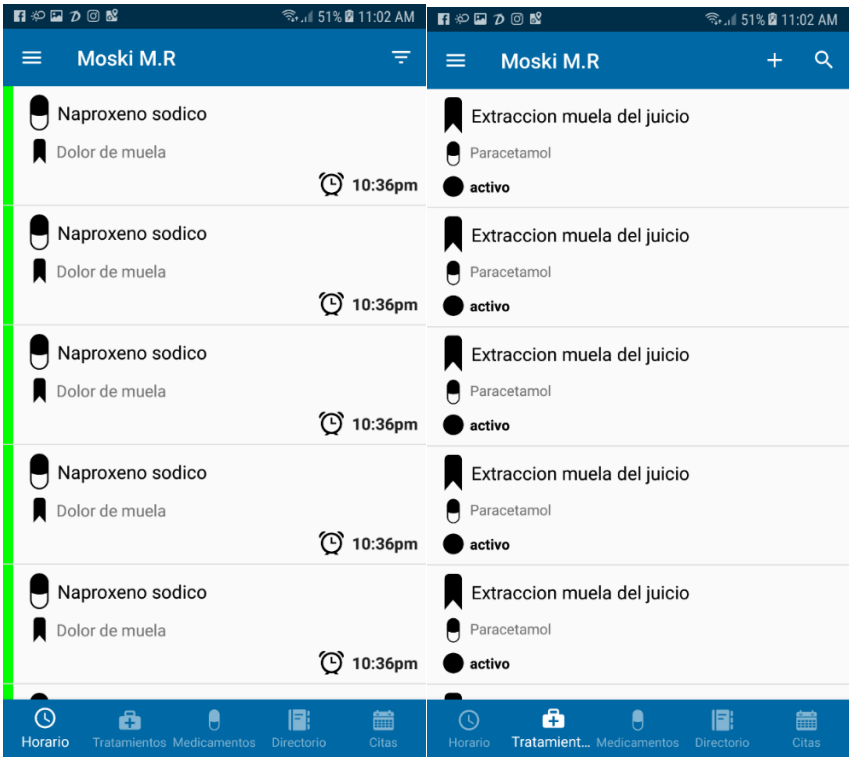

Figura 20 Pantalla principal de la aplicación mostrando la pestaña horario y pestaña tratamiento respectivamente.

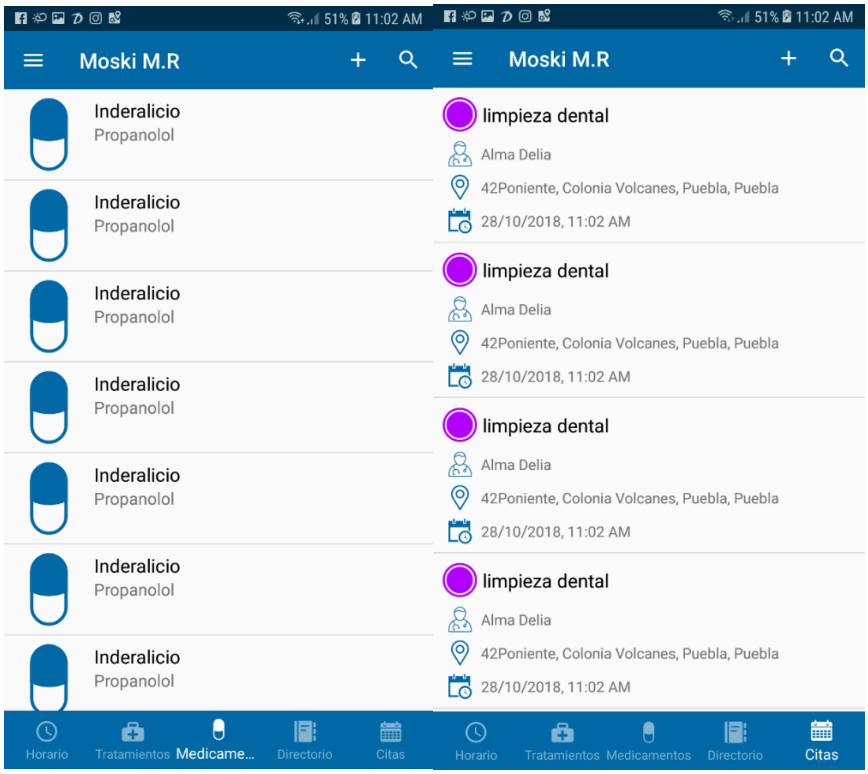

Figura 21 Pantalla principal de la aplicación mostrando la pestaña de medicamentos y la pestaña de citas médicas.

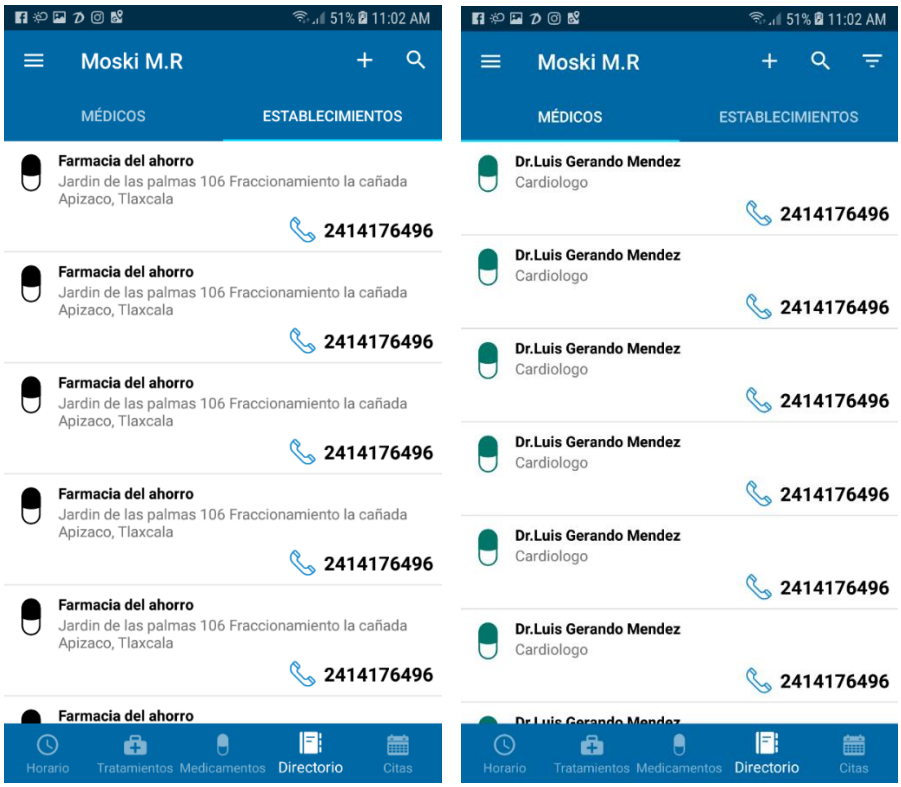

Figura 22 Pantalla principal de la aplicación, mostrando la pestaña de directorio con sus dos apartados: Médicos y Establecimientos.

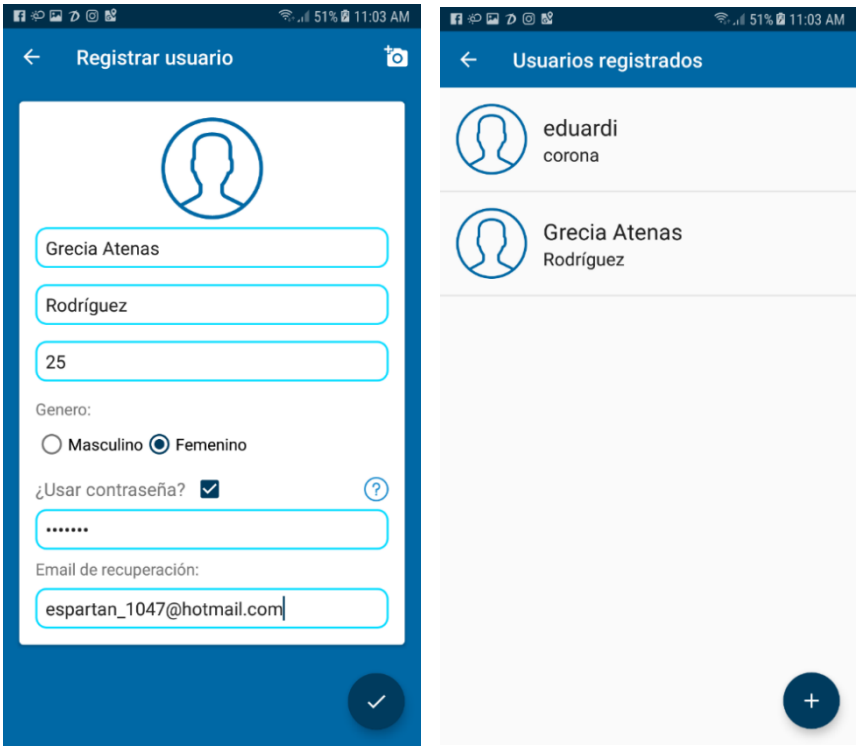

Figura 23 Formulario de registro y listado de usuarios.

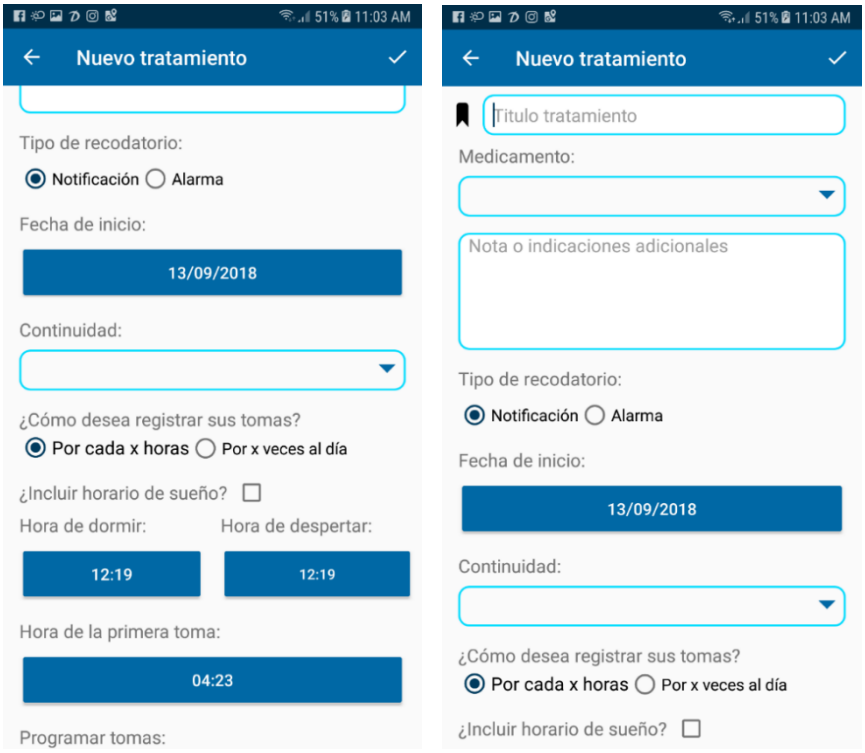

Figura 24 Detalles del formulario de registro de un nuevo tratamiento.

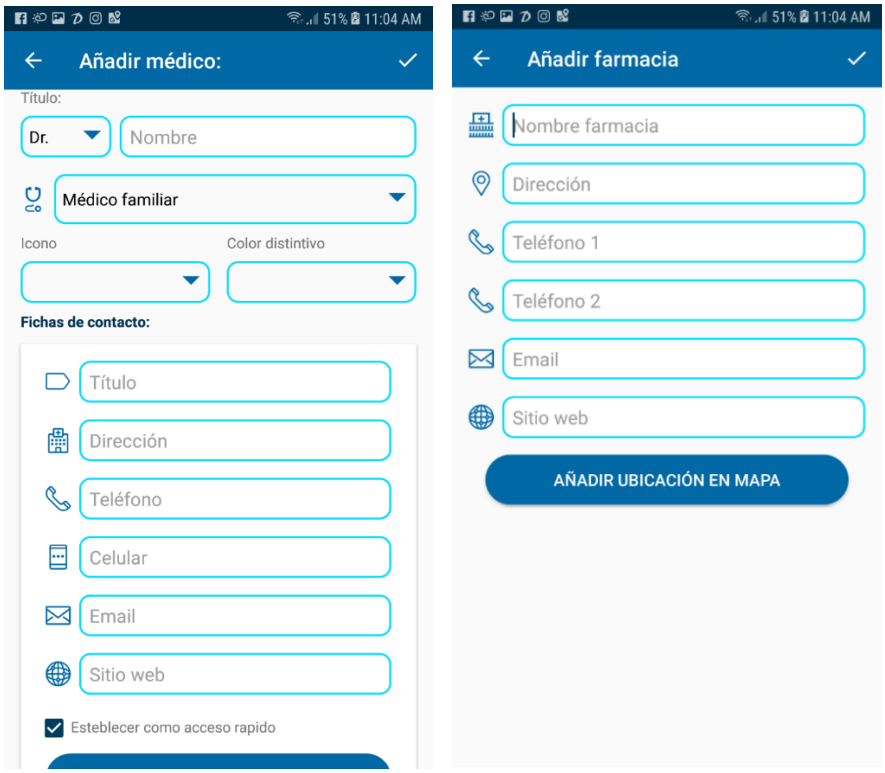

Figura 25 Formularios de registro para un médico y una farmacia (establecimiento).

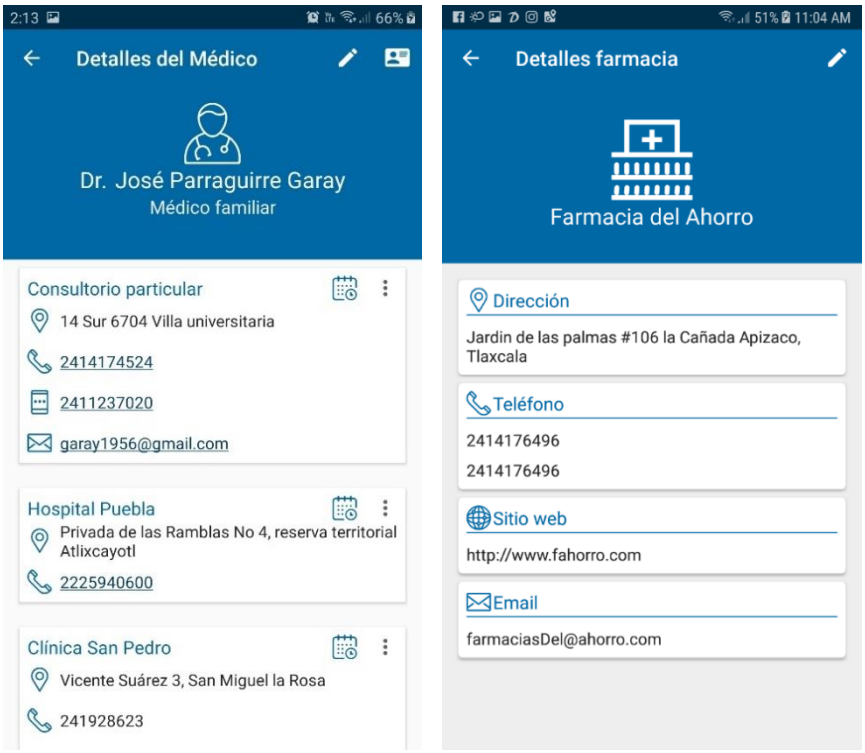

Figura 26 Pantallas para detalles de Un médico y un establecimiento registrado.

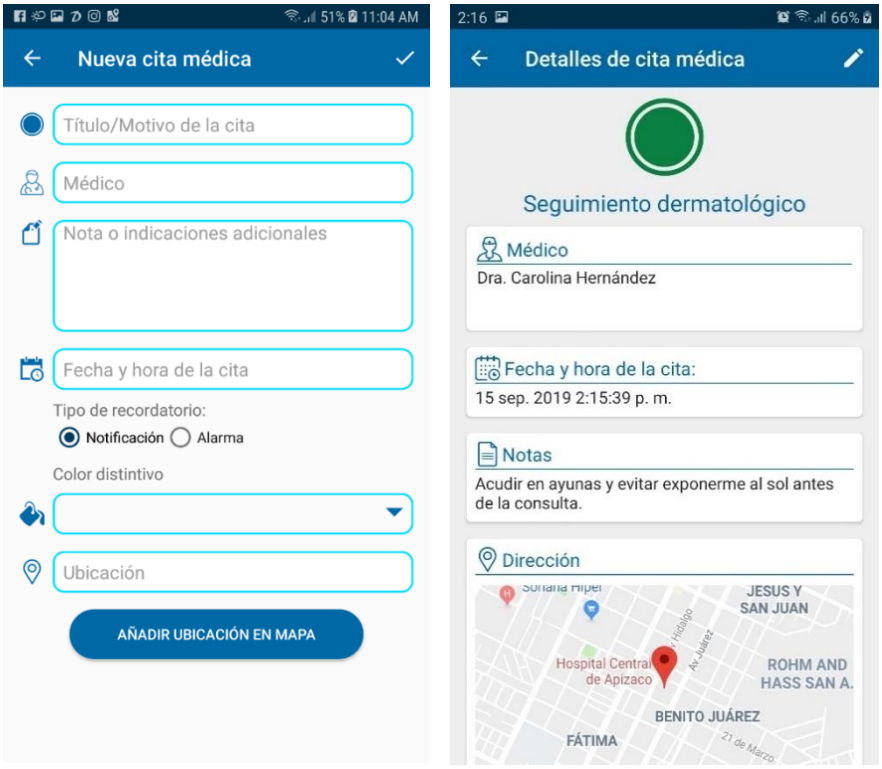

Figura 27 Formulario de registro y pantalla de detalles de una cita médica.

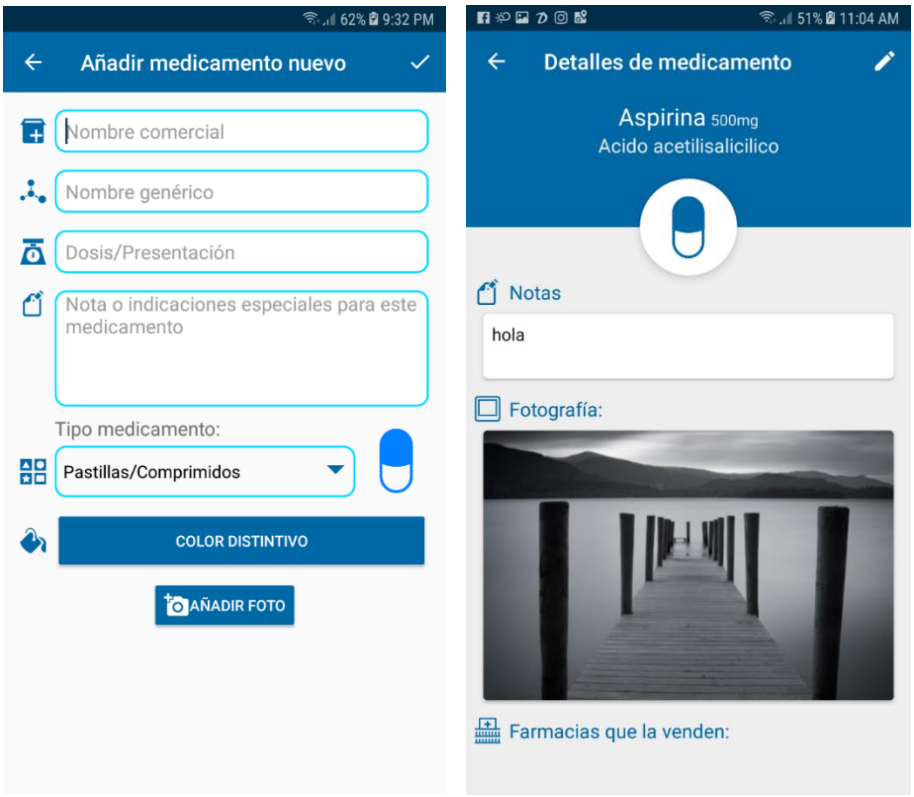

Figura 28 Formularios para el registro y pantalla de detalles de un medicamento.

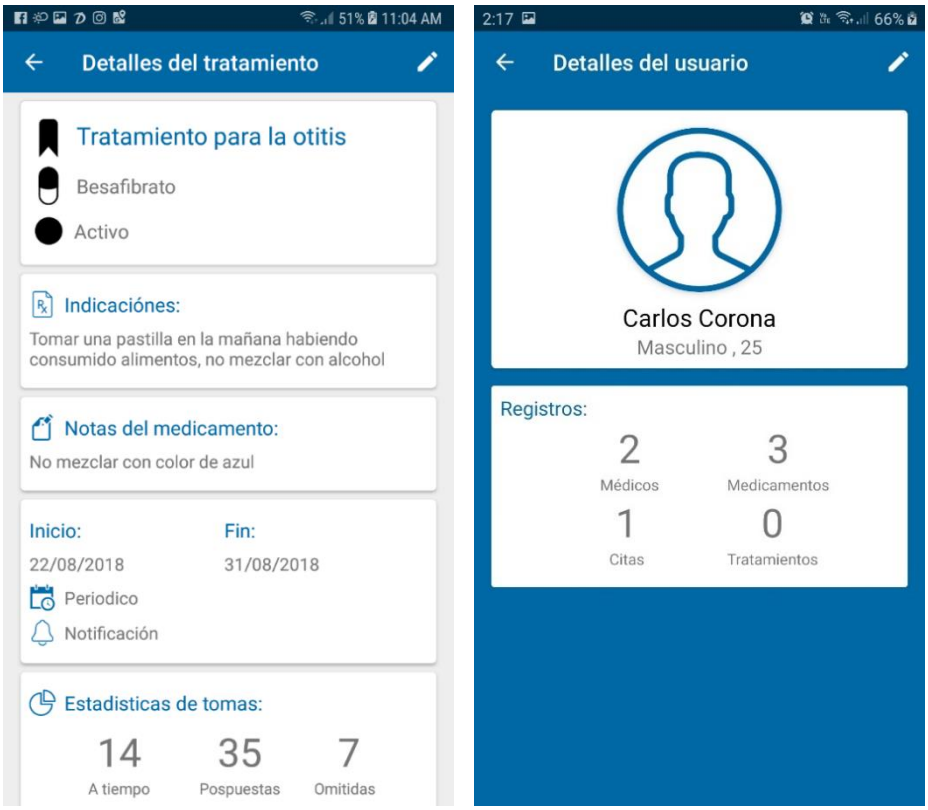

Figura 29 Pantalla de detalles de un tratamiento y de un usuario registrado.

3.6 Control de versiones y el proceso de integración continua

Todo proyecto de desarrollo de software alcanzará cierto grado de complejidad en algún punto de su ciclo de vida, esto hace que sea más que difícil mantener el código fuente pues hay una mayor probabilidad de que ocurra un error, ya sea un error humano a la hora de crear código nuevo o actualizar código existente, añadir una nueva funcionalidad o incluso corregir un error previamente detectado.

Es por eso es necesario incorporar mecanismos que nos permitan hacer estos cambios con el menor impacto posible en la estabilidad del producto actual, pues imaginemos que tenemos algún sistema con millones de líneas de código que deje funcionar por un pequeño error humano, detectar donde se ha cometido este error puede llevar mucho tiempo y dinero.

Para ello se han implementado los conceptos de "control de versiones" y de "integración continua". El primer término como su nombre indica consiste en tener una gestión de las capacidades o configuraciones que tiene un elemento en nuestro caso código fuente. Por ejemplo, dos programadores pueden tener la misma copia exacta del código fuente, siendo el programador A el encargado de añadir una característica mientras que el programador B es el encargado de cambiar el aspecto de la aplicación que ve el usuario final. Posteriormente será necesario fusionar ambas versiones suponiendo que todo se haya realizado sin problemas.

Sí bien el control de versiones se puede realizar de forma manual esto se convierte rápidamente en una tarea titánica, teniendo en cuenta que:

- Se pueden haber modificados decenas o cientos de archivos, ¿Cómo sabe que el archivo que yo modifiqué no lo ha modificado otro programador?
- ¿Cómo evitar tener varias copias del mismo proyecto cuando son varios los programadores trabajando en el código fuente?
- ¿Cómo saber que lo que ha realizado otro programador no tendrá conflictos con lo que yo he creado?
- ¿Cómo es posible trabajar en un mismo archivo de código fuente donde varios programadores tienen que modificar al mismo tiempo?

Para ello se han creado herramientas de software que nos permiten realizar estas tareas de una forma más simple y menos propensa a errores si es que se sigue cierta metodología.

Figura 30 Logos de Git y GitHub Figura descargada de: https://medium.com/cs-note/git-and-github-for-beginners-iii-tutorial-fb64f7430721.

Una de estas herramientas y quizás una de las más famosas es git [21], un motor de bajo nivel para el control de versiones, siendo manejado con una consola o terminal, permite llevar la administración de “branches” o ramas, es decir bifurcaciones independientes de un proyecto desde una versión inicial de código fuente, posteriormente es posible añadir características a cada una de estas ramas y finalmente fusionar los cambios hacía una versión “central”, documentando cada uno de estos cambios (conocidos comúnmente como commits) realizados para que otros programadores tengan un conocimiento rápido de lo que se implementó en dicha rama, pero quizás una de sus características más poderosas es que es posible regresar a una versión anterior fácilmente, así que siempre y cuando se realicen cambios y commits atómicos, en caso de que se presente un error es posible regresar a una versión funcional anterior.

Git es un software gratuito, el cual se instala en el equipo de cómputo donde queremos tener estás características de manejo de versiones, pero Git por sí solo no suele ser suficiente, pues es necesario poder compartir las nuevas ramas o versiones a otros programadores para que puedan comenzar a realizar sus propias implementaciones, además de que si todo el modelo de ramas gira en torno a un solo equipo de cómputo es probable que terminemos con decenas o cientos de variantes de un mismo proyecto de software. Para solucionar este problema, se suele delegar el almacenamiento de estas versiones a un servidor en internet o en una nube privada, este servidor debe incluir otro software especializado para poder administrar a los proyectos, usuarios y ramas.

GitHub es un servicio de tipo “forge”, pues permite alojar proyectos usando el sistema de control de versiones Git en internet, manteniendo ellos la infraestructura y el “uptime” del servicio. Ofrecen dos variantes. Una gratuita y una de pago, aunque ambas opciones ofrecen las siguientes características:

- Capacidad de publicar repositorios/proyectos públicos o privados ilimitados.
- La capacidad de añadir colaboradores ilimitados a estos repositorios.
- Publicar una página web de dicho proyecto.
- Una wiki del proyecto.
- Grafica de las aportaciones de los programadores y sus repositorios, así como de las ramas generadas.
- Seguimiento visual de los cambios en el código.

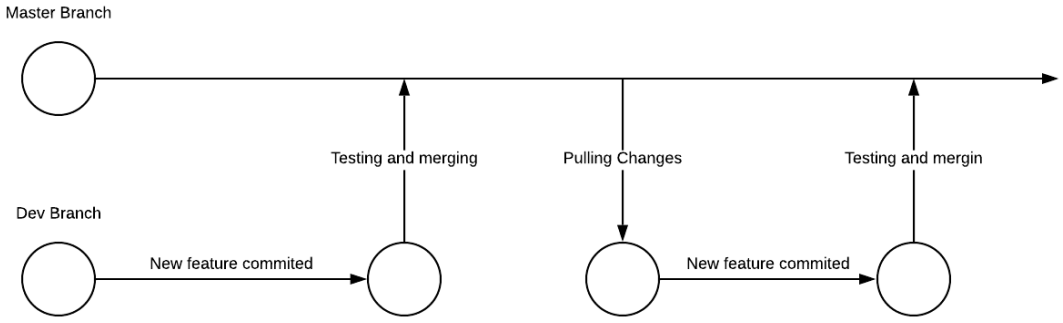

Figura 31 Modelo de ramas.

Se creará un repositorio privado en GitHub, el cual contendrá 2 ramas (representado en la Figura 31) con características y funcionalidades diferentes siendo:

- Master: Será la rama que contendrá todo el código funcional que haya pasado las pruebas de funcionamiento en otras ramas y procesos de desarrollo. De esta rama es que se compilarán las versiones de la aplicación que serán preparadas para ser publicadas en la tienda de aplicaciones.
- Dev: Contendrá código experimental que será creando cuando se vayan implementando las funcionalidades, por tanto, es posible que produzca resultados inestables o descartables. Cuando se tenga una versión de la aplicación que cumpla con todas las características base planteadas en este documento se hará un merge con Master excluyendo el código de pruebas.

Gracias a este modelo de manejo de ramas podemos añadir funcionalidades de forma continua con una menos probabilidad de dañar tanto el código funcional como el producto que actualmente compila satisfactoriamente.

3.7 Pruebas manuales y unitarias del sistema.

A medida que vayamos creando código fuente de la aplicación esta comenzará a realizar más funcionalidades, ya sean visibles para el usuario (cómo cambiar de una actividad a otra) o invisibles (cómo registrar un elemento en la base de datos, iniciar un servicio, etc.), por tanto, es necesario utilizar medios que permitan probar estos dos tipos de funcionalidades. Para ello utilizaremos dos tipos de pruebas: Manuales y Unitarias.

Las pruebas manuales son como su nombre indican pruebas que se realizan a mano por el programador, usualmente relacionadas con funcionalidades en las que la interfaz gráfica está involucrada como mostrar un valor en pantalla, navegación, inserción de valores correctamente (por ejemplo, solo aceptar caracteres numéricos donde solo se usan valores de este tipo). Este tipo de pruebas son bastante simples de realizar y permiten detectar errores que un usuario humano podría llegar a cometer durante un caso no considerado durante el flujo normal de la aplicación. Las pruebas manuales suelen apoyarse de puntos de interrupción para saber el valor de las variables y el estado de la aplicación ante ciertos eventos, así que es posible leer esos valores y saber que contienen el valor correcto o el que se espera.

El según tipo son las pruebas unitarias, las cuales son pruebas semiautomatizadas que permiten probar fácilmente le lógica de componentes individuales del sistema. Ejecutar estas pruebas unitarias cada vez que crea una compilación o ejecución permiten detectar y corregir rápidamente fallos, así como realizar optimizaciones a la lógica del sistema. Las pruebas unitarias son código que prueba código, utilizando el concepto dada una entrada se espera una salida en concreto. Una buena arquitectura permite saber las salidas esperadas incluso antes de codificar las funcionalidades verdaderas. Por tanto mientras se cubran las condiciones de entradas y salidas una prueba unitaria ya escrita puede ser utilizada durante el proceso de desarrollo de la aplicación.

El código fuente creado para las pruebas unitarias es código que no se incluye en una compilación final de reléase, por tanto no afecta al tamaño del archivo final de instalación ni al funcionamiento original planeado.

Android Studio incluye un framework de pruebas unitarias, las cuales corren cuando un dispositivo físico o un emulador son utilizados para desplegar la aplicación en modo de depuración.

Para incluir el framework para pruebas unitarias es necesario añadir las siguientes líneas al archivo build.gradle de la aplicación:

```
dependencies {
    // Required -- JUnit 4 framework
    testImplementation 'junit:junit:4.12'
    // Optional -- Robolectric environment
    testImplementation 'androidx.test:core:1.0.0'
    // Optional -- Mockito framework
    testImplementation 'org.mockito:mockito-core:1.10.19'
}
```

Una vez terminada la descarga de las dependencias, el proyecto se actualizará e incluirá una nueva carpeta llamada "Test" en la cual debemos incluir todas las clases que contendrán las pruebas unitarias. Así que por último crear una prueba unitaria consiste en crear una clase que contiene métodos públicos que no retornan valores y que usan la anotación @Test para decirle al framework que son pruebas. Por ejemplo la siguiente prueba verifica que el valor ingresado en la interfaz gráfica es la dirección de email "name@email.com".

```
import com.google.common.truth.Truth.assertThat;
import org.junit.Test;

public class EmailValidatorTest {
    @Test     public void emailValidator_CorrectEmailSimple_ReturnsTrue()
{
        assertThat(EmailValidator.isValidEmail("name@email.com")).isTrue(
);
    }
}
```

Las pruebas unitarias suelen utilizar booleanos para determinar si una prueba ha pasado o no pero depende del programador determinar en qué condiciones una prueba pasa o falla.

Utilizaremos estos dos tipos de pruebas para ir verificando la calidad y la funcionalidad de cada una de las características que se vayan desarrollando, por tanto es muy probable que se escriban pruebas unitarias según se vayan necesitando.

3.8 Configuración del entorno básico de desarrollo

Para comenzar a crear nuestra aplicación necesitamos crear y configurar la base de nuestro entorno de desarrollo. Primeramente, instalamos Android Studio, el IDE oficial para desarrollo de aplicaciones Android.

Esto lo conseguimos ingresando al sitio web: https://developer.android.com/studio/ y descargando el instalador. El proceso es bastante sencillo pues simplemente tenemos que especificar la carpeta

donde queremos que se instale el IDE. Una vez terminado ejecutamos Android Studio y podemos crear un nuevo proyecto.

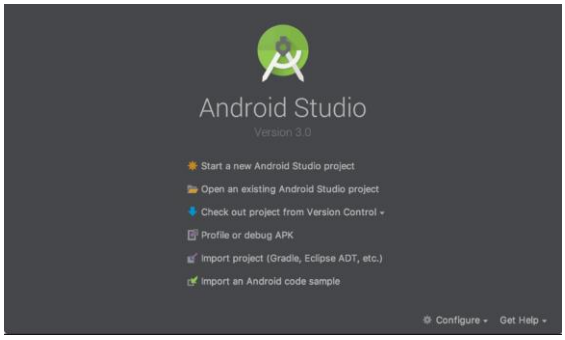

Figura 32 Pantalla de inicio de Android Studio.

Pero antes debemos crear nuestro repositorio remoto, para implementar desde un principio nuestro modelo de ramas. Para ellos debemos ingresar a https://github.com/ y registrarnos usando el correo institucional.

Hacemos clic en “New Repository”, rellenamos el formulario con los datos básicos de nuestro nuevo proyecto para finalmente hacer clic en “Create Repository”.

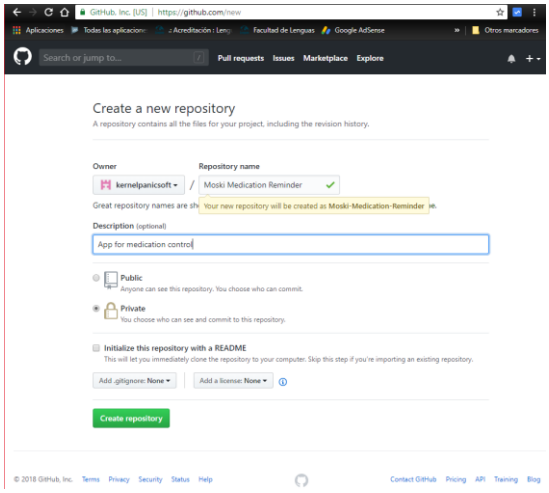

Figura 33 Formulario para la creación de un repositorio en GitHub.

Una vez creado el repositorio es posible configurarlo con las indicaciones que nos mostrará el sitio web.

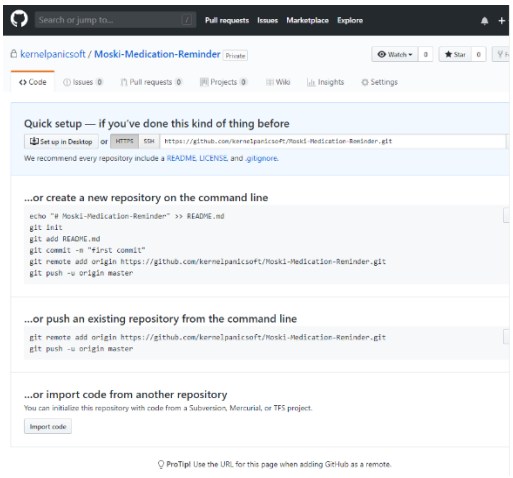

Figura 34 Repositorio remoto creado y listo para configurar.

Después viene la creación del repositorio local, para ello es necesario instalar git desde el sitio web: https://git-scm.com/, Una vez hecho esto debemos crear nuestro repositorio y ramas locales, para posteriormente subir estos cambios a nuestro repositorio remoto.

Debemos crear el directorio que contendrá nuestro proyecto, en este caso haremos uso de Android Studio para crear la configuración inicial de nuestro proyecto. Explicaremos más a detalle la instalación y uso de Android Studio más adelante, solo haremos este pequeño salto en el tiempo para demostrar cómo es que quedará nuestra configuración de modelo de ramas.

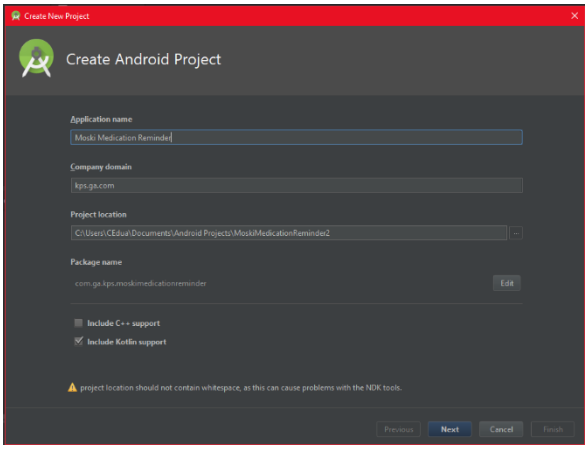

Figura 35 Pantalla de creación de un proyecto nuevo en Android Studio.

Especificamos el nombre de la aplicación (el nombre que el usuario visualiza cuando esta es instalada en un dispositivo Android), el nombre de dominio (usado para identificar la aplicación a nivel técnico dentro de los servicios de Google y usando otras API's), la ubicación del proyecto, que es el directorio donde se almacenará nuestro proyecto y también será nuestro repositorio local, finalmente hay que hacer clic en el cuadro de "Include Kotlin support" para poder hacer uso de la compatibilidad con este lenguaje de programación de forma nativa en la aplicación.

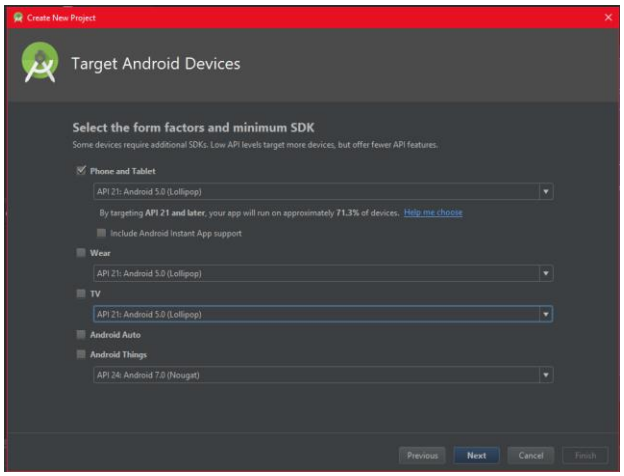

Figura 36 Pantalla de especificación de soporte para un proyecto Android.

En la pantalla siguiente hay que especificar la versión mínima soportada en nuestra aplicación, como nuestra aplicación solo está pensado para teléfonos y tabletas, dejamos intactas las opciones de Wear, TV, Android Auto y Android Things.

Finalmente debemos de especificar la plantilla de la pantalla inicial de la aplicación, para finalmente hacer clic en “Finish”, se crearán los archivos iniciales y configuración básica del proyecto.

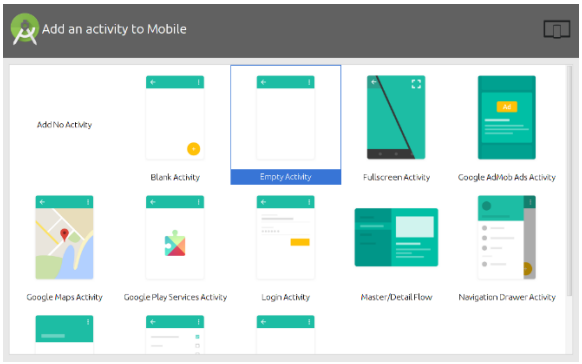

Figura 37 Pantalla de la primera actividad de un proyecto Android.

Una vez terminado el proceso de creación de nuestro proyecto base es necesario ir al directorio de este último e iniciar una terminal de git en esa ubicación para posteriormente usar el comando “git init”, el cual crea un repositorio local.

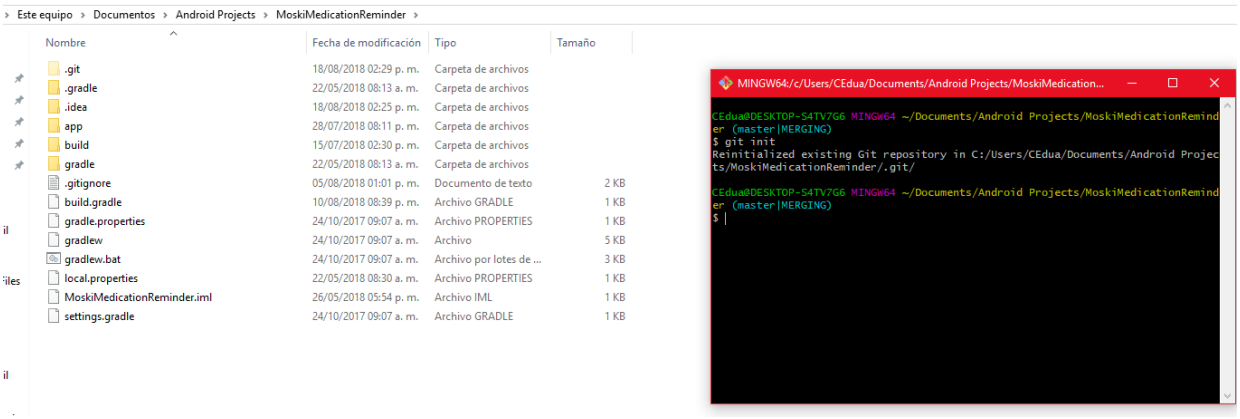

Figura 38 Pantalla de creación de un repositorio local usando Git.

Debemos añadir nuestro repositorio remoto a esta configuración, para ello ejecutamos los siguientes comandos en la terminal:

```
git init
git add -A
git commit -m "Initial commit with code"
git remote add origin https: //github.com/kernelpanicsoft/Moski-
Medication-Reminder.git
git push -u origin master
```

Una vez añadido estos cambios es posible recargar la página del proyecto en https://github.com/kernelpanicsoft/Moski-Medication-Reminder ahora podremos ver que todo el código fuente se encuentra en nuestro repositorio remoto.

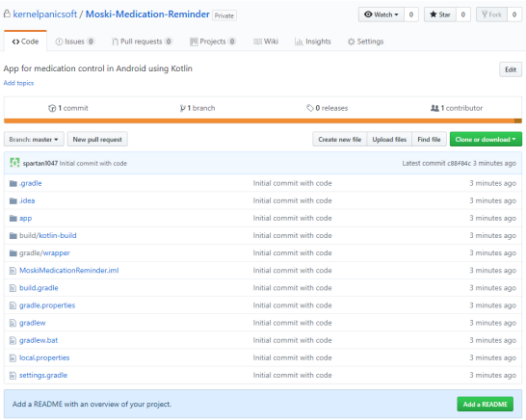

Figura 39 Repositorio remoto sincronizado con el repositorio local.

La primera vez que se sube contenido al repositorio remoto este es añadido a una rama creada automáticamente, está es la rama “master”, debemos añadir las otras dos ramas necesarias para el proyecto, para esto debemos crear primero las 2 ramas de forma local y luego publicarlas en nuestro repositorio remoto, para ello ejecutamos los siguientes comandos:

```
git checkout -b dev
git push origin dev
git checkout -b feature
git push origin dev
```

Finalmente, antes de comenzar con el desarrollo del proyecto tenemos que fusionar el contenido de la rama master a las ramas dev y Feature, pues en este punto del proyecto todo el código debe ser funcional y compilar sin problemas, así que a partir de ahora debemos añadir funcionalidades a Feature, fusionarla con dev, probar en dev y fusionarla con master cuando todo funcione correctamente.

Para listar las ramas locales usamos el comando: “git branch” mostrando la rama activa con un color distinto y un asterisco (*) del lado izquierdo.

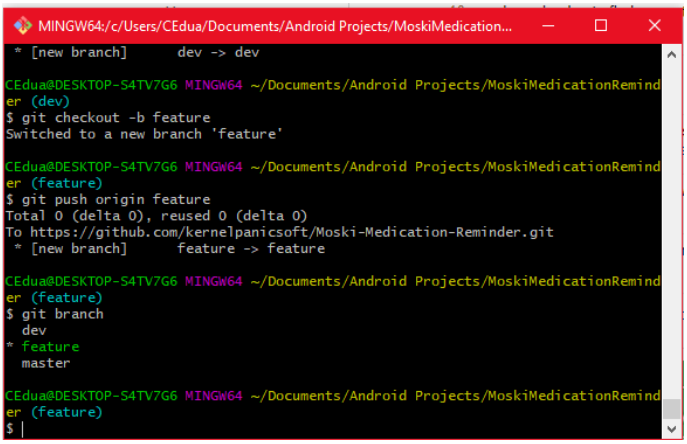

Figura 40 Creación y visualización de ramas locales en Git.

Fusionamos la rama master con la rama activa usando el comando: “git merge *RAMA a FUSIONAR CON RAMA ACTUAL*”. En nuestro caso al crear la rama copiamos el contenido de master en las ramas nuevas con el comando “git push origin RAMA A COPIAR”, pues volvemos a hacer mención que en este punto el código actual es idéntico en todas las ramas.

Podemos comprobar que se ha realizado la creación y código de las ramas visitando el repositorio remoto del proyecto y navegando entre ellas.

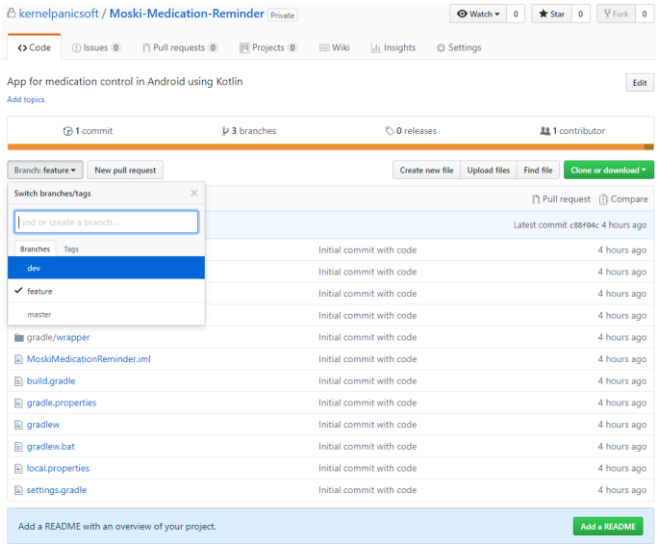

Figura 41 Visualización de ramas remotas en GitHub.

Es probable que ocurran "conflictos" a la hora de hacer merge a las ramas, ya sea por inconsistencias entre las ramas, archivos o código fuente. En estos casos seremos notificados por git y habremos de corregir estos problemas de forma manual antes de enviar los cambios.

Por último, explicaremos el comando "git commit", quizás el más importante y el más usado, pues es el comando que nos permite reportar los cambios que se han realizado en cada tanda de actualizaciones, justo antes de enviarla a nuestras ramas locales o remotas.

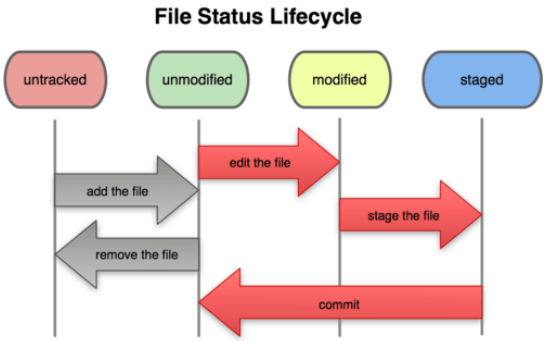

Figura 42 Ciclo de vida de un archivo en Git Figura descargada de: https://git-scm.com/book/es-ni/v1/Git-Basics-Recording-Changes-to-the-Repository.

En este punto debe quedar claro que, si se tiene un repositorio completo de Git, se tiene también una copia completa de los archivos de ese proyecto. Según la documentación oficial de Git el ciclo de vida de un archivo es el siguiente:

> *"Cada archivo en el directorio de trabajo puede estar en uno de dos estados: bajo seguimiento (tracked), o sin seguimiento (untracked). Los archivos bajo seguimiento son aquellos que existían en la última instantánea (commit); pueden estar sin modificaciones, modificados, o preparados. Los archivos sin seguimiento son todos los demás —cualquier archivo del*

directorio que no estuviese en tu última instantánea ni está en el área de preparación—. La primera vez que clonas un repositorio, todos tus archivos estarán bajo seguimiento y sin modificaciones, ya que los acabas de copiar y no has modificado nada.

A medida que se editan los archivos, Git los ve como modificados, porque los hemos cambiado desde tu última confirmación. Preparas estos archivos modificados y luego confirmas con un commit todos los cambios que hayas preparado, y el ciclo se repite. "

El comando "commit" es usado para guardar los cambios en el repositorio local (notificando los cambios "globales del proyecto", no de los archivos en sí). Para notificar los cambios antes de realizar el "commit" es necesario decirle a git explícitamente que cambios deseamos incluir. Esto quiere decir que un archivo no será añadido automáticamente en el siguiente commit solo porque ha sido modificado, en cambio, necesitamos hacer uso del comando "git add" para marcar cuales cambios deberán ser incluidos en la siguiente inclusión o lote de cambios.

Además, es importante notar que, en Git, un "commit" no se ve reflejado automáticamente en el repositorio remoto, sino que solo guarda los cambios en el repositorio local hasta que explícitamente enviamos los cambios al repositorio remoto. Esta es una característica muy poderosa pues nos da la oportunidad de verificar que todo está de forma local antes de enviar los cambios al repositorio remoto, pues en caso de haber un fallo podría retrasar el proceso de desarrollo del proyecto por invertir tiempo en correr los fallos, aunque esto se previene usando varias ramas como es que tenemos planeado.

Capítulo 4. Implementación del Sistema

En este capítulo documentamos el proceso de desarrollo del proyecto, el cual como especificamos con anterioridad está basado en sprints que a su vez contienen cada una de las tareas que se llevaron a cabo. Es este capítulo que se reportarán todos los cambios y problemas a los que nos enfrentamos cuando se estaba codificando la aplicación y las soluciones propuestas para afrontarlos.

4.1 Aplicación del patrón MVVM

En este punto durante la recopilación de requerimientos hemos comenzado a configurar la mayoría de las herramientas que serán necesarias para la codificación y ejecución del proyecto, pero seguramente aparecerán nuevas herramientas y procesos que han de utilizarse en puntos avanzados del desarrollo de la aplicación, los cuales explicaremos en su momento.

Podemos comenzar a especificar las tareas que son necesarias para avanzar en la creación de la aplicación, debemos tener en mente que estas tareas deberán ser realistas en términos de alcance, es decir tareas que podamos cumplir en un periodo de un mes especificando como intervalo de duración “días” y que deberán ser atómicas pero que permitan fusionarse con otras para ir construyendo el producto incrementalmente, no podemos por ejemplo crear el servicio de respaldo de datos si no sabemos qué datos contendrá la aplicación en sí.

Las tareas seguirán el siguiente formato: Título, descripción/requisitos, ejecución y resultados.

Ya tenemos un proyecto vacío, pero que se puede compilar y ejecutar, así que este será nuestro punto inicial. A partir de este punto explicaremos las tareas que fueron necesarias para la realización de la aplicación.

4.1.1 Creación del icono de la aplicación

Descripción/Requisitos:

Android requiere de un icono para toda aplicación el cual será usado para distintas partes del sistema: el lanzador de aplicaciones, el listado de multitarea, la ficha de la aplicación, etc…

El icono deberá cumplir las medidas para las distintas densidades de pantalla soportadas por Android, así como las líneas de diseño establecidas por Google y que pueden ser encontradas en el siguiente enlace:

https://developer.android.com/guide/practices/ui_guidelines/icon_design_launcher_archive

Ejecución:

Para la creación del icono se utilizó el software de diseño Adobe Photoshop. El icono de la aplicación estará disponible en distintas resoluciones por lo que es necesario tener en cuenta que el icono se escalará a distintos tamaños y que a su vez será usado en dispositivos con variaciones de densidad de pixeles.

La mejor aproximación es realizar un proyecto que soporte gráficos vectoriales, pues estos pueden ser escalados en el espacio de trabajo de Photoshop sin que pierdan calidad para posteriormente generar las imágenes en mapas de bits usando el formato .png para exportarlos a nuestro proyecto

en Android Studio. Adicionalmente es posible utilizar este mismo archivo para generar la Splash screen.

Debemos generar iconos en las siguientes resoluciones:

192x192px para xxxhdpi
144x144px para xxhdpi
96x96px para xhdpi
72x72px para hdpi
48x48px para mdpi

Los cuales deberán ser colocadas dentro de las carpetas de su respectiva densidad en: Android Projects\MoskiMedicationReminder\app\src\main\res

Finalmente, para que nuestra aplicación pueda hacer uso del icono debemos especificar su declaración en el archivo manifest.xml

```
<  application
        android:allowBackup  =  "true"
        android:icon  =  "@mipmap/ic_launcher"
        android:label  =  "@string/app_name"
        android:roundIcon  =  "@mipmap/ic_launcher_round"
        android:supportsRtl  =  "true"
```

Resultados:

Se creó un archivo .PSD el cual contiene la base de nuestro icono, usaremos este archivo para generar todos los iconos actuales y posteriores en los formatos necesarios.

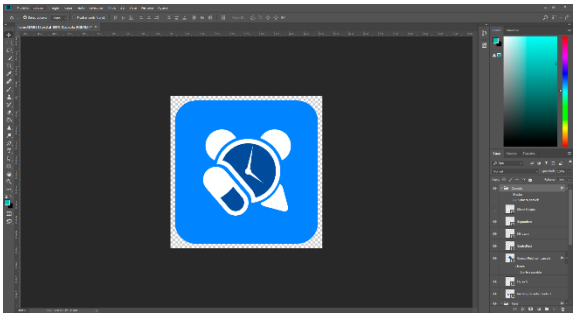

Figura 43 Proyecto en Adobe Photoshop conteniendo el icono de la aplicación.

La versión inicial del icono se generó en formato .png, con los tamaños y versiones necesarias, las cuales se colocaron en las carpetas indicadas por Android, posteriormente se modificó el manifiesto para que nuestra aplicación haga uso de este icono en todas las partes del sistema donde se requiera.

La aplicación al ser instalada en algún dispositivo Android mostrará nuestro icono según sea la resolución y tamaño de pantalla de nuestro dispositivo.

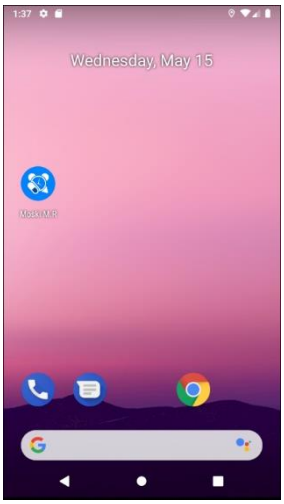

Figura 44 Aplicación instalada en Android con el icono creado.

4.1.2 Creación de la Splash Screen de la aplicación

Descripción/Requisitos:

Al abrir la aplicación (inicializando el proceso) se debe mostrar una pantalla estática mientras se cargan los datos de la aplicación, esto previene que el usuario vea pantallas vacías que se llenan abruptamente de datos y previenen errores del tipo Null Pointer Exception sí es que intenta interactuar con elementos aun inexistentes en la vista.

Ejecución:

Se creó un nuevo archivo XML dentro de res/drawable especificando la ubicación e imagen mostrará, quedando como:

```
<?xml  versión = "1.0" encoding = "utf-8"?>
   <layer-list xmlns:
        android =  http://schemas.android.com/apk/res/android">
        <item android: drawable = "@color/gray"/>
                <bitmap
                        android:gravity  =  "center"
                        android:src = "@mipmap/ic_launcher"
                />
        </item>
   </layer-list>
```

Debemos crear un elemento que sea el estilo “padre” de la actividad que contendrá la splash screen, esta no deberá contener la barra de titulo o appbar.

```
<resources>
   <!-- Base application theme. -->
       <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar
">
       <style name="SplashTheme" parent="Theme.AppCompat.NoActionBar">
                <item name="android:windowBackground">
                @drawable/background_splash
```

```
                </item>
        </style>
</resources>
```

Toda actividad, provider, resolver y en sí cualquier elemento mayor de Android debe estar registrado en el archivo AndroidManifest.xml, en nuestro caso debemos especificar que esta actividad será el punto de entrada de la aplicación:

```
<activity android: name = ".SplashActivity"
 android: theme = "@style/SplashTheme" >
     <intent-filter >
         <action android:name = "android.intent.action.MAIN"/>
          <  category android:
name = "android.intent.category.LAUNCHER"/>
      </intent-filter> </activity >
```

Finalmente se crear un nuevo archivo que inicialice nuestra la actividad, espere la carga de datos y la termine quedando el código como:

```
class SplashActivity: AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle ? ) {
        super.onCreate(savedInstanceState)
         startActivity(Intent(this@ SplashActivity, MainActivity::class.j
ava))
         finish()
     }
}
```

Resultados:

La aplicación al abrirse muestra inicialmente la imagen que especificamos hasta que todos los datos están correctamente cargados. La Splash Screen se muestra cada vez que la aplicación inicial por primera vez que es cargada en memoria y dura solamente el tiempo que le tome al procesador cargar lo datos, por lo que en equipos de gama alta la pantalla de carga es muy corta y en equipos de prestaciones más modestas la pantalla solo dura de 1 a 2 segundos.

Figura 45 Splash Screen durante la apertura de la aplicación

4.1.3 Creación de las jerarquías de navegación

Descripción/Requisitos:

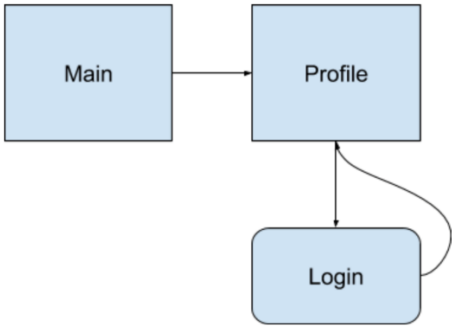

Figura 46 Lógica detrás del proceso de navegación en Android usando la navigation-stack, Figura descargada de: https://stackoverflow.com/questions/51582674/navigation-architecture-component-login-screen/54015319.

Se deberá especificar las rutas de navegación que el usuario seguirá a hacer toques en ciertas partes dentro de la aplicación de la forma en la que es especificada en los diagramas de secuencia, adicionalmente se deberá de brindar navegación "superior" en la nav bar de cada actividad distinta a la principal evitando que se recargue la actividad previa al regresar con los botones "back" o "up". Debemos definir la secuencia de navegación e implementarla, pues a futuro esta será usada para pasar datos de una actividad a otra. Por ejemplo, al tocar un elemento de la lista posteriormente se cargará la actividad mostrando los detalles de dicho elemento seleccionado, pudiendo el usuario volver a atrás y ver de nuevo el mismo listado y seleccionar otro elemento distinto.

Ejecución:

Cuando creamos las maquetas de la interfaz gráfica indirectamente estábamos creando las actividades de la aplicación, el problema es que no hay manera directa de navegar de una a otra dentro de la aplicación compilada, pues de momento su contenido únicamente es visible desde Android Studio abriendo directamente los archivos que forman parte de esa vista.

Nos apoyaremos de los diagramas de secuencia, así como del modelo de la base de datos para saber cómo se vinculan las actividades entre sí.

Primeramente, debemos especificar el código que permite navegar de una actividad a otra, el cual será invocado desde eventos del usuario, como toques a botones o elementos en las listas. El código es el siguiente:

```
val nav
= Intent(this@AnadirEstablecimientoActivity, MapsActivity::class.java)
startActivityForResult(nav, 6832)
```

Donde se crea una variable del tipo Intent que recibe 2 parámetros, el primero es el contexto de la actividad de origen y el segundo es el nombre de la clase que maneja la actividad de destino.

Debemos crear cada una de las actividades usando el diagrama de la base de datos y hacer que estas sean “hijas” de la actividad “principal”, eso lo conseguimos modificando el manifiesto de la siguiente manera:

```
    <application
         android:allowBackup="true"
         android:icon="@mipmap/ic_launcher"
         android:label="@string/app_name"
         android:roundIcon="@mipmap/ic_launcher_round"
         android:supportsRtl="true"
         android:theme="@style/AppTheme">
         <activity
             android:name=".SplashActivity"
             android:theme="@style/SplashTheme">
             <intent-filter>
                 <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER"
                    />
             </intent-filter>
          </activity>
        <activity android:name=".MainActivity" />
         <activity
              android:name=".AnadirMedicamentoActivity"
              android:parentActivityName=".MainActivity" />
        <activity
             android:name=".AnadirMedicoActivity"
              android:parentActivityName=".MainActivity" />
         <activity
             android:name=".AnadirEstablecimientoActivity"
              android:parentActivityName=".MainActivity" />
         <activity
             android:name=".AnadirTratamientoActivity"
              android:parentActivityName=".MainActivity" />
         <activity
             android:name=".AnadirCitaMedicaActivity"
              android:parentActivityName=".MainActivity" />
         <activity
             android:name=".DetallesMedicoActivity"
              android:parentActivityName=".MainActivity" />
         <activity
              android:name=".DetallesFarmaciaActivity"
              android:parentActivityName=".MainActivity" />
         <activity
              android:name=".DetallesMedicamentoActivity"
              android:parentActivityName=".MainActivity" />
         <activity
              android:name=".DetallesCitaMedicaActivity"
              android:parentActivityName=".MainActivity" />
         <activity
              android:name=".MapsActivity"
              android:label="@string/title_activity_maps"
              android:parentActivityName=".MainActivity" />
         <activity
              android:name=".DetallesTratamientoActivity"
              android:parentActivityName=".MainActivity" />
         <activity
```

```
            android:name=".RegistrarUsuarioActivity"
            android:parentActivityName=".MainActivity" />
        <activity
            android:name=".ListarUsuariosActivity"
            android:parentActivityName=".MainActivity" />
        <activity
            android:name=".DetallesPerfilActivity"
            android:parentActivityName=".MainActivity"/>
    </application>
```

Ahora debemos habilitar el botón up en cada una de las actividades especificadas en el código anterior y registrar el evento que genera cuando un usuario lo presiona, la implementación de este comportamiento es muy útil en pantallas de gran tamaño como las que disponen las tablets, pues hace más cómoda la navegación.

Para esto usaremos el siguiente código en cada una de las actividades:

```
    val toolbar = findViewById<Toolbar>(R.id.toolbar)
       setSupportActionBar(toolbar)
       val ab = supportActionBar
       ab!!.setDisplayHomeAsUpEnabled(true)
```

Finalmente registramos el evento que escucha la interacción del usuario y regresa a la pantalla anterior de la pila de navegación.

```
      override fun onOptionsItemSelected(item: MenuItem): Boolean {
       when (item.itemId) {
           android.R.id.home -> {
               onBackPressed()
               return true             }
       }
       return super.onOptionsItemSelected(item)
   }
```

Resultados:

Ahora es posible navegar entre las distintas actividades que forman parte de la aplicación, creando en tiempo de ejecución la pila de navegación, pudiendo el usuario regresar a una actividad anterior usando el botón back del sistema Android o el botón up de las actividades. La persistencia de los datos de cada actividad es mantenida durante cada push, pero destruida en cada pop, liberando memoria y permitiendo reiniciar procesos cuando el usuario lo desee (por ejemplo, datos erróneos en un formulario casi lleno).

4.1.4 Creación de adaptadores para las listas de datos

Descripción/Requisitos:

Se deberá de crear clases o métodos que permitan poblar las listas con alguna fuente de datos, ya sean arreglos o desde una base de datos.

Ejecución:

En el punto anterior a esta tarea las listas (RecyclerViews y ListViews) eran pobladas usando la directiva: android:entries="@array/dummy_data" la cual permite mostrar de forma rápida datos usando arreglos especificados de forma manual. Es necesario crear clases que permitan poblar las listas en tiempo de ejecución desde una base de datos. La clase deberá ser fácilmente adaptable a cualquier tabla especificada y brindando como salida el respectivo layout que representa un elemento en la vista de la lista.

Debemos crear una clase que extienda de ListAdapter que aplique el modelo ViewHolder y de la clase View.OnClickListener para responder a los eventos del usuario. El modelo ViewHolder permite al sistema mantener los valores de las variables cuando no se están mostrando en la pantalla del dispositivo, liberando la memoria que ocupan las vistas (TextViews, ImageView, ListItems, etc.) usando de nuevo aquellas que son vecinas inmediatas de las que son visibles para el usuario (Reciclándolas, de ahí el nombre de RecyclerView).

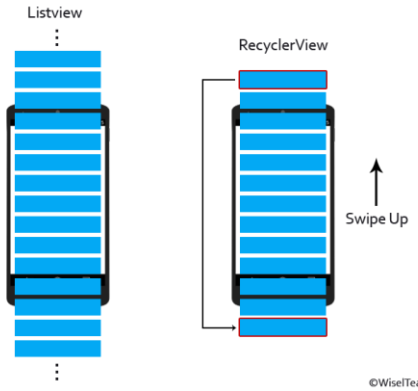

Figura 47 Explicación del modelo ViewHolder (Derecha) vs proceso tradicional, Figura descargada de: https://medium.com/@kish.imss/listview-vs-recyclerview-2965d50b363

Debemos especificar los métodos que nos permiten interactuar con la fuente de datos desde el adaptador, por ejemplo, saber que elemento es el que responde a un evento de usuario y extraer su información para ver sus detalles en otra actividad, saber cuántos elementos hay en la fuente de datos, cambiar de fuente de datos y vincular los datos con la vista.

Comenzamos incluyendo las librerías para soporte de LiveData. Los objetos con estas características tienen la ventaja de estar asociados al ciclo de vida de una actividad o fragmento, para ello incluimos dentro del archivo build.gradle las siguientes líneas:

```
    def lifecycle_version = "1.1.1"
    // ViewModel and LiveData
    implementation "android.arch.lifecycle:extensions:$lifecycle_version"
```

Con esto podremos comenzar a hacer uso de los tipos especiales creados para manejar el patrón MVVM sin necesidad de implementar aún sus clases por ejemplo LiveData y ViewModel, pudiendo compilar sin tener errores por referencias no encontradas.

La clase queda de la siguiente manera:

```
package com.kps.spart.moskimedicationreminder


import android.graphics.Bitmap
import android.graphics.BitmapFactory
import android.support.v7.util.DiffUtil
import android.support.v7.widget.RecyclerView
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.ImageView
import android.support.v7.recyclerview.extensions.ListAdapter
import android.widget.TextView
import elements.Usuario
import java.io.FileNotFoundException


class UsuariosAdapter:
ListAdapter<Usuario,UsuariosAdapter.ViewHolder>(DIFF_CALLBACK()),
View.OnClickListener {
    private var listener: View.OnClickListener? = null

    class DIFF_CALLBACK : DiffUtil.ItemCallback<Usuario>() {
        override fun areItemsTheSame(oldItem: Usuario, newItem: Usuario):
Boolean {
            return oldItem.uid == newItem.uid
        }

        override fun areContentsTheSame(oldItem: Usuario, newItem:
Usuario): Boolean {
            return oldItem.nombre.equals(newItem.nombre)
        }
    }

    inner class ViewHolder(v: View) : RecyclerView.ViewHolder(v){
        val icono : ImageView = v.findViewById(R.id.UsuarioIV)
        val nombre : TextView = v.findViewById(R.id.NombreUsuarioTV) as
TextView
        val apellidos : TextView =
v.findViewById(R.id.ApellidosUsuarioTV) as TextView
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
        val v = LayoutInflater.from(parent.context)
                .inflate(R.layout.list_item_usuario,parent,false)
        v.setOnClickListener(this)
        return ViewHolder(v)
    }
```

```
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val usuarioActual = getItem(position)


        if(usuarioActual.imagen.isNullOrEmpty()){
            holder.icono.setImageResource(R.drawable.ic_user)
        }else{
            val  valueInPixels =
holder.icono.context.resources.getDimension(R.dimen.listItemImagen)

holder.icono.setImageBitmap(setPic(usuarioActual.imagen!!,valueInPixels.t
oInt(),valueInPixels.toInt()))
        }

        holder.nombre.text = usuarioActual.nombre
        holder.apellidos.text = usuarioActual.apellidos
    }

    fun setOnClickListener(listener : View.OnClickListener){
        this.listener = listener
    }

    override fun onClick(v: View?) {
        listener?.onClick(v)
    }

    fun getUsuarioAt(position : Int) : Usuario{
        return getItem(position)
    }

    private fun setPic(mCurrentPhotoPath : String, targetW : Int, targetH
: Int) : Bitmap? {

        val bmOptions = BitmapFactory.Options().apply {
            inJustDecodeBounds = true
            BitmapFactory.decodeFile(mCurrentPhotoPath,this)
            val photoW: Int = outWidth
            val photoH: Int = outHeight

            val scaleFactor : Int = Math.min(photoW / targetW , photoH /
targetH )

            inJustDecodeBounds = false
            inSampleSize = scaleFactor
            inPurgeable = true
        }

        var scaledBitmap : Bitmap? = null
        try {

        BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions)?.also {
bitmap ->
            scaledBitmap = bitmap
        }

        }catch ( e : FileNotFoundException){
            return null
```

```
        }
        return scaledBitmap
    }
}
```

Ahora debemos hacer uso de este nuevo adaptador desde la vista que almacena nuestra lista, haciendo también una consulta a la base de datos para proporcionar un cursor que será pasado a nuestra instancia de clase:

```
        RecViewUsuarios.setHasFixedSize(true)
        val mLayoutManager =
LinearLayoutManager(this@ListarUsuariosActivity,LinearLayoutManager.VERTI
CAL,false)
        RecViewUsuarios.layoutManager = mLayoutManager

        val dividerIterator =
DividerItemDecoration(RecViewUsuarios.context, LinearLayout.VERTICAL)
        RecViewUsuarios.addItemDecoration(dividerIterator)

        //Asignamos el adaptador a nuestro Recyclerview
        val adapter = UsuariosAdapter()

        //Especificamos el escucha de eventos para definir el usuario
activo de la aplicacion
        adapter.setOnClickListener( View.OnClickListener {
            val usuarioSeleccionado =
adapter.getUsuarioAt(RecViewUsuarios.getChildAdapterPosition(it))

            if(!usuarioSeleccionado.password.isNullOrEmpty() &&
usuarioSeleccionado.uid != getCurrentUserID()){
                val builder = AlertDialog.Builder(this)
                builder.setTitle(getString(R.string.ingresar_contrasena))
                val inflater = layoutInflater
                val dialogView =
inflater.inflate(R.layout.dialog_input_password, null)
                builder.setView(dialogView)

                val passwordEditText : EditText =
dialogView.findViewById(R.id.userPasswordET)
                builder.setPositiveButton(getString(R.string.ingresar)) {
dialog, id  ->

if(usuarioSeleccionado.password.equals(passwordEditText.text.toString()))
{
                        val sharedPref =
PreferenceManager.getDefaultSharedPreferences(this@ListarUsuariosActivity
)
                        with(sharedPref.edit()){

putInt("actualUserID",usuarioSeleccionado.uid)
                            apply()
                        }
                        finish()
                    }else{
```

```
Snackbar.make(frameLayoutListaUsuarios,getString(R.string.contrasena_inco
rrecta), Snackbar.LENGTH_SHORT).show()
                    }
                }
                builder.setNegativeButton(getString(R.string.cancelar)) {
dialog, id ->
                }

builder.setNeutralButton(getString(R.string.recuperar_contrasena)) {
dialog, id ->

                    if(ContextCompat.checkSelfPermission(this,
Manifest.permission.WRITE_EXTERNAL_STORAGE) !=
PackageManager.PERMISSION_GRANTED){
                        ActivityCompat.requestPermissions(this,

arrayOf(Manifest.permission.WRITE_EXTERNAL_STORAGE),

CodigosDeSolicitud.SOLICITAR_PERMISO_ALMACENAMIENTO_EXTERNO)
                        userPassWordHelper = usuarioSeleccionado.password
                    }else{
                        recoverPassWord(usuarioSeleccionado.password)
                        Log.e("Contrasena", usuarioSeleccionado.password)
                    }

                }

                val alertDialog = builder.create()
                alertDialog.show()
            }else{
                val sharedPref =
PreferenceManager.getDefaultSharedPreferences(this@ListarUsuariosActivity
)
                with(sharedPref.edit()){
                    putInt("actualUserID",usuarioSeleccionado.uid)
                    apply()
                }
                finish()
            }
        }
        )

        RecViewUsuarios.adapter = adapter

        usuarioViewModel =
ViewModelProviders.of(this).get(UsuarioViewModel::class.java)
        usuarioViewModel.allUsuarios.observe(this,
Observer<List<Usuario>>{
            adapter.submitList(it)
        } )
```

Resultados:

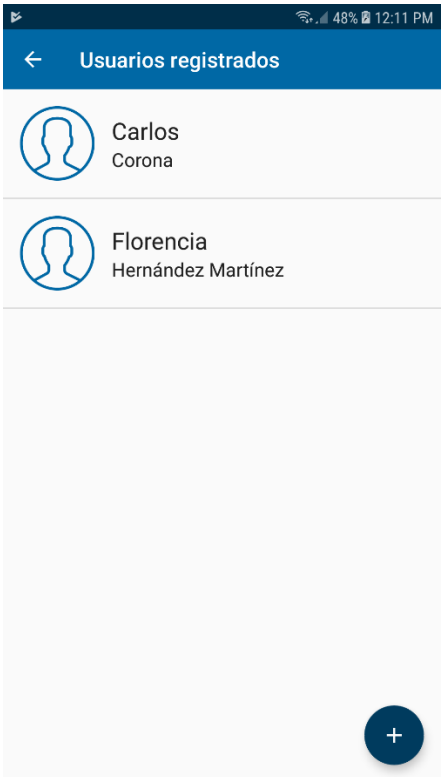

Figura 48 Lista de usuarios poblada usando un adaptador y un cursor.

Con nuestra base ahora es posible poblar nuestras listas usando una base de datos, como se puede ver en el código creamos una versión temporal de nuestra base de datos para probar el funcionamiento del adaptador. Un cursor es un objeto que almacena información de una consulta de base de datos permitiendo iterar los resultados de una forma más simple usando objetos.

Esta base puede ser rápidamente adaptada según sea la tabla que deseamos consultar por lo que será usada en todas las listas de nuestra aplicación. Las directrices de Google dictan que debe crearse una clase por cada lista, así que esta base será la plantilla para los demás Adaptadores.

4.1.5 Creación de la base de datos utilizando ROOM

Descripción/Requisitos:

Para comenzar a aplicar el patrón MVVM es necesario que creemos un modelo para que sea nuestra fuente de datos, en la implementación de los adaptadores se hace uso de una lista de objectos creados a mano del tipo usuario para poblar la interfaz gráfica, se hace uso de un ViewModel temporal pero aún es necesario crear las capas de los ViewModels y los repositorios para cada una de las entidades, pero por tanto debemos crear primero que nada las entidades.

Google propone el uso de ROOM, una librería de persistencia que genera una capa de abstracción sobre SQLite, pudiendo ser usada para crear bases de datos robustas, con todo el potencial del SQLite crudo pero con la facilidad de manejo de la programación orientada a objectos, pues ROOM es un sistema ORM (Object-Relational Mapping o Mapeo objeto-relacional), permitiendo convertir los datos de una tupla en una table dentro de la base de datos en un objeto, realizar operaciones con él y luego salvar el nuevo estado del objeto en la base de datos.

Por tanto, definir la base de datos con ROOM también define las clases de los objetos que serán usados, agilizando el proceso de desarrollo. Necesitamos definir la base de datos e instanciarla para comenzar a operar sobre ella, para esto utilizaremos el diagrama entidad-relación creado durante el proceso de arquitectura.

Ejecución:
Debemos incluir el soporte de ROOM en nuestra aplicación, para ello dentro del archivo build.gradle en su apartado de app incluimos la librería añadiendo las siguientes dependencias:

```
    def room_version = "1.1.1"
    implementation "android.arch.persistence.room:runtime:$room_version"
    kapt "android.arch.persistence.room:compiler:$room_version" // use
kapt for Kotlin
```

ROOM tiene 3 componentes principales:

- La base de datos: Es una clase especial que sirve como soporte y principal punto de acceso a la base de datos en SQLite, esta clase debe cumplir ciertas condiciones:
    1. Debe ser una clase abstracta que extiende de RoomDatabse y debe tener la anotación @Databse.
    2. Debe incluir una lista de entidades asociadas.
    3. Debe contener un método abstracto que tiene 0 argumentos y retorna la clase que tiene la anotación @Dao.
- Entidad: Representa una tabla dentro de la base de datos.
- DAO: Contiene los métodos necesrios para acceder a la base de datos.

La aplicación utiliza la base de datos en ROOM para generar DAOs (Data Access Objects, en español Objetos de Acceso a Datos), los cuales están asociados a la base de datos en SQLite, la aplicación entonces usa cada DAO para conseguir entidades que son manipulados y salvados de nuevo en la base de datos en SQLite.

La definición de las entidades queda de la siguiente manera:

Usuario.kt:

```
@Entity
data class Usuario(@PrimaryKey(autoGenerate = true) var uid: Int,
                   var nombre: String? = null,
                   var apellidos: String? = null,
                   var edad: Int? = null,
                   var genero: String? = null,
```

```
                var imagen: String? = null,
                var password: String? = null,
                var email_recuperacion: String? = null
                )
```

CitaMedica.kt:

```
@Entity(foreignKeys = arrayOf(ForeignKey(entity = Usuario::class,
        parentColumns = arrayOf("uid"),
        childColumns = arrayOf("usuarioID"),
        onDelete = ForeignKey.CASCADE)))
class CitaMedica(@PrimaryKey(autoGenerate = true) var id: Int,
                 var titulo: String? = null,
                 var doctor: String? = null,
                 var especialidad: String? = null,
                 var nota: String? = null,
                 var fecha: String? = null,
                 var hora: String? = null,
                 var ubicacion: String? = null,
                 var tipoRecordatorio: Int? = null,
                 var color: Int? = null,
                 var latitud : Double? = null,
                 var longitud : Double? = null,
                 var statusCita : Int? = null,
                 var usuarioID : Int? = null)
```

Establecimiento.kt:

```
@Entity(foreignKeys = arrayOf(ForeignKey(entity = Usuario::class,
        parentColumns = arrayOf("uid"),
        childColumns = arrayOf("usuarioID"),
        onDelete = ForeignKey.CASCADE)))
class Establecimiento(@PrimaryKey(autoGenerate = true) var id : Int,
                      var nombre: String? = null,
                      var tipo: String? = null,
                      var direccion: String? = null,
                      var telefono1: String? = null,
                      var telefono2: String? = null,
                      var email: String? = null,
                      var sitioWeb : String? = null,
                      var latitud : Double? = null,
                      var longitud : Double? = null,
                      var usuarioID : Int? = null
)
```

FichaContacto.kt:

```
@Entity(foreignKeys = arrayOf(ForeignKey(entity = Medico::class,
        parentColumns = arrayOf("id"),
        childColumns = arrayOf("medico_ficha_id"),
        onDelete = ForeignKey.CASCADE)))
class FichaContacto (@PrimaryKey(autoGenerate = true) var id: Int,
                     var titulo : String? = null,
                     var direccion : String? = null,
                     var telefono : String? = null,
```

```
                        var celular : String? = null,
                        var email : String? = null,
                        var sitioweb : String? = null,
                        var accesoRapido : Boolean? = false,
                        var medico_ficha_id : Int? = null)
```

Medicamento.kt

```
@Entity(foreignKeys = arrayOf(ForeignKey(entity = Usuario::class,
                        parentColumns = arrayOf("uid"),
                        childColumns = arrayOf("usuarioID"),
                        onDelete = ForeignKey.CASCADE)))
data class Medicamento (@PrimaryKey(autoGenerate = true) var id: Int,
                    var nombreMedicamento: String? = null,
                    var nombreGenerico: String? = null,
                    var dosis: String? = null,
                    var nota: String? = null,
                    var tipo: String? = null,
                    var color: Int? = null,
                    var fotografia: String? = null,
                    var usuarioID : Int? = null)
```

Medico.kt

```
@Entity(foreignKeys = arrayOf(ForeignKey(entity = Usuario::class,
        parentColumns = arrayOf("uid"),
        childColumns = arrayOf("usuarioID"),
        onDelete = ForeignKey.CASCADE)))
class Medico (@PrimaryKey(autoGenerate = true) var id: Int,
              var nombre: String? = null,
              var especialidad: String? = null,
              var titulo: String? = null,
              var fichaContactoAR: Int = -1,
              var usuarioID : Int? = null
              )
```

Toma.kt

```
@Entity(foreignKeys = arrayOf(
        ForeignKey(entity = Tratamiento::class,
                parentColumns = arrayOf("id"),
                childColumns = arrayOf("tratamientoID"),
                onDelete = ForeignKey.CASCADE
                )
))
class Toma(@PrimaryKey(autoGenerate = true) var id: Int,
    var statusToma : Int = 0,
    var horaToma: String? = null,
    var tratamientoID: Int? = null

    )
```

Tratamiento.kt

```
@Entity(foreignKeys =  arrayOf(
        ForeignKey(entity = Medicamento::class,
                   parentColumns = arrayOf("id"),
                   childColumns = arrayOf("medicamentoID"),
                   onDelete = ForeignKey.CASCADE),

        ForeignKey(entity = Usuario::class,
                parentColumns = arrayOf("uid"),
                childColumns = arrayOf("usuarioID"),
                onDelete = ForeignKey.CASCADE
                )
))
class Tratamiento (@PrimaryKey(autoGenerate = true) var id: Int,
                   var titulo: String? = null,
                   var usuarioID: Int? = null,
                   var medicamentoID: Int? = null,
                   var indicaciones: String? = null,
                   var fechaInicio: String? = null,
                   var fechaFin: String? = null,
                   var diasTratamiento: Int = 0,
                   var status: Int? = null,
                   var recordatorio: Int? = null,
                   var atiempo: Int = 0,
                   var pospuestas: Int = 0,
                   var omitidas: Int = 0)
```

Con las entidades creadas podemos proceder a implementar los DAOs para cada una, un DAO es una interface con la anotación @Dao, que puede contener métodos con las anotaciones @Insert, @Update, @Delete y @Query, cómo la propia anotación indica, representan los comandos INSERT, UPDATE Y DELETE de SQLite, mientras que la anotación @Query permite realizar consultas directamente en la tabla de esa entidad.

Los métodos con las etiquetas @Insert, @Update, @Delete realizarán sus respectivas acciones, respetando las reglas definidas en la declaración de las entidades, en nuestro caso siendo la variable id la única que no puede ser null, además de que al usar la etiqueta @PrimaryKey(autoGenerate = true) estamos especificando que esa variable será la llave primaria y el atributo autoGenerate = true indica que será autoincrementable. Haciendo uso de la etiqueta ForeignKey podemos especificar llaves foráneas, especificando la entidad/clase y la variable que se apuntarán, el atributo onDelete = ForeignKey.CASCADE especifica que al eliminar al usuario al cual pertenece un elemento todos los elementos relacionados a él, también son eliminados, de esta manera procuramos optimizar el uso del espacio en disco.

El DAO para la entidad usuario queda de la siguiente manera:

```
@Dao
interface UsuarioDao {
    @Insert
    fun insert( usuario : Usuario)

    @Update
```

```
    fun update( usuario : Usuario)

    @Delete
    fun delete( usuario : Usuario)

    @Query("DELETE FROM Usuario")
    fun deleteAllUsuarios()

    @Query("SELECT * FROM Usuario")
    fun getAllUsuarios() : LiveData<List<Usuario>>

    @Query("SELECT * FROM Usuario WHERE Usuario.uid = :id")
    fun getUsuario( id : Int?) : LiveData<Usuario>

    @Query("SELECT max(uid) FROM Usuario")
    fun getLastUserID() : LiveData<Long>
}
```

Y aquí podemos una de las principales ventajas de ROOM sobre el uso de SQLite puro, y es el encapsulamiento de las consultas, pues solo lo que está definido en el DAO puede ser consultado desde una fuente externa, cualquier otro intento de acceso causara una excepción, por tanto, se reduce enormemente la posibilidad de acceso no autorizado por inyección de SQL.

Se creó una DAO para cada una de las entidades siguiendo la plantilla de arriba, añadiendo los queries según sean necesarios.

Cómo se puede ver en el código, los métodos que retornan datos utilizan el enclosure LiveData<>, esto es un componente de la las librerías de AndroidX que permiten que un objeto sea observable dentro del contexto que fue creado (actividad, fragmento, ListItem) y estás “consciente” del ciclo de vida de la aplicación para actualizar su contenido y destruirse cuando sea necesario para ser recolectado por el Garbage Collector de la máquina virtual del Java que implementa Android, liberando así memoria.

La creación de la base datos, puede ya ser realizada, quedando de la siguiente manera:

```
@Database(entities = arrayOf(Usuario::class,
                             Medicamento::class,
                             Medico::class,
                             FichaContacto::class,
                             Establecimiento::class,
                             CitaMedica::class,
                             Tratamiento::class,
                             Toma::class
),
        version = 1)
abstract  class MMRDataBase : RoomDatabase() {


    abstract fun usuarioDao(): UsuarioDao
    abstract fun medicamentoDao() : MedicamentoDao
    abstract fun medicoDao() : MedicoDao
```

```
    abstract fun fichaContactoDao() : FichaContactoDao
    abstract fun establecimientoDao() : EstablecimientoDao
    abstract fun citaMedicaDao() : CitaMedicaDao
    abstract fun tratamientoDao() : TratamientoDao
    abstract fun tomaDao() : TomaDao
    companion object {
        @Volatile private var instance : MMRDataBase? = null

        fun getInstance(context: Context) : MMRDataBase =
                instance ?: synchronized(this){
                    instance ?: buildDatabase(context).also { instance =
it }
                }

        private fun buildDatabase(context : Context) =
                Room.databaseBuilder(context.applicationContext,
                        MMRDataBase::class.java, "MMRDatabase.db")
                        .build()
    }
}
```

La clase que da soporte a la base de datos está lista, ahora solo falta instanciarla para crear la base de datos en sí. Cuando la base de datos es creada verifica el número de versión de esta, si el número no ha existido se crea una base de datos nueva vacía, sí existe entonces retorna una instancia de la base de datos actual, sí el número es superior entonces se crea una base de datos nueva a la cual el programador es responsable de crear los mecanismos necesarios para migrar la versión antigua a la nueva.

Resultados:
La base de datos implementada está lista para ser utilizada, ahora es posible realizar operaciones sobre ella, así también se ha creado una parte del modelo, aquí podemos ver que, aunque se hayan creado más clases que si hubiéramos utilizado SQLite, cada uno de los elementos de ROOM están bien definidos por lo que su mantenimiento es mucho más fácil.

4.1.6 Creación de la capa de repositorios

Descripción/Requisitos:
Necesitamos crear la capa de repositorios para seguir el patrón MVVM en Android, esta capa interactuará directamente con los DAOs y será la encargada de devolver los datos a la capa superior. La capa de repositorio será agnóstica de la plataforma, es decir deberá evitar hacer uso de librerías exclusivas del SDK de Android. Las consultas realizadas con ROOM no pueden ser realizadas desde el hilo principal de la aplicación, pues en caso de ser muy pesadas por ejemplo cuando retornan una gran cantidad de datos, pueden bloquear la interfaz gráfica, creando un mensaje de ANR (Android Not Responding) cuando tardan más de 5 segundos, en estos casos el usuario puede matar la aplicación o el propio sistema puede hacerlo según sea la cantidad de recursos con los que cuente el dispositivo.

Los repositorios permiten realizar las consultas en hilos secundarios, pudiendo el programador definir cuál será el mejor mecanismo para lograr esto.

Ejecución:
La aplicación cuenta con una única fuente de datos, que es la base de datos, así que el repositorio debe acceder a esta, inicializarla en caso de ser necesario y realizar las operaciones que le sean solicitadas desde la capa superior.

Utilizamos la implementación de AsyncTask, realizar tareas en segundo plano sin intervenir en el hilo principal ni la interfaz gráfica. Creando una clase por cada una las acciones a realizar, el constructor recibirá el DAO del elemento a manipular. AsyncTask se encarga de crear los hilos cuando una nueva instancia es ejecutada y de destruirlos una vez que termina de realizar la tarea asignada, además de poder retornar un valor a modo de indicador de progreso que puede ser utilizado para dar un indicativo visual del estado de la tarea.

La clase que usaremos como plantilla quedará de la siguiente manera:
UsuarioRepository.kt

```
class UsuarioRepository(application: Application) {
    val usuarioDao : UsuarioDao

    init{
        val database = MMRDataBase.getInstance(application)
        usuarioDao = database.usuarioDao()
    }


    fun insert(usuario : Usuario){
        InsertUsuarioAsynTask(usuarioDao).execute(usuario)
    }

    fun update(usuario : Usuario){
        UpdateUsuarioAsyncTask(usuarioDao).execute(usuario)
    }

    fun delete(usuario : Usuario){
        DeleteUsuarioAsyncTask(usuarioDao).execute(usuario)
    }

    fun deleteAllUsuarios(){
        DeleteAllUsuariosAsyncTask(usuarioDao).execute()
    }

    fun getAllUsuarios() : LiveData<List<Usuario>>{
        return usuarioDao.getAllUsuarios()
    }

    fun getUsuario( id : Int) : LiveData<Usuario>{
        return usuarioDao.getUsuario(id)
    }

    fun getLastInsertedUserID() : LiveData<Long>{
        return usuarioDao.getLastUserID()
    }
```

```
    private class InsertUsuarioAsynTask constructor(private val
usuarioDao: UsuarioDao) : AsyncTask<Usuario, Void, Void>(){
        override fun doInBackground(vararg usuarios: Usuario): Void? {
            usuarioDao.insert(usuarios[0])
            return null
        }
    }

    private class UpdateUsuarioAsyncTask constructor(private val
usuarioDao: UsuarioDao) : AsyncTask<Usuario, Void, Void>(){
        override fun doInBackground(vararg params: Usuario): Void? {
            usuarioDao.update(params[0])
            return null
        }
    }

    private class DeleteUsuarioAsyncTask constructor(private val
usuarioDao : UsuarioDao) : AsyncTask<Usuario, Void, Void>(){
        override fun doInBackground(vararg params: Usuario): Void? {
            usuarioDao.delete(params[0])
            return null
        }
    }

    private class DeleteAllUsuariosAsyncTask constructor(private val
usuarioDao: UsuarioDao ) : AsyncTask<Void, Void, Void>(){
        override fun doInBackground(vararg params: Void?): Void? {
            usuarioDao.deleteAllUsuarios()
            return null
        }
    }

}
```

Es posible notar que aquellas operaciones dentro del DAO que retornan objetos con la propiedad de LiveData<> no requieren una clase que implemente AsyncTask y es que las librerías de Android Jetpack se encargan de manejar internamente las consultas en segundo plano y que serán enviadas a las vistas, así el programador debe preocuparse únicamente por aquellas consultas de inserción o actualización.

Resultados:

Se creó un repositorio para cada DAO, así que se ha creado la capa de repositorios para la aplicación, ahora es necesario crear la última capa del modelo MVVM: los ViewModels.

4.1.7 Creación de la capa de ViewModels

Descripción/Requisitos:

Debemos crear la última capa del patrón MVVM: Los ViewModels, esta será utilizada como medio canal de comunicación entre las vistas y el modelo, siendo el "code-behind" de cada actividad el lugar donde realizaremos la consulta, inserción y actualización de datos.

Ejecución:

La creación de un ViewModel es bastante simple cuando se tiene un solo repositorio, pues se podría decir que es una capa que encapsula los métodos del repositorio, la lógica que determina a que método llamar desde el ViewModel se realiza usualmente desde el código de la actividad.

UsuarioViewModel.

```
class UsuarioViewModel (application: Application) :
AndroidViewModel(application){
    val repository : UsuarioRepository
    val allUsuarios : LiveData<List<Usuario>>

    init{
        repository = UsuarioRepository(application)
        allUsuarios = repository.getAllUsuarios()
    }
    fun insert(usuario: Usuario){
        repository.insert(usuario)
    }
    fun update(usuario: Usuario){
        repository.update(usuario)
    }
    fun delete(usuario: Usuario){
        repository.delete(usuario)
    }
    fun getUsuario(id : Int) : LiveData<Usuario>{
       return repository.getUsuario(id)
    }
}
```

Es posible ver que el constructor de esta clase inicializa también el repositorio de usuario, es decir una inicialización en cadena, el ViewModel se crea e inicializa, se crea el repositorio que crea al DAO y finalmente el DAO crea la base de datos en caso de ser necesario. Cuando la base de datos existe el ViewModel entonces es usado únicamente para Insertar, Eliminar y Actualizar datos, pero no puede eliminar la base de datos, así que aquí tenemos un mayor nivel de seguridad que usando el camino tradicional de crear una base de datos con SQLite puro.

Resultados:

Ahora cada entidad cuenta con su respectivo repositorio, ahora será posible comenzar a manipular la base de datos desde alto nivel ya sea desde un elemento de interfaz gráfica cómo una actividad o desde servicios, providers y receivers.

4.1.8 Creación de los elementos gráficos

Descripción/Requisitos:

Usando algún programa de diseño se deberá de generar los iconos y cualquier otro elemento grafico que pueden ser requerido por la aplicación hasta este punto, como por ejemplo iconos que representan el tipo de medicamentos.

Ejecución:

Usando Adobe Illustrator se creó un archivo "base" el cual definirá las "dimensiones" del canvas para crear nuestros iconos, así también debemos definir el grosor de línea para que todos los iconos que se creen tengan el mismo aspecto sin importar cual sea su escala cuando sean usados.

El grosor de línea será de 11 pixeles, todos los iconos deberán ser del mismo color, creándolos en negro para poder cambiar su color desde Android Studio de una forma simple y rápida. El tamaño del canvas será de 512 x 512 Unidades.

Una vez que se ha creado un icono que cumpla con nuestros requerimientos debemos exportarlo desde Illustrator a Android Studio, para ello debemos guardarlo en formato .SVG y abrirlo en Android Studio desde File > New > Vector Assets y especificar la ruta y archivo de nuestra elección.

Figura 49 Creación de un icono para la aplicación en Adobe Illustrator.

Resultados:

Ahora disponemos de los elementos gráficos para nuestra aplicación en su prototipo inicial, pero además contamos con los archivos fuente para crear iconos adicionales con un mismo look and feel de una forma sencilla.

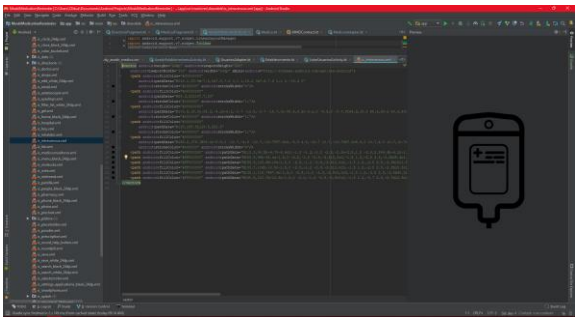

Figura 50 Importando un archivo .SVG en Android Studio.

4.1.9 Resultados de la primera iteración de desarrollo

En esta primera iteración de desarrollo se logró la aplicación del patrón del patrón MVVM en Android de forma satisfactoria, pues fue posible utilizar los diagramas creados durante el proceso de diseño de la arquitectura para tener una implementación en código clara y por tanto más rápida.

Las librerías de Android X, así como de ROOM permitieron acelerar el proceso de desarrollo en esta primera fase. Gracias a los adaptadores es posible poblar las listas desde una fuente de datos ya sea estática o dinámica, qué como pudimos ver durante el desarrollo de estos utilizamos un arreglo creado a mano, pero en cuanto se implementó la base de datos fue posible realizar el cambio de forma fácil a el ViewModel como fuente de datos cambiando una sola línea de código en cada uno de los fragmentos donde hay un ReclyclerView.

En este punto únicamente las listas muestran los elementos registrados en la base de datos, tenemos que refinar el proceso de visualización y registro. Es decir, crear utilizar los formularios de registro para añadir a la base de datos su respectiva entidad y de igual manera crear usar las actividades correspondientes para ver elementos de forma individual.

4.2 Refinamiento de la capa de Vistas (Activity/Fragment)

Debemos crear los mecanismos para registrar nuevos elementos dentro de la base de datos, sí bien esto es una tarea relativamente fácil utilizando los ViewModels y las actividades ya creados, debemos utilizar el "code-behind" de cada actividad para asegurarnos que cada dato sea válido antes de registrarlo pues estos datos serán usados más adelante para realizar tareas en específico, (como abrir un mapa o el marcador de teléfono).

4.2.1 Creación de mecanismos para la toma de fotografías y selección de imágenes desde galería

Descripción/Requisitos:

En las reglas de negocio se especifica que la aplicación debe ser capaz de tomar fotografías, así como la opción de elegir imágenes de la galería, ya sea para identificar usuarios o para incluirlas en el registro de medicamentos, por tanto, debemos crear los mecanismos necesarios para esto.

Ejecución:

Comenzaremos creando un menú genérico que será usado para elegir el origen de las imágenes: La galería del dispositivo o desde la cámara, el usuario una vez que elija uno se realizará el respectivo comportamiento.

menu_add_photo.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/add_photo_item"
        app:showAsAction="always"
        android:title="Añadir foto"
        android:icon="@drawable/ic_add_a_photo_white_24dp"/>

</menu>
```

Ahora podemos incluir este menú en la toolbar de cualquier actividad con el siguiente código:

```
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_add_photo,menu)
        return true
    }
```

Ahora al visualizar el layout del menú, cuando sea añadido a cualquier actividad o fragmento se verá de la siguiente manera:

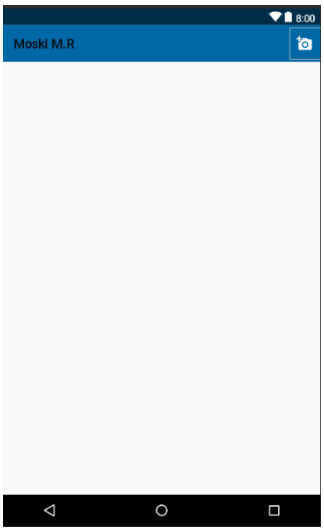

Figura 51 Menú para añadir imágenes en una actividad o fragmento.

Comencemos con la posibilidad de elegir una imagen de una galería, para esto creamos un método que toma como parámetro Intent.ACTION_PICK que indica al sistema que disparar el selector de acciones, en nuestro caso especificamos que deseamos elegir una imagen usando selectPictureIntent.type y pasando = "image/*", esto especifica que deseamos que cualquier aplicación disponible en el dispositivo para elegir una imagen sea mostrada al usuario, el parámetro Intent.EXTRA_MIME_TYPES utiliza un arreglo donde especificamos los formatos de imágenes soportados, en nuestro caso .jpg y .png.

```
    fun pickFromGallery(){
        Intent(Intent.ACTION_PICK).also { selectPictureIntent ->
            selectPictureIntent.type = "image/*"
            val mimeTypes = arrayOf("image/jpg", "image/png")
            selectPictureIntent.putExtra(Intent.EXTRA_MIME_TYPES,
mimeTypes)

startActivityForResult(selectPictureIntent,CodigosDeSolicitud.SELECCIONAR
_IMAGEN)

        }
    }
```

El método startActivityForResult como su nombre indica inicializa una actividad esperando un resultado, un código de identificación es usado para saber qué actividad termina y recibe el resultado en el método onActivityResult a usando un objeto intent.

Con esto podemos seleccionar imágenes de la galería, pero en el caso de que el usuario selecciones la cámara como fuente de la imagen, el siguiente método es utilizado:

```
    private fun dispatchTakePicktureIntent(){
        Intent(MediaStore.ACTION_IMAGE_CAPTURE).also {takePictureIntent -
>
```

```
            takePictureIntent.resolveActivity(packageManager)?.also {
                val photoFile : File? = try{
                    createImageFile()
                } catch (ex : IOException){
                    null
                }
                photoFile?.also {
                    val photoURI: Uri = FileProvider.getUriForFile(
                            this,
                            "com.kps.spart.android.fileprovider",
                            it
                    )
                    takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
photoURI)
                    startActivityForResult(takePictureIntent,
CodigosDeSolicitud.ANADIR_FOTOGRAFIA)
                }
            }
        }
    }
```

El método crea un intent que recibe como parámetro la constante MediaStore.Action_IMAGE_CAPTURE, el cual al ser inicializado abre la cámara del dispositivo y permite al usuario tomar una fotografía, posteriormente cierra la cámara y regresa el archivo binario de la fotografía en un intent, el cual debemos procesar en onActivityResult.

El método onActivityResult queda de la siguiente manera:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
        if(requestCode == CodigosDeSolicitud.ANADIR_FOTOGRAFIA &&
resultCode == Activity.RESULT_OK){
            setPic()
            displayPic()
        }
        else if(requestCode == CodigosDeSolicitud.SELECCIONAR_IMAGEN &&
resultCode == Activity.RESULT_OK){
            val selectedImageUri = data?.data
            val filePathColum = arrayOf(MediaStore.Images.Media.DATA)
            val cursor = this.contentResolver.query(selectedImageUri,
filePathColum,null, null, null)
            cursor.moveToFirst()
            val columnIndex = cursor.getColumnIndex(filePathColum[0])
            val imgDecodableString = cursor.getString(columnIndex)
            cursor.close()

            val bmOptions = BitmapFactory.Options().apply {
                inJustDecodeBounds = true
                BitmapFactory.decodeFile(imgDecodableString,this)
                val photoW: Int = outWidth
                val photoH: Int = outHeight

                val scaleFactor : Int = Math.min(photoW /
resources.getDimension(R.dimen.UserProfileImageSingle).toInt() , photoH /
resources.getDimension(R.dimen.UserProfileImageSingle).toInt() )
```

```
                inJustDecodeBounds = false
                inSampleSize = scaleFactor
                inPurgeable = true
            }

            val imageFile : File? = try{
                createImageFile()
            } catch (ex : IOException){
                null
            }
            val out = FileOutputStream(imageFile)
            val exif = ExifInterface(imgDecodableString)
            val orientation =
exif.getAttributeInt(ExifInterface.TAG_ORIENTATION,ExifInterface.ORIENTAT
ION_UNDEFINED)

            BitmapFactory.decodeFile(imgDecodableString, bmOptions).also
{ bitmap ->

                var rotatedBitmap : Bitmap? = null
                when (orientation) {
                    ExifInterface.ORIENTATION_ROTATE_90 -> {
                        rotatedBitmap = rotateImage(bitmap,90f)
                    }
                    ExifInterface.ORIENTATION_ROTATE_180 -> {
                        rotatedBitmap = rotateImage(bitmap,180f)
                    }
                    ExifInterface.ORIENTATION_ROTATE_270 -> {
                        rotatedBitmap = rotateImage(bitmap, 270f)
                    }
                    ExifInterface.ORIENTATION_NORMAL -> {
                        rotatedBitmap = bitmap
                    }
                    else -> {
                        rotatedBitmap = bitmap
                    }
                }

rotatedBitmap?.compress(Bitmap.CompressFormat.JPEG,85,out)
            }
            out.close()
            displayPic()
        }
    }
```

En ambos casos la imagen seleccionada es guardada en el archivo que se creó, el cual está dentro del almacenamiento interno de la aplicación, el usuario puede tener tantas imágenes cómo desee, pero estas se eliminaran al desinstalar la aplicación.

Debemos optimizar la imagen capturada con la cámara pues es posible que esta tenga una resolución muy grande, que, si bien se vería sin problemas en la aplicación, el consumo de memoria sería muy grande, debemos re-escalar la imagen a la resolución de la vista que es dinámica según sea la de la pantalla del dispositivo.

Para ello nos apoyamos del siguiente método:

```
    private fun setPic(){

        val bmOptions = BitmapFactory.Options().apply {
            inJustDecodeBounds = true
            BitmapFactory.decodeFile(mCurrentPhotoPath,this)
            val photoW: Int = outWidth
            val photoH: Int = outHeight

            val scaleFactor : Int = Math.min(photoW /
resources.getDimension(R.dimen.UserProfileImageSingle).toInt() , photoH /
resources.getDimension(R.dimen.UserProfileImageSingle).toInt() )

            inJustDecodeBounds = false
            inSampleSize = scaleFactor
            inPurgeable = true
        }

        val exif = ExifInterface(mCurrentPhotoPath)
        val orientation =
exif.getAttributeInt(ExifInterface.TAG_ORIENTATION,ExifInterface.ORIENTAT
ION_UNDEFINED)


        BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions)?.also {
bitmap ->

            var rotatedBitmap : Bitmap? = null
            when (orientation) {
                ExifInterface.ORIENTATION_ROTATE_90 -> {
                    rotatedBitmap = rotateImage(bitmap,90f)
                }
                ExifInterface.ORIENTATION_ROTATE_180 -> {
                    rotatedBitmap = rotateImage(bitmap,180f)
                }
                ExifInterface.ORIENTATION_ROTATE_270 -> {
                    rotatedBitmap = rotateImage(bitmap, 270f)
                }
                ExifInterface.ORIENTATION_NORMAL -> {
                    rotatedBitmap = bitmap
                }
                else -> {
                    rotatedBitmap = bitmap
                }
            }
            rescaleImage(rotatedBitmap)
        }
    }
```

El método calcula un factor de estala basado en:

$$Factor\ de\ escala = Minimo(\frac{WI}{WV}, \frac{HI}{HV})$$

Donde WI = Ancho de la imagen, WV = Ancho de la vista, HI = Alto de la imagen y HV = Alto de la vista, por último, debemos mostrar la imagen en una vista del tipo ImageView.

Resultados:
Ahora el usuario puede elegir el origen de la imagen que desee incluir en el registro de un perfil o de un nuevo medicamento. Así también podemos eliminarla en caso de ser necesario previniendo que la aplicación genere archivos basura dentro de su espacio.

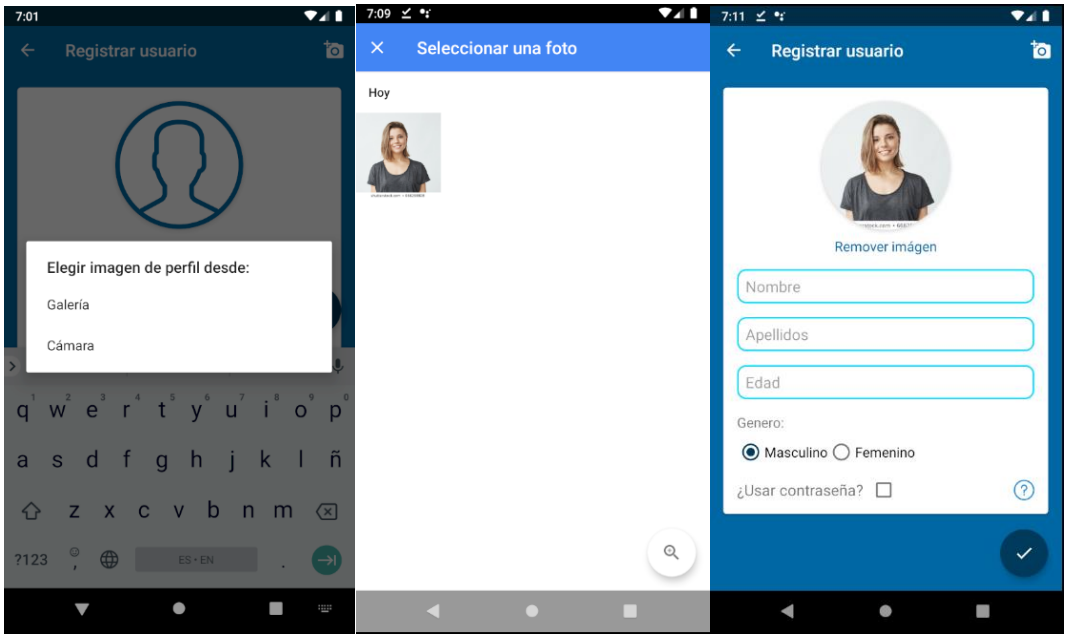

Figura 52 Seleccionando una foto de la galería para usarlo dentro de la aplicación.

4.2.1 Registrar elementos en la base datos usando los ViewModel

Descripción/Requisitos:
Debemos crear el código necesario que valide los datos provenientes de los formularios contenidos en las actividades, crear un nuevo objeto para cada entidad y registrarlo utilizando el ViewModel. Todos los datos que forman parte de las entidades son datos primitivos, la fotografía no será almacenada cómo datos binarios, sino que almacenaremos la dirección del archivo que guarda la imagen cómo una cadena de texto.

Ejecución:
Debemos declarar una variable global que contendrá el ViewModel respectivo de la actividad:

```
val usuarioViewModel =
ViewModelProviders.of(this@RegistrarUsuarioActivity).get(UsuarioViewModel
::class.java)
```

El ViewModel necesita del contexto de la actividad o fragmento que lo está instanciando y el nombre de la clase que extiende el ViewModel.

Se creó un método que recupera los datos de las cajas de texto (EditTextbox), Checkboxes, Listas y en sí de cualquier otra vista de la cual queramos utilizar para obtener información del usuario. Se crea un objeto del mismo tipo que soporta el ViewModel y se asignan sus valores usando los datos recuperados.

```
savePerfilFAB.setOnClickListener{
            val usuario : Usuario


                 usuario = Usuario(0)

            if(nombreUsuarioET.text.isEmpty() ||
apellidoUsuarioET.text.isEmpty() || Edad.text.isEmpty()){

Snackbar.make(it,getString(R.string.nombre_apellido_necesario),Snackbar.L
ENGTH_LONG).show()
            }else{
                usuario.nombre = nombreUsuarioET.text.toString()
                usuario.apellidos = apellidoUsuarioET.text.toString()
                usuario.imagen = mCurrentPhotoPath

                try{
                    usuario.edad = Edad.text.toString().toInt()
                }catch( e : Exception){


Snackbar.make(it,getString(R.string.edad_necesaria),Snackbar.LENGTH_SHORT
).show()
                }

                val selectedRadioButton =
findViewById<RadioButton>(GeneroRadioGroup.checkedRadioButtonId)

                usuario.genero = selectedRadioButton.text.toString()

                if(usaPasswordCheckbox.isChecked){
                    if(PasswordEditText.text.isEmpty() ||
RecoveryET.text.isEmpty()){

Snackbar.make(it,getString(R.string.contrasena_necesaria),Snackbar.LENGTH
_LONG).show()
                    }else{
                        usuario.password =
PasswordEditText.text.toString()
                        usuario.email_recuperacion =
RecoveryET.text.toString()
                        saveUserToDB(usuario)
                    }
                }else{
                    saveUserToDB(usuario)
                }
            }

        }
```

Desde el método podemos validar los datos que son ingresados desde la interfaz gráfica, en este caso no es posible ingresar cadenas vacías, si el usuario intenta hacerlo una leyenda es mostrada advirtiendo el caso. Hacemos uso de un método auxiliar, el cual recibe la entidad creada y determina si esa entidad debe ser actualizada o insertada como un nuevo registro en la base de datos:

```
    private fun saveUserToDB(usuario: Usuario){
        if(intent.hasExtra("USER_ID")){
            usuarioViewModel.update(usuario)

        }else{
            usuarioViewModel.insert(usuario)
        }
        setResult(Activity.RESULT_OK)
        finish()
    }
```

El método de arriba termina la actividad actual y regresa al usuario al listado de entidades en este caso el listado de usuarios.

Resultados:
Ahora es posible registrar elementos desde la interfaz gráfica de usuario, es posible notar que el proceso es bastante sencillo y rápido, pues únicamente debemos encargarnos de crear un objeto/entidad utilizando datos válidos y el ViewModel se encarga de registrarlos en la base de datos en una simple llamada. El ViewModel actualiza automáticamente las listas así que el usuario puede tocar cualquier elemento de ellas y realizar una acción sobre ese elemento en específico.

4.2.1 Poblar vistas para entidades individuales

Descripción/Requisitos:
Los elementos registrados solo pueden ser visualizados en su elemento de la lista, debemos poder ver los detalles de cada uno en su respectiva actividad.

Ejecución:
Primero debemos crear un método que consiga una referencia al ID del elemento que esté vinculado a su espacio en la lista e inicializar la actividad para mostrar los detalles de esa entidad.

el método toma el adaptador definido en la lista y le añade un escucha que recibe el evento del toque de usuario, utiliza el método getAt declarado en la definición de la clase del adaptador. Finalmente se crea un intent que adjunta el id de la entidad e inicializa la actividad.

En la actividad de la entidad recuperamos el ID desde el intent enviado e instanciamos una variable de su respectivo ViewModel, después usamos el ID para recuperar un objeto que es usado para poblar los elementos de la interfaz:

```
        val user_id = intent.getIntExtra("USER_ID", -1)
```

```
        usuarioViewModel =
ViewModelProviders.of(this).get(UsuarioViewModel::class.java)

        usuarioActualLive =  usuarioViewModel.getUsuario(user_id)
        usuarioActualLive.observe(this, Observer {
            populateUserFieldsFromDB(it)

        })
```

El método auxiliar que pobla la interfaz queda:

```
    private fun populateUserFieldsFromDB(usuario: Usuario?){

            NombreApellidosUsuarioTV.text = "${usuario?.nombre}
${usuario?.apellidos}"
            GeneroUsuarioTV.text = usuario?.genero
            EdadUsuarioTV.text = usuario?.edad.toString()
            val valueInPixels =
resources.getDimension(R.dimen.UserProfileImageSingle)

            if(usuario?.imagen.isNullOrEmpty()){
                PerfilIV.setImageResource(R.drawable.ic_user)
            }else{

setPic(usuario?.imagen!!,valueInPixels.toInt(),valueInPixels.toInt())
            }
    }
```

Resultados:

Ahora tocar los elementos de la lista, su respectiva actividad es inicializada usando los datos del objeto recuperado, de nuevo vemos cómo el uso del ViewModel simplifica enormemente el proceso de visualización de datos.

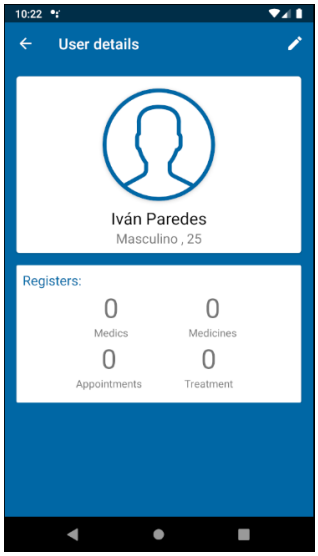

Figura 53 Detalles de un perfil poblado dinámicamente.

4.2.1 Editar elementos existentes en la base de datos

Descripción/Requisitos:

Los elementos registrados no pueden ser modificados una vez que se han insertado en la base de datos, debemos actualizar nuestros mecanismos para poder obtener una referencia a una entidad en concreto y cambiar sus datos e ingresarlo de nuevo en la base de datos.

Ejecución:

En la tarea anterior creamos un método que registra o actualiza una entidad, pero ¿Cómo distingue cuando insertar o actualizar un elemento?, esto es utilizando la variable que representa la llave primaria en su respectiva tabla. Cuando creamos un objeto desde la actividad y le asignamos una llave primaria que ya existe y tratamos de ingresarlo en el ViewModel este actualiza su respectiva tupla. En caso contrario es ingresado como un registro nuevo.

Crear un objeto usando su constructor predeterminado evita que exista una colisión por repetición de valor de la llave primaria. Asignar el valor de forma manual habilita el comportamiento antes mencionado.

Debemos editar el código de registro para verificar sí existe una referencia en el intent que inicializa la actividad, en caso de ser así debemos recuperar el objeto asociado y poblar las vistas. Eso se realiza con el siguiente código:

```
        val usuarioViewModel =
ViewModelProviders.of(this@RegistrarUsuarioActivity).get(UsuarioViewModel
::class.java)

        if(intent.hasExtra("USER_ID")){
            title = getString(R.string.editar_usuario)
            usuarioActualLive =
usuarioViewModel.getUsuario(intent.getIntExtra("USER_ID", -1))
            usuarioActualLive.observe(this,
android.arch.lifecycle.Observer {


                if(mNombre != null){
                    nombreUsuarioET.setText(mNombre,
TextView.BufferType.EDITABLE)
                }else{
                    nombreUsuarioET.setText(it?.nombre,
TextView.BufferType.EDITABLE)
                }

                if(mApellido != null){
                    apellidoUsuarioET.setText(mApellido,
TextView.BufferType.EDITABLE)
                }else{
                    apellidoUsuarioET.setText(it?.apellidos,
TextView.BufferType.EDITABLE)
                }

                if(mEdad != null){
                    Edad.setText(mEdad, TextView.BufferType.EDITABLE)
```

```
                }else{

Edad.setText(it?.edad.toString(),TextView.BufferType.EDITABLE)
                }

                val genero : String?
                if(mGenero != null){
                    genero = mGenero
                }else{
                    genero = it?.genero
                }

                if(genero!!.equals(getString(R.string.masculino))){
                    GeneroRadioGroup.check(R.id.masculinoRB)
                }else{
                    GeneroRadioGroup.check(R.id.femeninoRB)
                }


                if(mPassword != null){
                    usaPasswordCheckbox.isChecked = true

PasswordEditText.setText(mPassword,TextView.BufferType.EDITABLE)

                }else{
                    if(!it?.password.isNullOrEmpty()){
                        usaPasswordCheckbox.isChecked = true
                    }

PasswordEditText.setText(it?.password,TextView.BufferType.EDITABLE)
                }

                if(mRecoveryEmail != null){

RecoveryET.setText(mRecoveryEmail,TextView.BufferType.EDITABLE)
                }else{

RecoveryET.setText(it?.email_recuperacion,TextView.BufferType.EDITABLE)
                }


                if(!it?.imagen.isNullOrEmpty() &&
mCurrentPhotoPath.isNullOrEmpty()){
                    mCurrentPhotoPath = it?.imagen!!
                    if(!mCurrentPhotoPath.isNullOrEmpty()){
                        displayPic()
                    }
                }

            })

        }else{
            title = getString(R.string.registrar_usuario)
        }
```

Por último, debemos actualizar el código que toma los datos de las vistas para definir sí el objeto debe ser creado como nuevo o establecer el ID desde el ViewModel:

```
            if(intent.hasExtra("USER_ID")){
                usuario = usuarioActualLive.value!!
            }else{
                usuario = Usuario()
            }
```

Resultados:
Ahora es posible editar las entidades previamente registradas, observamos que la capa de vistas es la encargada de administrar los datos provenientes desde el ViewModel, pero no hemos actualizado o realizado cambios en las capas inferiores, demostrando una vez más las ventajas del patrón MVVM.

4.2.1 Soportar cambios por rotación de la pantalla durante el registro de datos

Descripción/Requisitos:
Cuando un usuario está en alguno de los formularios para el registro de nuevas entidades es posible que se realice un cambio en la orientación del dispositivo, esto provoca que los datos ingresados se eliminen otro caso en lo que esto sucede es cuando teniendo formularios con datos y entra una llamada o inicial cualquier otra actividad automática el sistema puede llegar a eliminar el estado actual de nuestra aplicación. El motivo de esto porque es el comportamiento predeterminado de Android, al rotar la pantalla la actividad es destruida y creada nuevamente, por tanto, hay que implementar un mecanismo de persistencia para todas las variables en memoria.

Ejecución:
Google propone una solución bastante simple para atacar estos problemas, es la implementación de los métodos onSaveInstanceState() y onRestoreInstanceState(). El primero toma los valores especificados por usuario y los almacena en un objeto Bundle. Un Bundle es un objeto utilizado para pasar datos entre varias actividades o en este caso a sí misma. Depende del programador cual variable puede ser sostenida pero una gran ventaja es que los Bundle soportan cualquier tipo de datos. El Bundle se crea antes de crear la actividad y usualmente es destruido por el recolector de basura de la máquina virtual de Java de Android.

El método onSaveInstanceState() utiliza claves para añadir objetos al Bundle de la actividad, mientras que el método onRestoreInstanceState() es llamado cuando Android tiene que recrear una actividad después de que ha sido “matada” por el sistema, más no por el usuario, por ejemplo cuando usa el botón back para regresar una pantalla y luego volver a lanzar la actividad.

La implementación de los métodos quedaría de la siguiente manera:

```
    override fun onSaveInstanceState(outState: Bundle?) {
        super.onSaveInstanceState(outState)

        outState?.run {
            putString("ActualPhotoPath", mCurrentPhotoPath)
```

```
            putString("NombreUsuarioActualizado",
nombreUsuarioET.text.toString())
            putString("ApellidosUsuarioActualizado",
apellidoUsuarioET.text.toString())
            putString("EdadUsuarioActualizado", Edad.text.toString())

            val selectedRadioButton =
findViewById<RadioButton>(GeneroRadioGroup.checkedRadioButtonId)
            putString("GeneroUsuarioActualizado",
selectedRadioButton.text.toString() )

            putString("PassWordUsuarioActualizado",
PasswordEditText.text.toString())
            putString("RecoveryEmailUsuarioActualizado",
RecoveryET.text.toString())

        }
    }

    override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
        super.onRestoreInstanceState(savedInstanceState)

        savedInstanceState?.run {
            mCurrentPhotoPath = getString("ActualPhotoPath")!!
            if(!mCurrentPhotoPath.isEmpty()){

                displayPic()
            }

            mNombre = getString("NombreUsuarioActualizado")
            mApellido = getString("ApellidosUsuarioActualizado")
            mEdad = getString("EdadUsuarioActualizado")
            mGenero = getString("GeneroUsuarioActualizado")
            mPassword = getString("PassWordUsuarioActualizado")
            mRecoveryEmail = getString("RecoveryEmailUsuarioActualizado")

        }
    }
```

Resultados:

Ahora la aplicación puede mantener los datos ingresados en los formularios utilizando el sistema de persistencia de google, con este paso comenzamos a darle robustes a la aplicación anadiendo caracteristicas que hacen más simple su uso.

4.2.2 Eliminación de registros en la base de datos

Descripción/Requisitos:

Ahora es posible añadir y modificar nuevas entidades dentro de la base de datos y visualizarlas en sus respectivas actividades, ahora es necesario implementar mecanismos que nos permitan eliminar de forma permanente las entidades registradas, así como los archivos generados vinculados en caso de haberlos.

Ejecución:

El proceso para eliminar una entidad es bastante sencillo pues de nuevo la implementación del ViewModel nos permite realizarlo con pocas líneas de código:

```
    private fun deleteUser(){
        if(usuarioActualLive.hasObservers()){

            if(!usuarioActualLive.value?.imagen!!.isEmpty()){
                deleteImageFile(usuarioActualLive.value?.imagen)
            }


usuarioActualLive.removeObservers(this@DetallesPerfilActivity)
            usuarioViewModel.delete(usuarioActualLive.value!!)
            finish()
        }
    }
```

El código se incluyó dentro de cada una de las actividades que muestran los detalles de las entidades registradas, tomando el objeto cargado en el ViewModel y removiendo los "Observadores", estos observadores son añadidos cuando se obtiene el objeto con el ViewModel y se pasa al método que pobla la lista. Debemos remover los observadores antes de eliminar la entidad porque de no ser así causarían un error en la aplicación al tratar de mostrar un objeto que ya es nulo.
Finalmente se termina la actividad con el método finish().

El método anterior debe ser llamado explícitamente por el usuario, para ello haremos uso del menú previamente especificado en la actividad para editar la entidad, añadiendo un nuevo caso donde buscaremos el ítem utilizado para eliminar. Eliminar la entidad sin una previa confirmación del usuario puede ser peligroso, pues puede que el usuario cometa un error sin querer y termine eliminando una entidad para accidente.

El siguiente código crea un dialogo preguntando la confirmación de la acción de eliminación, sí el usuario hace un clic sí, entonces se elimina la entidad en caso contrario no se realiza ninguna acción.

Resultados:

Los objetos registrados en la base de datos ahora pueden ser eliminados, llevándose consigo toda la información y archivos vinculados a ellos (Imágenes o fotografías). Al haber definido las entidades con el atributo de eliminación en cascada, sí un registro en una tabla padre es eliminado todos los registros en las tablas hijas también son eliminados.

4.2.3 Creación de clase auxiliar para almacenamiento y despliegue de ubicaciones usando Google Maps

Descripción/Requisitos:

Se deberá crear una clase que permita incluir soporte para mapas dentro de la aplicación, pudiendo el usuario seleccionar su ubicación, añadir y actualizar marcadores dentro de estos mapas. Los mapas podrán ser incluidos como actividades completas o como fragmentos dentro de otras actividades. Se deberá de brindad la posibilidad de que sí un mapa contiene un marcador este deberá tener acciones de interacción con otras aplicaciones externa como, por ejemplo, abrir alguna aplicación de GPS y calcular la ruta para llegar a dicho punto.

Ejecución:

Incluir un mapa en una aplicación Android es un proceso elaborado pues dependemos de distintos servicios de Google que son externos a la aplicación en sí, ya que es Google quien se encarga de brindar las API's de acceso a dichos servicios desde sus propios portales de configuración desde distintos sitios web.

Se incluirán primero los Google Play Services, las cuales son librerías que permiten comunicar nuestra aplicación con servicios de Google tanto en el sistema operativo Android como servicios externos (por mencionar algunos como búsquedas en Google, logros en videojuegos, comunicación con Google Drive, Gmail y Maps, siendo este último el de interés para esta tarea).

Podemos consultar el listado de servicios que pueden ser incluidos en la documentación oficial de Android en el siguiente enlace: https://developers.google.com/android/guides/setup

Para añadir el soporte a los Google Play Services tenemos que añadir una nueva dependencia a nuestro archivo build.gradle que incluya al repositorio Maven de Google, para ello incluimos el siguiente código debajo del conjunto allprojects, quedando así:

```
allprojects {
    repositories {
        jcenter()
        maven {
            url "https://maven.google.com"          }

        maven {
            url "https://jitpack.io"          }
        google()
    }
}
```

Ahora podemos especificar la familia de API's que deseamos utilizar para ello en nuestro módulo de aplicación (Module App) de nuestro archivo build.gradle bajo el conjunto de dependencias incluimos las siguientes líneas:

```
implementation 'com.google.android.gms:play-services-maps:16.0.0'
implementation 'com.google.android.gms:play-services-location:16.0.0'
```

La primera nos permitirá usar los mapas de Google dentro de nuestra aplicación, mientras que la segunda permite a nuestra aplicación acceder a la ubicación del usuario en sus distintas modalidades.

El siguiente paso es conseguir una API Key para Google Maps, la cual es un identificador a nivel aplicación para los servidores de Google, pues devuelven respuestas a peticiones que se hagan (ubicación, direcciones, los mapas en sí, etc.…), el uso de Google Maps por terceros tiene un costo que depende de la plataforma y de la cantidad de peticiones diarias que se hagan, pero en el caso de las aplicaciones para Android, Google permite el uso de mapas estáticos (imágenes que contiene mapas) y mapas dinámicos (mapas que pueden ser escalados, rotados, visto en capas, añadir locaciones, y obtener información de locaciones) de forma ilimitada.

Podemos solicitar hasta 25 API's Keys para distintos proyectos, para esto tenemos que ingresar a la consola de Google Cloud Platform en siguiente enlace: https://console.cloud.google.com/home/dashboard?organizationId=0 y registrar un proyecto nuevo, especificando un nombre que lo identifique y una organización (que puede dejar como organización predeterminada).

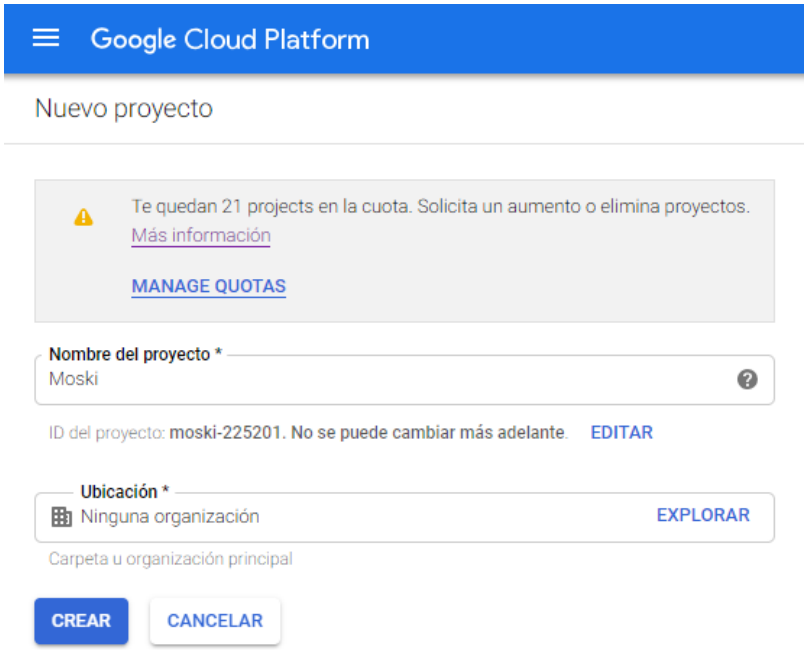

Figura 54 Formulario de registro de un proyecto en Google Cloud Platform.

Una vez creado el proyecto se nos mostrará un listado de las API's que podemos habilitar, en nuestro caso elegimos "Maps SDK for Android".

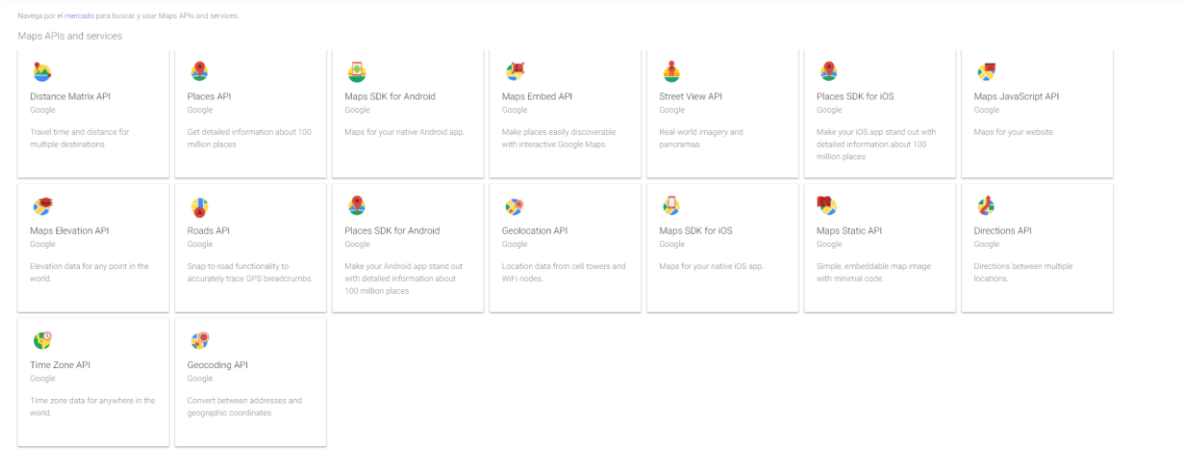

Figura 55 Listado de API's que podemos habilitar en Cloud Platform.

Finalmente debemos identificar nuestro proyecto del lado de Cloud Platform, la manera más sencilla es generando una nueva API key que será incluida en nuestro proyecto. Pero existe un riesgo, sí alguna persona descompila nuestra aplicación y visualiza los archivos de cadenas donde pueda existir nuestra API Key, puede tomarla y usarla en otras aplicaciones o servicios, pudiendo resultar en un uso excesivo de solicitudes e incluso generar gastos por uso de servicios.

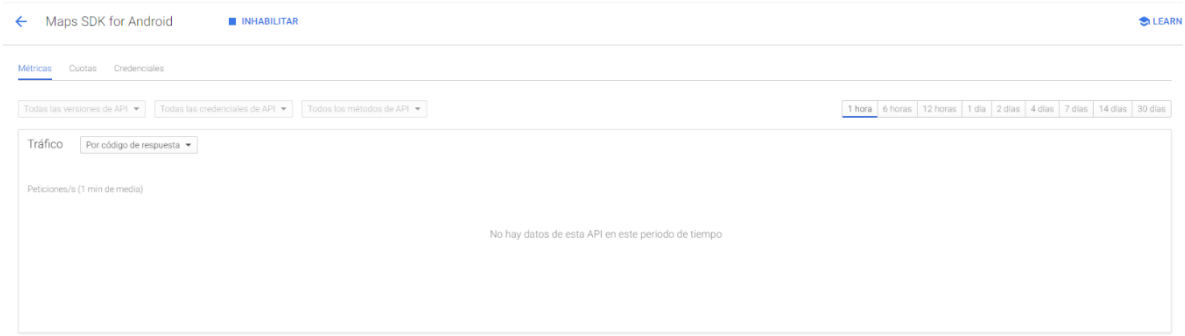

Figura 56 Panel de estadísticas de la API habilidad Maps SDK for Android.

Para ello debemos vincular la API key a la aplicación, de este modo solo nuestra aplicación podrá hacer uso de los mapas de Google y las características mencionadas con anterioridad. Para esto debemos conseguir la firma SHA1 de nuestra Key Store.

La Key Store es una firma digital que vincula a un paquete de aplicación .APK con un programador, la cual es requerida para muchos otros procesos como publicarla en la Play Store y enviar actualizaciones, de momento no entraremos en detalles sobre las Key Stores, pues serán descritas cuando lleguemos al proceso de publicar nuestra aplicación.

La forma más fácil de conseguir nuestra firma SHA1 es desde Android Studio con nuestro proyecto abierto, ejecutar el siguiente comando en la terminal:

```
keytool -list -v -keystore "%USERPROFILE%\.android\debug.keystore" -alias
androiddebugkey -storepass android -keypass Android
```

Dando como salida lo siguiente:

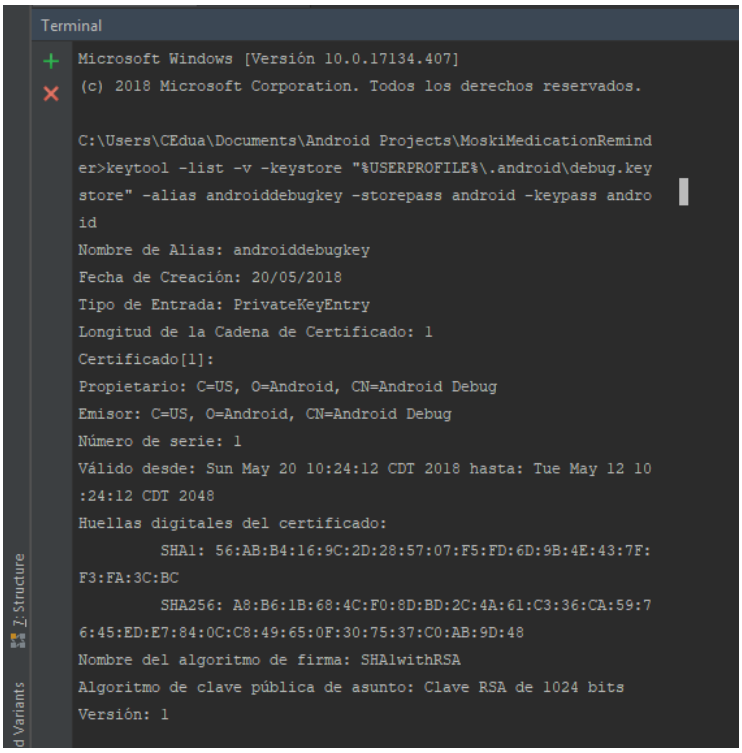

Figura 57 Firma SHA1 de la aplicación.

Registramos la clave SHA1 en la consola de Google Cloud Platform para solicitar acceso a las API's de Google Maps:

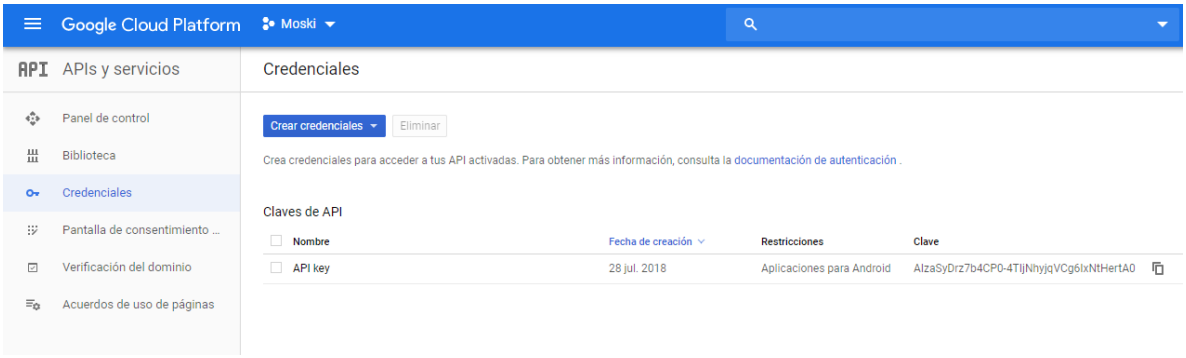

Figura 58 Firma SHA1 registrada en para uso de la Google Maps API.

Una vez registrada la llave, esta nos retornará una API Key, la cual debemos incluir en el proyecto de Android Studio de nuestra aplicación en el folder de recursos, quedando así:

```
<resources>
    <string name="google_maps_key" templateMergeStrategy="preserve"
translatable="false">AIzaSyDrz7b4CP0-4TIjNhyjqVCg6IxNtHertA0</string>
</resources>
```

Con esto ya podemos comenzar a crear mapas en la aplicación, crearemos la actividad que nos permite inicializar un mapa en la ubicación actual del usuario, seleccionarla y retornarla como valores a la actividad o fragmento que lo invocó.

Ahora al inicializar la actividad desde un intent el usuario puede elegir su ubicación:

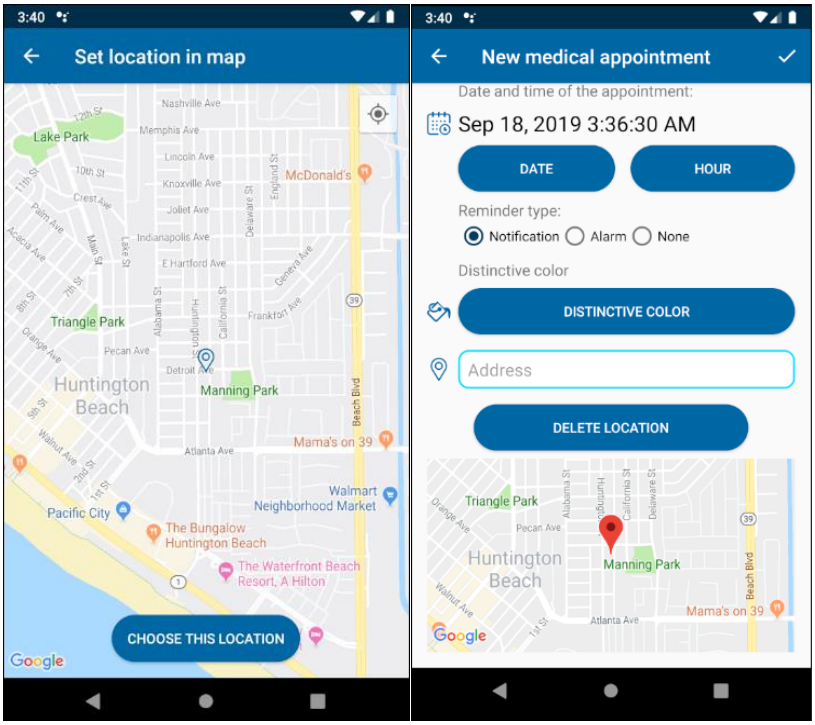

Figura 59 Incluyendo mapas dentro de la aplicación.

Resultados:
Ahora se cuenta con una actividad independiente que puede ser utilizada para que el usuario seleccione una ubicación usando un mapa y un marcador central, retornando las coordenadas latitud y longitud a la actividad o fragmento que invoca al mapa, para utilizarlos según sea el contexto actual, en nuestro caso para registrar ubicaciones de las citas o los establecimientos.

4.2.4 Resultados de la segunda iteración

La segunda iteración sirvió para refinar la capa de vistas y su comunicación con la capa de ViewModel. Se puede apreciar que el patrón MVVM facilitó esta segunda iteración, pues los procesos de inserción, edición y eliminación utilizando los formularios y actividades diseñadas para ello hicieron uso de una sola variable del tipo ViewModel que especificaba la entidad con la cual se buscaba interactuar. Pudimos centrarnos en la validación de los datos que se utilizaron para crear objetos que serían insertados en la base de datos como entidades sin tener que modificar las capas inferiores del patrón MVVM.

4.3 Interacción de la aplicación con el sistema Android

La aplicación ya permite añadir, editar y eliminar todos los elementos que habían sido definidos durante el proceso de diseño y arquitectura, pudiendo además especificar datos más complejos cómo imágenes o coordenadas geográficas, pero todos estos datos solo interactúan con la aplicación cuando el usuario la inicializa y manipula explícitamente con las distintas actividades.

El siguiente paso fue crear los mecanismos que permitan que la información contenida dentro de la aplicación pudiera interactuar con el sistema Android para poder crear notificaciones, alarmas, enviar información a otras aplicaciones, usar preferencias entre algunas otras cosas.

4.3.1 Soporte para múltiples usuarios

Descripción/Requisitos:

La aplicación actualmente permite registrar distintos usuarios, pero al hacer esto la aplicación muestra todos los elementos registrados en las listas, sin distinción de usuarios por tanto es posible eliminar o modificar entidades de forma incorrecta. Debemos implementar mecanismos que nos permitan separar los elementos de los usuarios para que sean mostrados según sea el usuario activo en ese momento, así también en caso de añadir, editar o eliminar entidades estas correspondan al usuario activo.

Ejecución:

Se creó un mecanismo que permite tener siempre a un usuario activo dentro de la aplicación, esto se consigue utilizando las preferencias compartidas del sistema Android, estas permiten utilizar archivos a los cuales se acceden por medio de una clave-valor permitiendo almacenar distintos tipos de datos: Strings, boléanos, enteros, etc. los cuales son guardados dentro del almacenamiento interno de la aplicación. Cuando las preferencias compartidas cuentan con una interfaz gráfica pueden ser utilizadas para crear pantallas de configuraciones.

En nuestro caso creamos el archivo de preferencias compartidas justo después de crear el primer usuario, para posteriormente crear tres preferencias: un valor booleano que nos permite saber sí la aplicación se está ejecutando por primera vez, la primera fecha de ejecución y otra que almacena el ID del usuario activo, el ultimo usuario registrado será siempre el usuario activo, pero será posible cambiarlo desde la pantalla de usuarios registrados.

En el código para registrar un nuevo usuario en la base de datos añadimos el siguiente código:

```
private fun saveUserToDB(usuario: Usuario){
    if(intent.hasExtra("PRIMER_USUARIO")){
        val sharedPref =
PreferenceManager.getDefaultSharedPreferences(this)

        if(sharedPref.getBoolean("firstrun", true)){
            val calendar = Calendar.getInstance()
            val sdf = SimpleDateFormat.getDateInstance()

            with(sharedPref.edit()){
                putString("firstrundate", sdf.format(calendar.time))
                putBoolean("firstrun", false)
```

```
                apply()
            }
        }
    }

    if(intent.hasExtra("USER_ID")){
        usuarioViewModel.update(usuario)

    }else{
        usuarioViewModel.insert(usuario)
    }
    setResult(Activity.RESULT_OK)
    finish()
}
```

Debemos conseguir el ID el ultimo usuario registrado, en las tareas anteriores mencionábamos que este era asignado automáticamente por el constructor de la entidad cuando era registrado en la base de datos usando el ViewModel, por tanto, no podemos salvarlo en el este punto usando la preferencia. Tenemos que actualizar el ViewModel de la entidad Usuario para conseguir el ID, recordemos que cuando creamos la entidad definimos que el ID sería auto incrementable, así que el ultimo usuario registrado será aquel que tenga el ID más grande en la tabla Usuario.

Actualizamos el DAO de la tabla Usuario, para crear un nuevo query para conseguir el ultimo ID registrado, para ello usamos el siguiente código:

```
@Query("SELECT max(uid) FROM Usuario")
fun getLastUserID() : LiveData<Long>
```

Para poder utilizar el query tenemos que actualizar también el repositorio y el ViewModel del usuario para bridar acceso de alto nivel. Los archivos quedan:
UsuarioRepositoy.kt:

```
fun getLastInsertedUserID() : LiveData<Long>{
    return usuarioDao.getLastUserID()
}
```

UsuarioViewModel.kt:

```
fun getLastUserID() : LiveData<Long>{
    return repository.getLastInsertedUserID()
}
```

Ahora podemos utilizar el código de estado después de registrar un usuario para invocar al método getLasUserID() para recuperar el ID y guardarlos en las preferencias como el usuario activo actual:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
    if(requestCode == CodigosDeSolicitud.REGISTRAR_USUARIO && resultCode
== Activity.RESULT_OK){
        usuarioViewModel.getLastUserID().observe(this, Observer {
            val sharedPref =
```

```
PreferenceManager.getDefaultSharedPreferences(this)
            with(sharedPref.edit()){
                putInt("actualUserID",it!!.toInt())
                apply()
            }
            finish()
        })
        Toast.makeText(this,
getString(R.string.usuario_creado_correctamente),
Toast.LENGTH_SHORT).show()
```

Creamos un método que acceda a la preferencia y retorne su valor de forma simple, esto lo logramos con el código:

```
fun getCurrentUserID() : Int{
    val sharedPref = PreferenceManager.getDefaultSharedPreferences(this)
    return sharedPref.getInt("actualUserID", -1)

}
```

Actualizamos el código que pobla las listas para utilizar el ID del usuario actual:

```
val sharedPref = PreferenceManager.getDefaultSharedPreferences(context)
val usuarioID = sharedPref.getInt("actualUserID",-1)


tomaViewModel =
ViewModelProviders.of(this).get(TomaViewModel::class.java)
tratamientoViewModel =
ViewModelProviders.of(this).get(TratamientoViewModel::class.java)
val adapter = HorarioAdapter(context, tomaViewModel,tratamientoViewModel)
tomaViewModel.getTomasDelDiaUsusuario(usuarioID).observe(this, Observer {
    adapter.submitList(it)
})
```

Podemos ver que poblar las listas usando el ID del ultimo usuario registrado es muy sencillo gracias al ViewModel pues desde el principio los diseñamos pensando en poder poblar la listas usándolo, el proceso de buena arquitectura nos evitó trabajo adicional o redundante.

Para el proceso de cambiar de usuario activo es bastante simple, al tocar un elemento de la lista de usuarios este se convierte en el usuario activo actualizando el valor de la preferencia, en caso de que ese perfil cuente con contraseña está será solicitada, sí es ingresada correctamente la contraseña la preferencia es actualizada, en caso contrario un mensaje de advertencia es mostrado y el usuario activo no cambia.

El código es el siguiente:

```
//Especificamos el escucha de eventos para definir el usuario activo de
la aplicacion
adapter.setOnClickListener( View.OnClickListener {
    val usuarioSeleccionado =
adapter.getUsuarioAt(RecViewUsuarios.getChildAdapterPosition(it))
```

```
    if(!usuarioSeleccionado.password.isNullOrEmpty() &&
usuarioSeleccionado.uid != getCurrentUserID()){
        val builder = AlertDialog.Builder(this)
        builder.setTitle(getString(R.string.ingresar_contrasena))
        val inflater = layoutInflater
        val dialogView = inflater.inflate(R.layout.dialog_input_password,
null)
        builder.setView(dialogView)

        val passwordEditText : EditText =
dialogView.findViewById(R.id.userPasswordET)
        builder.setPositiveButton(getString(R.string.ingresar)) { dialog,
id  ->
if(usuarioSeleccionado.password.equals(passwordEditText.text.toString()))
{
                val sharedPref =
PreferenceManager.getDefaultSharedPreferences(this@ListarUsuariosActivity
)
                with(sharedPref.edit()){
                    putInt("actualUserID",usuarioSeleccionado.uid)
                    apply()
                }
                finish()
            }else{

Snackbar.make(frameLayoutListaUsuarios,getString(R.string.contrasena_inco
rrecta), Snackbar.LENGTH_SHORT).show()
            }
        }
        builder.setNegativeButton(getString(R.string.cancelar)) { dialog,
id ->         }

        val alertDialog = builder.create()
        alertDialog.show()
    }else{
        val sharedPref =
PreferenceManager.getDefaultSharedPreferences(this@ListarUsuariosActivity
)
        with(sharedPref.edit()){
            putInt("actualUserID",usuarioSeleccionado.uid)
            apply()
        }
        finish()
    }
}
)
```

Resultados:

La aplicación cuenta con la capacidad de registrar múltiples usuarios y mostrar sus elementos de forma individual. También se implementó la posibilidad de usar una contraseña para prevenir registros incorrectos o mal intencionados.

4.3.2 Recuperación de la contraseña en caso de extravió

Descripción/Requisitos:

La aplicación ya permite almacenar los datos de distintos usuarios, así como también permite utilizar la contraseña en aquellos perfiles que cuenten con ella. Pero hay un problema y es que existe la posibilidad de un usuario olvide su contraseña, debemos implementar una manera simple para que un usuario recupere su contraseña.

Ejecución:

Durante el proceso de diseño y arquitectura definimos que en caso de que el usuario actual ingresará una contraseña para ser usada en su perfil sea necesario ingresar también un correo de recuperación, haremos uso de la información contenida en este campo para poder proporcionar un mecanismo de recuperación.

En el dialogo utilizado para ingresar la contraseña añadimos un tercer botón, el cual contiene el texto de "Recuperar contraseña", al tocarlo invocará al método recoverPassword que tiene como argumento la contraseña del usuario actual la cual es recuperada utilizando el ViewModel:

```
builder.setNeutralButton(getString(R.string.recuperar_contrasena)) {
dialog, id ->

    if(ContextCompat.checkSelfPermission(this,
Manifest.permission.WRITE_EXTERNAL_STORAGE) !=
PackageManager.PERMISSION_GRANTED){
        ActivityCompat.requestPermissions(this,
                arrayOf(Manifest.permission.WRITE_EXTERNAL_STORAGE),

CodigosDeSolicitud.SOLICITAR_PERMISO_ALMACENAMIENTO_EXTERNO)
        userPassWordHelper = usuarioSeleccionado.password     }else{
        recoverPassWord(usuarioSeleccionado.password)
        Log.e("Contrasena", usuarioSeleccionado.password)
    }

}
```

El método recoverPassWord() queda de la siguiente manera:

```
fun recoverPassWord(passWordUsuario: String?){
    val sendIntent = Intent(Intent.ACTION_SEND)
    sendIntent.type = "message/rfc822"
sendIntent.putExtra(Intent.EXTRA_SUBJECT,
getString(R.string.MMR_Recuperar_Contrasena))
    sendIntent.putExtra(Intent.EXTRA_EMAIL, arrayOf(emailRecuperacion))
    sendIntent.putExtra(Intent.EXTRA_TEXT,
getString(R.string.su_contrasena_archivo_adjunto))


    val file = createPasswordFile()
    writePasswordToFile(file, passWordUsuario!!)
    Log.e("Contrasena", userPassWordHelper)

    val fileURI : Uri = FileProvider.getUriForFile(this,
            "com.kps.spart.android.fileprovider",
```

```
            file             )

        sendIntent.putExtra(Intent.EXTRA_STREAM, fileURI)
        sendIntent.setType(".txt -> text/plain")

startActivityForResult(Intent.createChooser(sendIntent,getString(R.string
.enviar_email_recuperacion)), CodigosDeSolicitud.RECUPERAR_CONTRASENA)
}
```

El proceso consiste en crear un intent con la propiedad ACTION_SEND esto le dice al sistema que el intent deberá permitir al usuario elegir entre todas las aplicaciones que permitan él envió de archivos. Especificamos el tipo MIME “message/rfc822” para decirle a Android que el mensaje enviado será basado en texto.

Al especificar las propiedades Intent.EXTRA_SUBJECT y Intent.EXTRA_TEXT le decimos al sistema que muestre las aplicaciones que soporten estos atributos, cómo la idea es que la contraseña sea enviada por correo electrónico cómo un archivo adjunto, entonces estos atributos son usados cómo asunto y cuerpo del email, pero tenemos que crear el archivo que contiene la contraseña. Todo dispositivo con Android y que esté activado en los servidores de Google deben contar con al menos una cuenta de email valida activa, así que se usará de forma predeterminada esta, aunque es posible que el usuario elija con cual cuenta de correo enviar la contraseña sí es que así lo desea.

Creamos otro método auxiliar que crear un archivo temporal al cual le adjunta la contraseña, posteriormente se crear la URI a este archivo y es añadido al intent. Finalmente se inicializa el intent con el método startActivityForResult.

Una vez que la contraseña ha sido enviada debemos eliminar el archivo temporal para evitar la acumulación de archivos basura, los métodos auxiliares quedan así:

```
private fun createPasswordFile() : File{
    val storageDir : File =
getExternalFilesDir(Environment.DIRECTORY_DCIM)
    return File.createTempFile(
            "MMRPASSWORD",
            ".txt",
            storageDir     ).apply {
        userPassWordHelper = absolutePath     }
}

@Throws(IOException::class)
private fun deletePasswordFile(){
    val passwordFile = File(userPassWordHelper)
    val passwordUri: Uri = FileProvider.getUriForFile(
            this,
            "com.kps.spart.android.fileprovider",
            passwordFile     )

    this.contentResolver.delete(passwordUri,null,null)
}
```

La eliminación del archivo temporal es realizada con la siguiente línea de código dentro onActivityResult():

```
if (requestCode == CodigosDeSolicitud.RECUPERAR_CONTRASENA && resultCode
== Activity.RESULT_CANCELED){
  //  deletePasswordFile()
}
```

Resultados:

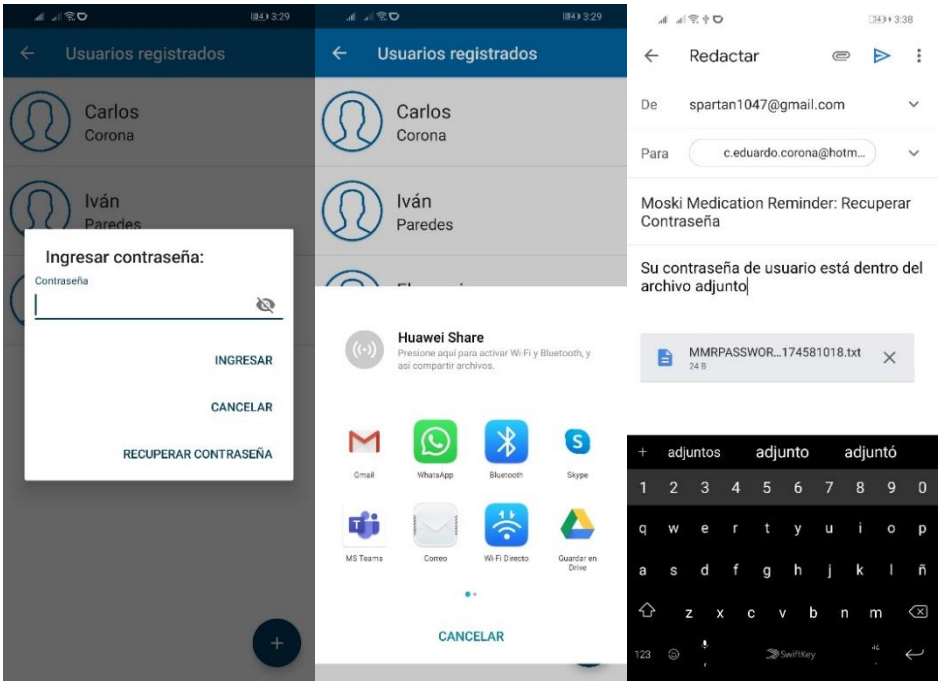

Figura 60 Pasos para recuperar la contraseña de un perfil de usuario.

4.3.3 Creación de métodos de búsqueda

Descripción/Requisitos:

A medida que se registren más entidades dentro de la base de datos la navegabilidad de la aplicación será mucho más complicada, pues será necesario realizar más movimientos de scroll dentro de las listas, además en caso de que existan elementos con nombres similares será necesario que el usuario ponga más atención a la lista con la intención de buscar el elemento de su interés. Se tienen que implementar un mecanismo de búsqueda en cada una de las secciones de elementos: tratamientos, medicamentos, médicos, establecimientos y citas.

Ejecución:

Google proporciona una opción para realizar búsquedas en adaptadores o cualquier otra colección de datos vinculado a una actividad, para ello actualizamos los menús de cada sección de la siguiente manera:

```
<?xml version="1.0" encoding="utf-8"?> <menu
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto">
```

```
    <item          android:id="@+id/itemADD"
android:icon="@drawable/ic_add_white_24dp"
android:title="@string/anadir"          app:showAsAction="always"
/>

    <item          android:id="@+id/itemSearch"
android:icon="@drawable/ic_search_white_24dp"
android:title="@string/buscar"          app:showAsAction="always"
app:actionViewClass="android.support.v7.widget.SearchView"          />
</menu>
```

El ítem incluye la propiedad SeachView que habilita un formulario en la topbar, en la cual el usuario puede ingresar texto, se recupera el texto ingresado y se realiza la búsqueda desde el code behind de la actividad o fragmento que mantiene la lista donde deseamos buscar.

El código para realizar la búsqueda queda de la siguiente manera:

```
override fun onQueryTextSubmit(query: String?): Boolean {
    return false
}

override fun onQueryTextChange(newText: String): Boolean {
    Log.d("Medicamento", newText)
    val userInput = newText?.toLowerCase()
    val newList = arrayListOf<Medicamento>()


    for(medicamento: Medicamento in medicamentos!!){
        if(medicamento.nombreMedicamento?.toLowerCase()?.contains(userInput)!!){
            newList.add(medicamento)
        }
    }

    try {
        adapter.updateList(newList)
    }catch (e: Exception){
        Log.e("Error:", e.localizedMessage )
    }
    return true
}
```

El método onQueryTextSubmit() es invocado cuando el usuario envía el texto manualmente utilizando un botón, en nuestro caso retornamos un valor false para indicar a Android que no utilizaremos este método, en caso contrario tendríamos que pedirle al usuario que haga clic en algún elemento de la interfaz gráfica para actualizar los resultados.

El método onQueryTextChange() es invocado cada vez que el texto en el formulario cambia, podemos utilizar este método para poder implementar el algoritmo de búsqueda. En nuestro caso utilizamos la cadena ingresada del usuario para ir construyendo o descartando listas donde cada elemento tenga contenga los caracteres que hay se ingresan en el formulario de búsqueda.

Resultados:

Ahora es posible realizar búsquedas en las listas, al realizar el algoritmo de búsqueda sobre un adaptador que ya ha sido poblado no hay necesidad de realizar consultas adicionales sobre la base de datos, por lo que no hay caídas en performance. Además cómo las listas ya son únicas para cada usuario no existe el riesgo de que la búsqueda arroje un resultado que corresponda a otro perfil distinto al del usuario activo.

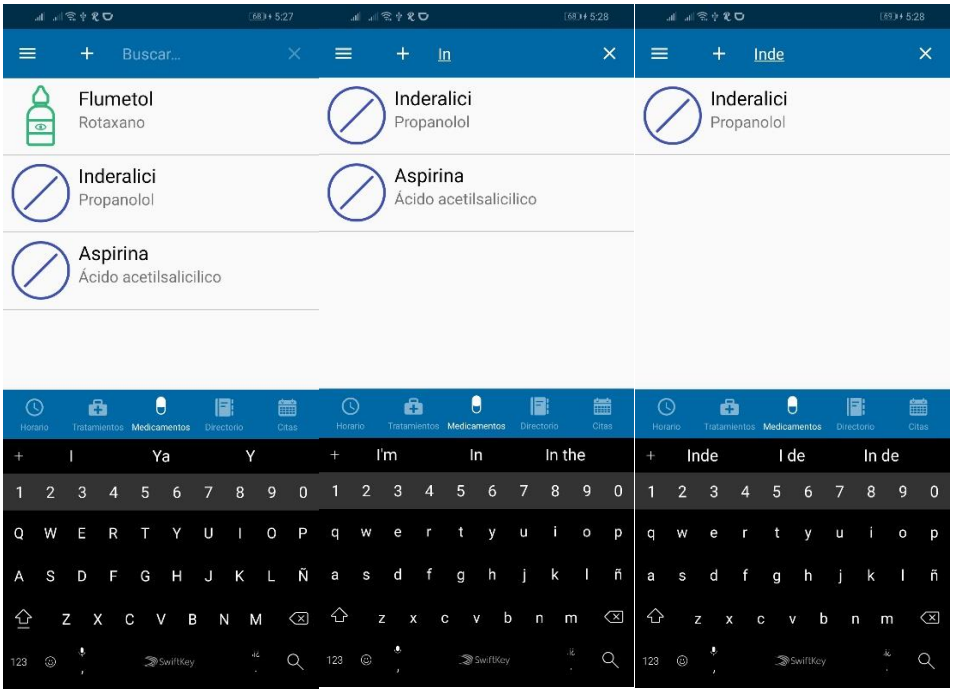

Figura 61 Pasos para realizar una búsqueda en la lista de medicamentos.

4.3.4 Soporte para alarmas

Descripción/Requisitos:

Una vez que se han registrado tratamientos y sus tomas es necesario permitir que la aplicación dispare alarmas sonoras a la hora de una toma.

Ejecución:

La alarmas en Android permiten ejecutar tareas programas por tiempo fuera del ciclo de vida de la aplicación. Por ejemplo, podríamos establecer una alarma que cuando se cumple se inicializa una operación de ejecución larga cómo por ejemplo inicializar un servicio una vez al día para descargar datos desde un servidor.

Las alarmas cuentan con las siguientes características:

- Permiten disparar intents en tiempos específicos o intervalos definidos.
- Pueden ser utilizadas en conjunto con Broadcast receivers para inicializar un servicio y realizar otras operaciones.

- Operan fuera de la aplicación, así que pueden ser usadas para disparar eventos incluso si la aplicación no está corriendo o sí el dispositivo entra en modo de suspensión.
- Permiten reducir el consumo de recursos por parte de la aplicación, pues estos solo son solicitados cuando la alarma es disparada, luego el recolector de basura libera la memoria utilizada.

Creamos una clase dedicada, la cual utilizará el contexto de la aplicación para establecer las alarmas, las alarmas pueden ser usadas tanto para las alarmas sonoras cómo para las notificaciones, necesitamos especificar la hora del día, el minuto cómo elementos básicos para la alarma, pero usamos cadenas para mostrar datos adicionales, la clase queda de la siguiente manera:

```
class AlarmHelper (val context: Context) {

    lateinit var alarmItent: PendingIntent     var alarmMgr:
AlarmManager? = null


    fun createAlarmForNotifications(hourOfDay: Int, minute: Int,
tratamiento: String?, medicamento: String?, tomaID: Int?, recordatorio:
Int?){
        alarmMgr = context.getSystemService(Context.ALARM_SERVICE) as
AlarmManager

        alarmItent = Intent(context, AlarmReceiver::class.java).let{
intent ->             intent.putExtra("Tratamiento", tratamiento)
            intent.putExtra("Medicamento", medicamento)
            intent.putExtra("IDToma", tomaID)
            intent.putExtra("Recordatorio", recordatorio)
            PendingIntent.getBroadcast(context,tomaID!!,intent,
PendingIntent.FLAG_ONE_SHOT)

        }

        val calendar: Calendar = Calendar.getInstance().apply {
            timeInMillis = System.currentTimeMillis()
            set(Calendar.HOUR_OF_DAY, hourOfDay)
            set(Calendar.MINUTE, minute)
        }

        alarmMgr?.set(
                AlarmManager.RTC_WAKEUP,
                calendar.timeInMillis,
                alarmItent         )

    }

    fun createAlarmForAppointments(hourOfDay: Int, minute: Int, titulo:
String?, doctor: String?,especialidad : String?, citaID: Int?,
recordatorio: Int?){
        alarmMgr = context.getSystemService(Context.ALARM_SERVICE) as
AlarmManager

        alarmItent = Intent(context,
AppointmentAlarmReceiver::class.java).let{ intent ->
intent.putExtra("Titulo", titulo)
```

```
            intent.putExtra("Doctor", doctor)
            intent.putExtra("Especialidad", especialidad)
            intent.putExtra("Recordatorio", recordatorio)
            intent.putExtra("IDCita", citaID)
            PendingIntent.getBroadcast(context,citaID!!,intent,
PendingIntent.FLAG_ONE_SHOT)

        }

        val calendar: Calendar = Calendar.getInstance().apply {
            timeInMillis = System.currentTimeMillis()
            set(Calendar.HOUR_OF_DAY, hourOfDay)
            set(Calendar.MINUTE, minute)
        }

        alarmMgr?.set(
                AlarmManager.RTC_WAKEUP,
                calendar.timeInMillis,
                alarmItent         )

    }


    fun cancelAlarm(){
        alarmMgr?.cancel(alarmItent)
    }

    fun enableReceiver(){
        val receiver = ComponentName(context,
TreatmentBroadcastReceiver::class.java)

        context.packageManager.setComponentEnabledSetting(
                receiver,
                PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
                PackageManager.DONT_KILL_APP

        )
    }

    fun disableReceiver(){
        val receiver = ComponentName(context,
TreatmentBroadcastReceiver::class.java)

        context.packageManager.setComponentEnabledSetting(
                receiver,
                PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
                PackageManager.DONT_KILL_APP

        )
    }

}
```

Los métodos createAlarmForNotifications() y createAlarmForAppointments utilizan una instancia de AlarmManager, el cual permite programar alarmas en el sistema utilizando un calendario para especificar el punto del día.

Para que una alarma sea "escuchada" por la aplicación debemos de registrar un broadcast receiver que esté atento a los eventos del sistema. Para esto añadimos las siguientes líneas de código al manifiesto:

```
<receiver     android:name="alarms.AlarmReceiver"
android:exported="false" />
```

Con esto indicamos que nuestras alarmas deben ser atendidas dentro de la clase AlarmReceiver.kt

```
class AlarmReceiver : BroadcastReceiver(){
    override fun onReceive(context: Context?, intent: Intent?) {

        val tratamiento = intent?.getStringExtra("Tratamiento")
        val medicamento = intent?.getStringExtra("Medicamento")
        val idToma = intent?.getIntExtra("IDToma", -1)
        val recordatorio = intent?.getIntExtra("Recordatorio", -1)

        val notificationManager = NotificationsManager(context!!)


        if(recordatorio == TipoRecordatorio.ALARMA){

notificationManager.sendNotificationForAlarm(tratamiento!!,medicamento!!,
idToma!!)
            val mediaPlayer = MediaPlayer()
            val alarmSound =
RingtoneManager.getDefaultUri(RingtoneManager.TYPE_ALARM)
            mediaPlayer.setDataSource(context!!,alarmSound)
            mediaPlayer.prepare()


            val handler = Handler()
            handler.postDelayed(Runnable {
                mediaPlayer.start()
            }, 5000)

            val timer = object : CountDownTimer(10000,5000){
                override fun onTick(millisUntilFinished: Long) {

                }
                override fun onFinish() {
                    if(mediaPlayer.isPlaying){
                        mediaPlayer.stop()
                        mediaPlayer.release()
                    }
                }
            }
            timer.start()

        }else if(recordatorio == TipoRecordatorio.NOTIFICACION){
```

```
notificationManager.sendNotification(tratamiento!!,medicamento!!,
idToma!!)
        }

        Log.d("Alarma","Se disparo: " + tratamiento + " | " + medicamento
+ " | " + idToma)

    }
}
```

La clase permite crear una alarma sonora utilizando los sonidos predeterminados del sistema Android, esta alarma sonará durante 10 segundos, encendiendo la pantalla del dispositivo incluso sí está bloqueado.

Por último, debemos de crear un método que nos permita iterar sobre todas las tomas validas, extraer sus datos y utilizarlos para crear las alarmas. Esto es realizado desde el ViewModel de las tomas primero creando una clase interna que utiliza una tarea asíncrona para programar las tomas, luego creamos un método que permita inicializar dicha tarea, quedando así:

```
private inner class scheduleAlarmsForShotsAsyncTask constructor(private
val tomaDao: TomaDao) : AsyncTask<Int, Void, Void>(){
    override fun doInBackground(vararg params: Int?): Void? {
        val sdf = SimpleDateFormat("h:mm a")
        var shotDate : Date          val cal = Calendar.getInstance()

        val currentLocalTime = LocalTime.now()

        val sharedPrefs =
PreferenceManager.getDefaultSharedPreferences(application)
        val anticipation = sharedPrefs.getString("appointmentReminder",
"NA")

        val alarmAnticipation =
application.resources?.getStringArray(R.array.anticipacion_toma)



        for(shot in tomaDao.getTomasProgramadasWithoutLiveData()){
            shotDate = sdf.parse(shot.horaToma)
            cal.time = shotDate

            when(alarmAnticipation?.indexOf(anticipation)){
                1 ->{ cal.add(Calendar.MINUTE, 1)}
                2 ->{ cal.add(Calendar.MINUTE, 5)}
                3 ->{ cal.add(Calendar.MINUTE, 10)}
            }

            Log.d("Tomas", shot.toString() + " $ " +
cal.get(Calendar.HOUR_OF_DAY) + " $ " + cal.get(Calendar.MINUTE) + " $ "
+ shot.id)
            val shotTime = LocalTime.of(cal.get(Calendar.HOUR_OF_DAY),
cal.get(Calendar.MINUTE))
```

```
            if(shotTime.isAfter(currentLocalTime)) {

alarmHelper.createAlarmForNotifications(cal.get(Calendar.HOUR_OF_DAY),
cal.get(Calendar.MINUTE), shot.tituloTratamiento, shot.medicamento,
shot.id, shot.tipoRecordatorio)

            }

        }


        return null     }
}
```

```
fun scheduleAlarmsForShots() {
    scheduleAlarmsForShotsAsyncTask(tomaDao).execute()
}
```

Con el método arriba mostrado ahora posible hacer sonar una alarma sonora cuando llega el momento de administrar una toma o cuando llega la hora de asistir a una cita médica. Simplemente tenemos que llamarlo en onActivityResult() después de registrar un tratamiento o una cita médica.

Resultados:
Ahora la aplicación lanza una alarma sonora al usuario de forma automática cada vez que se registrar un tratamiento o una cita médica sin necesidad de mantener la aplicación abierta.

4.3.5 Soporte para notificaciones

Descripción/Requisitos:
La aplicación de mostrar notificaciones que permitan al usuario saber cuándo una toma debe ser tomada o se debe asistir a una cita médica. La ventaja de las notificaciones respecto a las alarmas sonoras es que al ser visuales es menos probable que el usuario las ignore o las olvide, pues deben ser deben ser descartadas manualmente o hacer clic ellas para abrir la aplicación y descartar la notificación.

Ejecución:
La implementación de notificaciones es un proceso bastante sencillo en Android, pues tenemos que hacer uso de un objeto que implemente los métodos básicos de la clase NotificationManager, esta clase permite registrar la aplicación el sistema para enviar notificaciones, luego podemos construir notificaciones utilizando una instancia de dicha clase y pasando los parámetros de icono, titulo y subtitulo además es posible adjuntar un intent a cada notificación lanzada para realizar una acción, en nuestro caso abrir la aplicación, el código queda de la siguiente manera:

```
class NotificationsManager(val context: Context) {
    lateinit var mNotifyManager : NotificationManager     lateinit var
mRingtone : Ringtone
```

```
    fun createNotificationChannel(){
        mNotifyManager = context.getSystemService(NOTIFICATION_SERVICE)
as NotificationManager

        if(android.os.Build.VERSION.SDK_INT >=
android.os.Build.VERSION_CODES.O){
            val notificationChannel =
NotificationChannel(CANAL_PRIMARIO_ID,"Moski medication reminder",
NotificationManager.IMPORTANCE_HIGH)
            notificationChannel.enableLights(true)
            notificationChannel.lightColor = Color.RED
notificationChannel.enableVibration(true)
            notificationChannel.description = "Notification from MMR to
shots and medical appointments"
mNotifyManager.createNotificationChannel(notificationChannel)
        }
    }

    fun getNotificationBuilder(title: String, content: String) :
NotificationCompat.Builder{
        val notificationIntent = Intent(context,
MainActivity::class.java)
        val notificationPendingIntent =
PendingIntent.getActivity(context, NOTIFICACION_ID, notificationIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
        val notifyBuilder = NotificationCompat.Builder(context,
CANAL_PRIMARIO_ID)
                .setContentTitle(title)
                .setContentText(content)
                .setSmallIcon(R.drawable.ic_notification_capsule)
                .setContentIntent(notificationPendingIntent)
                .setAutoCancel(true)
                .setPriority(NotificationCompat.PRIORITY_HIGH)
                .setDefaults(NotificationCompat.DEFAULT_ALL)

        return notifyBuilder     }

    fun sendNotification(title: String, content: String, tomaID: Int){
        val takeShotIntent = Intent(context,
TreatmentBroadcastReceiver::class.java).apply {
            putExtra("TomaID", tomaID)
            putExtra("AcctionToma", EstatusToma.TOMADA)

        }
        val takeShotPendingIntent = PendingIntent.getBroadcast(context,
AccionNotificacion.TOMAR, takeShotIntent, PendingIntent.FLAG_ONE_SHOT)

        val skipShotIntent = Intent(context,
TreatmentBroadcastReceiver::class.java).apply {
            putExtra("TomaID", tomaID)
            putExtra("AcctionToma", EstatusToma.PASADA)

        }
        val skipShotPendingIntent = PendingIntent.getBroadcast(context,
AccionNotificacion.SALTAR, skipShotIntent, PendingIntent.FLAG_ONE_SHOT)
```

```
        val postPoneShotIntent = Intent(context,
TreatmentBroadcastReceiver::class.java).apply {
            putExtra("TomaID", tomaID)
            putExtra("AcctionToma", EstatusToma.POSPUESTA)
        }
        val postPoneShotPendingIntent =
PendingIntent.getBroadcast(context, AccionNotificacion.POSPONER,
postPoneShotIntent, PendingIntent.FLAG_ONE_SHOT)


        val notifyBuilder = getNotificationBuilder(title,content)
        notifyBuilder.addAction(R.drawable.ic_capsula,
context.getString(R.string.tomar), takeShotPendingIntent)

notifyBuilder.addAction(R.drawable.ic_capsula,context.getString(R.string.
saltar), skipShotPendingIntent)

notifyBuilder.addAction(R.drawable.ic_capsula,context.getString(R.string.
posponer), postPoneShotPendingIntent)

        mNotifyManager = context.getSystemService(NOTIFICATION_SERVICE)
as NotificationManager          mNotifyManager.notify(NOTIFICACION_ID,
notifyBuilder.build())
    }
    fun sendNotificationForAppointment(title: String, doctor: String,
especialidad: String, idCita: Int){
        val notificationIntent = Intent(context,
DetallesCitaMedicaActivity::class.java)
        notificationIntent.putExtra("CITA_ID", idCita)
        val notificationPendingIntent =
PendingIntent.getActivity(context, NOTIFICACION_ID, notificationIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
        val notifyBuilder = NotificationCompat.Builder(context,
CANAL_PRIMARIO_ID)
                .setContentTitle(title)
                .setContentText(especialidad + " " + doctor)
                .setSmallIcon(R.drawable.ic_notification_capsule)
                .setContentIntent(notificationPendingIntent)
                .setAutoCancel(true)
                .setPriority(NotificationCompat.PRIORITY_HIGH)
                .setDefaults(NotificationCompat.DEFAULT_ALL)

        mNotifyManager = context.getSystemService(NOTIFICATION_SERVICE)
as NotificationManager          mNotifyManager.notify(NOTIFICACION_ID,
notifyBuilder.build())
    }

    fun sendNotificationForAlarm(title: String, content: String, tomaID:
Int){
        val notifyBuilder = getNotificationBuilder(title,content)

        mNotifyManager = context.getSystemService(NOTIFICATION_SERVICE)
as NotificationManager          mNotifyManager.notify(NOTIFICACION_ID,
notifyBuilder.build())
    }
```

Creamos métodos para los 3 casos en los que lanzaremos una notificación al usuario: Para un tratamiento que lanza notificaciones únicamente, cuando un tratamiento lanza notificaciones con una alarma sonora y cuando una cita médica lanza una notificación.

Debemos recordar que desde la clase creada anteriormente AlarmReceiver filtrábamos el tipo de recordatorio de la entidad cuando su respectiva alarma era disparada así que no es necesario implementar cambios en alguna otra parte del código para mostrar estas nuevas notificaciones, así que registrar un tratamiento o una cita médica mostrará la notificación correctamente.

Resultados:

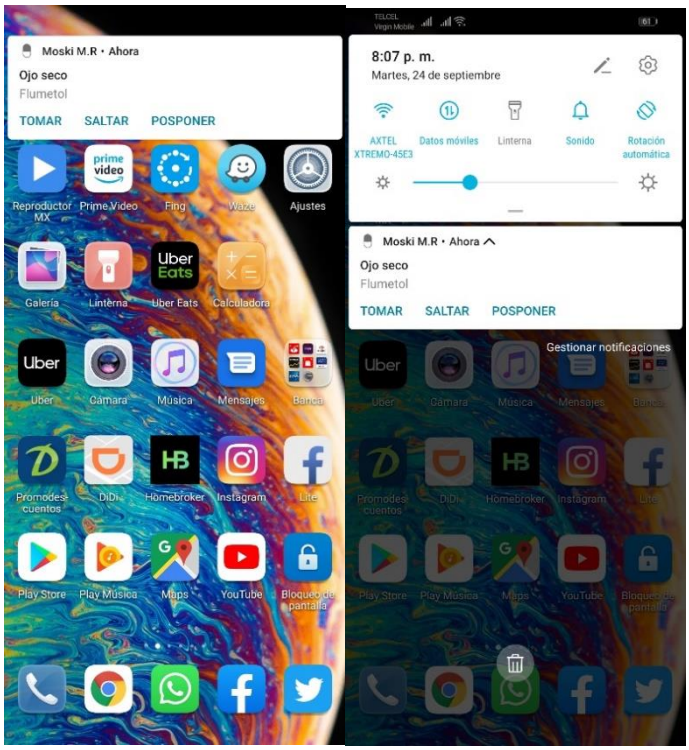

Figura 62 Notificaciones de la aplicación siendo mostrada en el sistema.

4.3.6 Reinicio de los días

Descripción/Requisitos:

En este punto la aplicación ya muestra notificaciones y alarmas al usuario cuando registra una nueva entidad en la base de datos, el problema es que las notificaciones y alarmas solo funcionan durante el día en que son creadas. Se debe implementar un sistema que permita volver a establecer las alarmas y notificaciones que sean válidas.

Ejecución:

Haremos uso de dos de las herramientas más potentes de Android: los servicios y los broadcast receivers. Los servicios son componentes que permiten realizar tareas de larga duración en segundo plano y sin ofrecer interfaz gráfica al usuario, mientras que los broadcast receivers son componentes que permiten escuchar distintos tipos de eventos ya sean de la propia aplicación que los declara, de aplicaciones externas o del sistema.

Recordemos que ya contamos con la lógica para agendar las alarmas que generan las notificaciones y alarmas sonoras que son válidas, pero estas son programadas manualmente. Primero debemos definir un servicio, este nos permitirá programar las alarmas sin tener que utilizar una interfaz gráfica:

Primero registramos el servicio en el manifiesto, de la siguiente manera:

```
<service     android:name="alarms.TomasSchedulerService"
android:exported="false"/>
```

Con esto permitimos que Android pueda acceder a nuestra clase así nosotros no hayamos inicializado la aplicación, pero el atributo Android:exported =”false” especifica que el servicio solo podrá ser lanzado por nuestra propia aplicación.

Los servicios son inicializados por intents, extendiendo de la clase IntentService será el sistema el encargado de determinar cuando el servicio haya realizado su trabajo y tenga que liberar memoria.

El código del servicio para agendar las tomas queda de la siguiente manera:

```
class TomasSchedulerService : IntentService("TomasSchedulerService") {
    override fun onHandleIntent(intent: Intent?) {
        val tomaRepository = TomaRepository(application)

        tomaRepository.scheduleAlarmsForShots()
    }
}
```

El servicio utiliza una instancia de los repositorios necesarios e inicializa la tarea asíncrona para agendar las alarmas de las notificaciones.

Ahora finalmente toca atacar el problema de invocar el servicio automáticamente, para ello utilizaremos un Broadcast receiver implícito, es decir detectaremos los eventos del sistema que sean de nuestro interés y ejecutaremos un método a modo de respuesta, en este caso para fijar las alarmas.

En el manifiesto registramos la clase SetUpAlarmOnReboot la cual extiende de la clase BroadcastReceiver.

```
<receiver android:name="alarms.SetUpAlarmsOnReboot"
android:exported="true" >     <intent-filter>         <action
android:name="android.intent.action.BOOT_COMPLETED" />         <action
android:name="android.intent.action.EVENT_REMINDER" />     </intent-
filter> </receiver>
```

Declaramos que estamos esperando dos eventos que pueden disparar nuestra clase que reinicia las alarmas: BOOT_COMPLETED, disparado cuando el teléfono se ha encendido y EVENT_REMINDER, disparado cuando el celular cambia de fecha. ¿Por qué no usar solo EVENT_REMINDER?, pues porque es probable que el usuario apague y encienda el dispositivo varias veces al día ya sea

voluntariamente o porque se le ha terminado la batería y usando únicamente EVENT_REMINDER las alarmas funcionarias únicamente durante el primer encendido del dispositivo.

La clase que atrapa los eventos queda:

```
class SetUpAlarmsOnReboot : BroadcastReceiver(){
    override fun onReceive(context: Context?, intent: Intent?) {
        val setupShoots = Intent(context,
TomasSchedulerService::class.java)
        context?.startService(setupShoots)

        val setUpAppointments = Intent(context,
CitasSchedulerService::class.java)
        context?.startService(setUpAppointments)

    }
}
```

Cuando una alarma es programada a la misma hora que otra existente por intent que contiene la misma firma (es decir los mismo datos adjuntos que uno ya existente) la alarma es actualizada por tanto no tenemos el riesgo de que se disparen múltiples notificaciones al mismo tiempo para la misma toma o cita médica.

Resultados:

Ahora la aplicación puede seguir mostrando alarmas sonoras y notificaciones tradicionales así el dispositivo donde esté instalada sea reiniciado o haya pasado más de un día. Cómo podemos ver consultar datos para construir funcionalidades más elaboradas utilizando los ViewModels de nuestro interés.

4.3.7 Creación de mecanismos de configuración de la aplicación

Descripción/Requisitos:

La aplicación ya realiza todas las funcionalidades planeadas en diseño de la arquitectura, pero algunos de los valores que podrían variar según las preferencias del usuario se encuentran fijas en las vistas, es decir siempre son las mismas. Debemos proporcionar mecanismos para que la aplicación que permitan establecer valores como el tipo de recordatorio predeterminado y la anticipación de las hora de las tomas.

Ejecución:

Crear una pantalla de configuración para una aplicación Android consiste en bridar una interfaz gráfica a las preferencias compartidas. Para ello se crea una actividad que sostiene un tipo de layout especial asocia una interfaz gráfica a cada una de las preferencias según sea su tipo (switches para los booleanos, edittext para las strings, pickers para los enteros, etc.…).

Comenzamos creando una actividad que contiene el ciclo de vida normal de una pantalla en Android:

```
class SettingsActivity : AppCompatActivity() {

    class SettingsFragment : PreferenceFragment(){
        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            addPreferencesFromResource(R.xml.preferences)
        }
    }

    override fun onCreate(savedInstanceState : Bundle?){
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_settings)

        toolbar.title = getString(R.string.configuracion)
        setSupportActionBar(toolbar)
        val ab = supportActionBar
ab!!.setDisplayHomeAsUpEnabled(true)


        fragmentManager.beginTransaction()
                .replace(R.id.fragmentSettings,SettingsFragment())
                .commit()

    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when(item.itemId){
            android.R.id.home ->{
                onBackPressed()
                return true             }
        }
        return super.onOptionsItemSelected(item)
    }

}
```

La diferencia es que esta actividad debe estar vinculada a un tipo de vista especial que contiene la representación gráfica de las preferencias que deseamos soportar.

Preference.xml:

```
<?xml version="1.0" encoding="utf-8"?> <PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android">
<PreferenceCategory         android:title="Alarmas y notificaciones"
android:key="pref_key_alerts_settings">
        <ListPreference
android:title="@string/recordatorio_predeterminado_tratamientos"
android:key="treatmentReminder"
android:entries="@array/recordatorio_preferido"
android:entryValues="@array/recordatorio_preferido"
android:defaultValue="@string/notificaci_n">         </ListPreference>

        <ListPreference
android:title="@string/recordatorio_predeterminado_citas"
android:key="appointmentReminder"
android:entries="@array/recordatorio_preferido"
android:entryValues="@array/recordatorio_preferido"
```

```
android:defaultValue="@string/notificaci_n">        </ListPreference>

    </PreferenceCategory>

    <PreferenceCategory         android:title="Anticipación
recordatorios"         android:key="pref_reminder_anticipation"
android:summary="Lanzar la con anticipación de ciertos minutos"         >
<ListPreference
android:defaultValue="@string/sin_anticipacion"
android:key="reminderAnticipation1"
android:title="Anticipación tomas"
android:entries="@array/anticipacion_toma"
android:entryValues="@array/anticipacion_toma"            />
</PreferenceCategory>
</PreferenceScreen>
```

Cuando la aplicación es compilada y se detecta una actividad con una lista de preferencias, estas son creadas en tiempo de compilación, pudiendo por tanto ser accedidas desde otras partes del código fuente utilizando el ID definido desde el archivo XML.

Por ejemplo, acceder a la cantidad de tiempo definida por el usuario para sonar la alarma antes de una toma se hace de la siguiente manera:

```
val anticipation = sharedPrefs.getString("appointmentReminder", "NA")
```

Las preferencias que se crean utilizando pantalla de configuración deben ser establecidas desde la vista y/o por el usuario, tratar de establecerlas desde código fuente puede producir un error pues no se actualiza la vista asociada.

Resultados:
Ahora la aplicación cuenta con su propio sistema de configuraciones, las cuales son persistentes y pueden ser utilizadas por el usuario para personalizar el comportamiento de la aplicación.

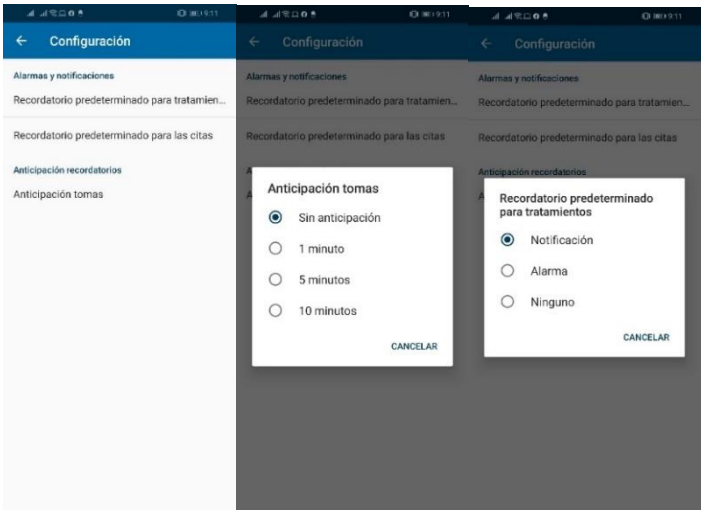

Figura 63 Detalles de las configuraciones de la aplicación.

4.3.6 Implementación de una pantalla de carga

Descripción/Requisitos:

La aplicación ya cuenta con todas las funcionalidades planeadas durante el proceso de arquitectura, pero al inicializar la aplicación el usuario puede experimentar una pantalla blanca que se muestra durante un tiempo variable mientras se cargan los elementos desde la base de datos, es necesario implementar una pantalla de carga o presentación que sea mostrada durante dicho tiempo.

Ejecución:

Debemos crear una actividad que será la que contendrá todo el código fuente que carga los datos de base de datos y cualquier otra operación de carga o verificación. Creando la actividad queda de la siguiente manera:

SplashActivity.kt

```
class SplashActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?){
        super.onCreate(savedInstanceState)

        val sharedPref =
PreferenceManager.getDefaultSharedPreferences(this)
       // val firstRun = !sharedPref.getBoolean("firstrun", true)
        if(!sharedPref.getBoolean("firstrun", true)){
            checkDateForRunService()
            Log.d("EstasReiniciando","Estas iniciando la actualizacion de
tomas ")
        }else{
            Log.d("EstasReiniciando", "no se está reiniciando")
        }

        val channelManager = NotificationsManager(this)
        channelManager.createNotificationChannel()

        startActivity(Intent
(this@SplashActivity,MainActivity::class.java))
        finish()
    }


    fun checkDateForRunService(){
        val sharedPref =
PreferenceManager.getDefaultSharedPreferences(this)

        val fechaInicioUso = sharedPref.getString("firstrundate", "")

        val sdf = SimpleDateFormat.getDateInstance()
        val todayDate = Calendar.getInstance().time
        val savedDate = sdf.parse(fechaInicioUso)

        val auxDate = sdf.parse(sdf.format(todayDate))
        Log.d("ComparaFechas",auxDate.compareTo(savedDate).toString() + "
# " + auxDate.time.toString() + " : " + savedDate.time.toString())

        if(auxDate.compareTo(savedDate) != 0){
```

```
            Log.d("ComparaFechas","Las fechas no son iguales")
            with(sharedPref.edit()){
                putString("firstrundate",
sdf.format(Calendar.getInstance().time))
                apply()
            }

            Log.d("ComparaFechas","Estas invocando el servicio")
            val restartShots =
Intent(this,MMRReiniciarDiaService::class.java)
            this.startService(restartShots)

            val tomaViewModel =
ViewModelProviders.of(this).get(TomaViewModel::class.java)
            tomaViewModel.scheduleShotsNotifications()

            val citasViewModel =
ViewModelProviders.of(this).get(CitaMedicaViewModel::class.java)
            citasViewModel.scheduleCitasAlarms()

            Log.d("ComparaFechas","Estas reiniciando tomas" +
auxDate.compareTo(savedDate) + " @ " + auxDate.time.toString() + " : " +
savedDate.time.toString())
        }else if(auxDate.compareTo(savedDate) == 0){
            Log.d("ComparaFechas","Las fechas son iguales")

        }else{
            Log.d("ComparaFechas","No está entrando " +
auxDate.compareTo(savedDate))
        }

    }
}
```

Las líneas:

```
startActivity(Intent (this@SplashActivity,MainActivity::class.java))
finish()
```

Son las encargadas finalizar esta actividad una vez que se han terminado de cargar todos los datos e inicializa la pantalla principal de la aplicación. Por último, se modificó el manifiesto para decirle al sistema Android que la pantalla de carga será el punto de entrada a la aplicación:

```
<activity     android:name=".SplashActivity"
android:screenOrientation="portrait"
android:theme="@style/SplashTheme">     <intent-filter>         <action
android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
</intent-filter> </activity>
```

Resultados:
Ahora la aplicación muestra una imagen con el logotipo y colores definidos cuando se inicializa, pasando automáticamente a la sección principal de la aplicación una vez que termina el proceso de carga.

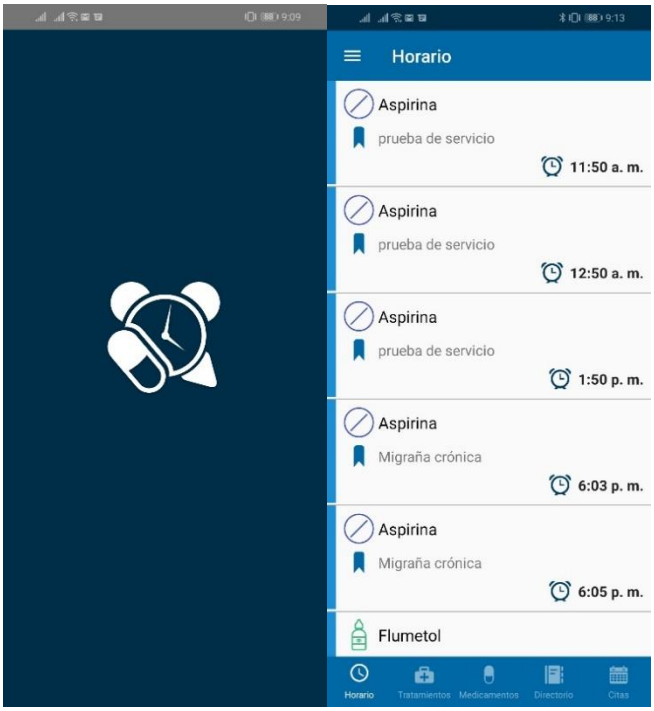

Figura 64 Detalles de la pantalla de carga y pantalla principal de la aplicación.

4.3.6 Soporte para múltiples idiomas

Descripción/Requisitos:
Una de las características más potentes de las aplicaciones Android es la capacidad de ofrecer varios idiomas de una forma simple, sin tener que generar varias versiones de la misma aplicación. Debemos aplicar estos mecanismos para ofrecer soporte para múltiples idiomas siendo español e inglés los idiomas inicialmente soportados.

Ejecución:
Actualmente todas las cadenas de texto que son visibles para el usuario están definidas en cada uno de los archivos de layout de las actividades y fragmentos de la siguiente manera:

```
<TextView     android:id="@+id/tagTelefonoTV"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginBottom="2dp"     android:layout_marginStart="2dp"
android:layout_toEndOf="@id/iconoLocFarmacia"     android:text="Dirección
farmacia"     android:textColor="@color/colorPrimary"
android:textSize="@dimen/subtitulos"     />
```

Se tiene que extraer todas cadenas y crear un diccionario, Android Studio ofrece una forma fácil para crear esto y es posicionarnos sobre cada una de las cadenas que están fijas en las vistas y presionar la combinación de teclas ALT + Enter, esto lanzara el asistente de traducciones en donde podemos especificar una clave y un valor para esa cadena en concreto.

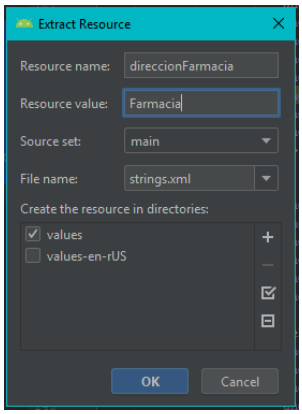

Figura 65 Asistente para la extracción de cadenas.

Una vez que se presiona el botón de OK se crea o actualiza un archivo con cada una de las cadenas que se han convertido y modifica el código donde se hace referencia a dicha cadena con un nuevo identificador basado en la clave que creamos, por ejemplo, la cadena anterior mostrada queda de la siguiente manera:

```
<TextView     android:id="@+id/tagTelefonoTV"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginBottom="2dp"     android:layout_marginStart="2dp"
android:layout_toEndOf="@id/iconoLocFarmacia"
android:text="@string/direccionFarmacia"
android:textColor="@color/colorPrimary"
android:textSize="@dimen/subtitulos"     />
```

Una vez que se han extraído todas las cadenas de las vistas tendremos un nuevo archivo llamado String.xml este contendrá todas las cadenas en un modo clave valor y será utilizado por la aplicación para poblar las vistas con el texto de forma correcta.

Desde el editor de traducciones podemos ver los diccionarios, donde podemos crear archivos para cada idioma y llenar sus respectivas cadenas utilizando la llave definida, quedando de la siguiente manera para el idioma inglés:

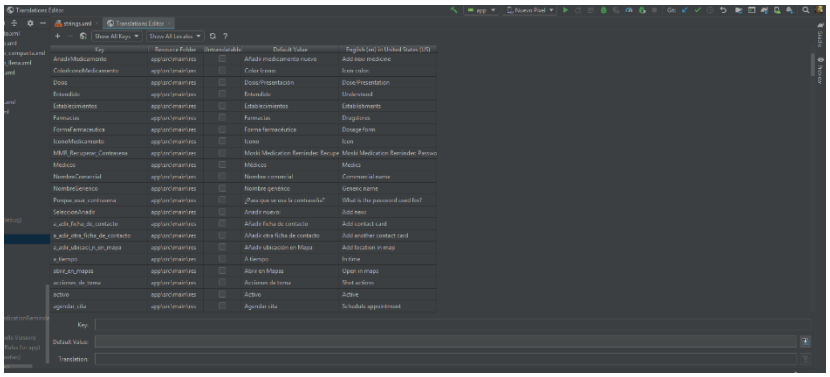

Figura 66 Editor de traducciones de Android Studio.

Una vez que se han llenado todas las cadenas para el idioma ingles podemos compilar de nuevo la aplicación, sí el dispositivo donde corre está en el idioma inglés se mostrarán las cadenas en este idioma, siendo el mismo caso para el idioma español.

Resultados:

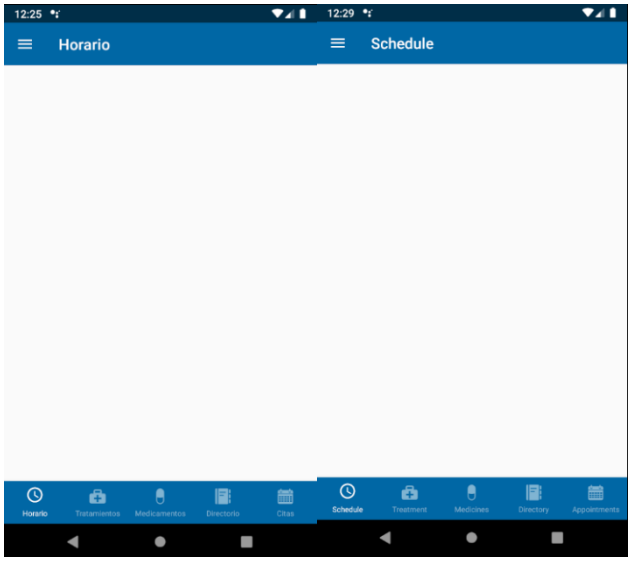

Figura 67 Visualización de la aplicación en 2 idiomas.

Ahora la aplicación puede ser visualizada en español o inglés según sea el idioma especificado en el dispositivo donde corre. Esto demuestra que ofrecer una aplicación en distintos idiomas es un proceso sencillo que no requiere manipulación como tal del modelo, por lo que es muy poco probable que causemos errores en el funcionamiento de la aplicación.

Capítulo 5. Lanzamiento y distribución

En este último capítulo mostraremos el proceso que hay que seguir para la preparación de nuestro código fuente en un archivo .APK que sea apto para la distribución en la Google Play Store, pero además mostraremos el proceso que hay que seguir para llenar la información de la ficha de la aplicación y que esta sea mostrada en al menos dos idiomas para su exposición dentro de la Play Store cuando es visitada en teléfonos activados en países donde se hablan dichos idiomas. Con la publicación de la aplicación viene una nueva responsabilidad y es el seguimiento de las estadísticas de descargas e ingresos para poder conseguir un feedback que nos permita planear futuras actualizaciones supliendo las nuevas necedades y requerimientos detectados.

5.1 Preparando la aplicación para su publicación

En este punto ya contamos un PMV (Producto Mínimo Viable) que cumple con las reglas de negocio establecidas al inicio del proyecto y que se encuentra funcionalmente en nuestra versión actual del código de la rama Master de nuestro repositorio local y remoto, ahora viene el proceso de empaquetado y publicación de la aplicación dentro de la Play Store. Hasta ahora hemos probado la aplicación en nuestro dispositivo físico conectándolo a la PC, abriendo Android Studio y cargando una de las versiones del código fuente haciendo clic en el botón Play.

Cada vez que compilamos e instalamos una aplicación usando Android Studio se crea una llave de depuración o certificado que permite dicha instalación, toda aplicación que se publicará en la Play Store u alguna otra tienda de aplicaciones debe ser compilada y “firmada”, es decir debe indicar de forma digital quien fue el programador que ha realizado la aplicación, solo la persona o equipo de desarrollo que tenga el certificado con la firma puede enviar actualizaciones de ese archivo empaquetado (.APK). Los dispositivos Android no instalarán aplicaciones sí es que no están firmadas, así se trate de instalar aplicaciones manualmente.

El certificado y firma funcionan como medio de autentificación y brinda ventajas de prevención de robo de identidad, imaginen que alguien roba nuestro código fuente y teniendo ya una aplicación publicada con millones de descargas e instalaciones activas, esa persona podría publicar de nuevo nuestra aplicación permitiendo robar datos confidenciales o causar daño a los dispositivos de las personas con malware. El certificado permite que solo quienes crearon la aplicación suba actualizaciones y permite verificar que la aplicación es “única” en la Play Store, pero podemos subir varias “versiones” de la misma aplicación siempre y cuando la persona que sube estas versiones sea la misma que firma la aplicación.

El proceso de firmado es bastante sencillo y no requiere que tengamos una cuenta de desarrollador en la Play Store.

5.2 Firmando la aplicación

Para firmar la aplicación necesitamos un archivo con terminación .jks, este archivo es el que será usado para actualizar nuestra aplicación únicamente por el desarrollador. Para generar una nueva llave debemos ir al menú Build > Generate Signed Bundle/APK, elegimos la opción de APK y hacemos clic en siguiente.

Se desplegará una ventana para firmar las aplicación utilizando una llave existente o dará la opción de crear una llave nueva, hacemos clic en “Create new…”

Aparecerá una ventana que pregunta datos como ruta donde guardar la llave, una contraseña para abrir la llave, un alias, una contraseña para validar la llave, la cantidad de años que la llave es válida y los datos del certificado, que incluyen el nombre del programador, el departamento y la organización a la que pertenece, ciudad, estado y país.

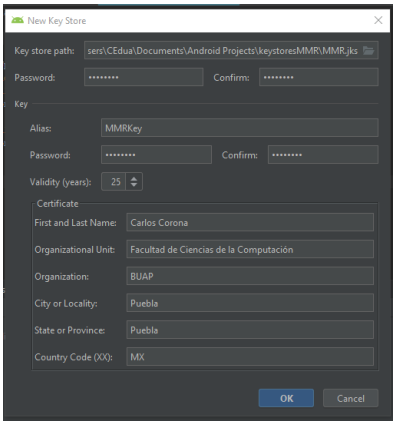

Figura 68 Formulario para creación de una nueva llave para la tienda.

Con la llave creada estamos lista para crear el archivo .APK de la aplicación, lo cual detallaremos en la siguiente sección.

5.2 Generación del archivo .APK de instalación

La aplicación ya cumple con todas los requerimientos funcionales y reglas de negocios que se planearon durante la fase de diseño, pero hasta el momento para poder utilizarla era necesario compilar el código fuente desde Android Studio, donde una vez terminado el proceso la aplicación es lanzada en un dispositivo Android conectado a la PC o en el emulador. Tenemos que generar el archivo .APK (Android Application Package o en español: Paquete de Aplicación Android) que el formato de archivos utilizado por Google para instalar aplicaciones en Android.

En la tarea anterior se firmó el código fuente, así que para generar una .APK que está firmada y lista para publicarse en la tienda se genera haciendo clic en el menú Build > Generate Signed Bundle/APK, elegimos la opción de APK y hacemos clic en siguiente.

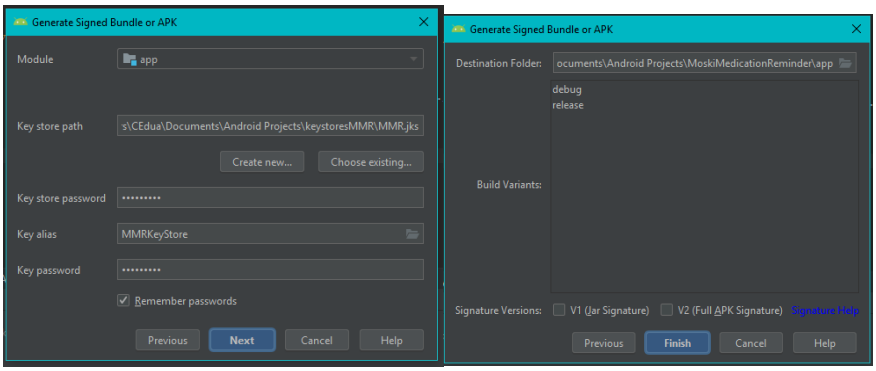

Figura 69 Formulario para la creación de un archivo .APK firmado.

Se nos preguntarán los datos de accesso a la llave, los mismos que fueron definidos cuando se creó. Sí todos los datos son correctos posteriormente se nos preguntará que tipo de compilación

deseamos: debug o release, finalmente hacemos hacemos clic en el boton Finish, Android Studio comenzará a crear el .APK notificandonos cuando esté listo.

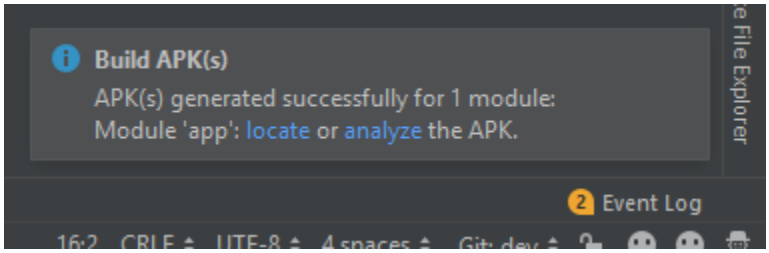

Figura 70 Mensaje de estado para la compilación de un archivo .APK

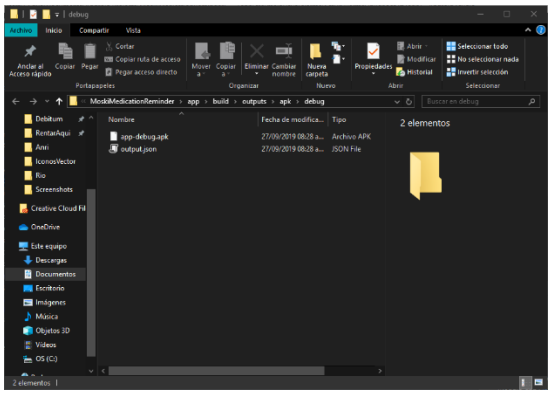

Figura 71 Archivo .APK compilado.

Al hacer clic en la etiqueta “locale” se abrirá el explorador de archivos y nos mostrará el archivo .APK, el cual puede ser instalado directamente en cualquier dispositivo Android que tenga habilitada la opción de instalar desde fuentes desconocidas.

Con este último paso la aplicación está completa, pues es posible utilizar todas sus funcionalidades desde una instalación hecha con el archivo .APK sin tener que depender del entorno de desarrollo Android Studio.

Capítulo 6: Trabajo futuro.

Gracias al proceso de arquitectura contamos con un proyecto modular de Android Studio, el cual podemos utilizar cómo base para otros proyectos de aplicaciones móviles. Tomando las clases, entidades y componentes que necesitemos e integrándolas en nuevos proyectos con apenas mínima modificación.

El archivo .apk generado anteriormente se encuentra listo para ser publicado en la Play Store o en cualquier otra tienda de aplicaciones Android, así como también podría distribuirse e instalarse manualmente sin problema, así que la posibilidad de crear una cuenta de desarrollador en Google Developer Console y publicar la aplicación para su descarga pública permitiría robustecer la aplicación pues de conseguirían estadísticas sobre su uso cómo cantidad de instalaciones, versión del sistema operativo utilizado, países donde se descarga, marca y modelo de los dispositivos, comentarios de los usuarios y por ende se conseguiría retroalimentación que puede ser usada para mejorar la aplicación . El hecho de que la aplicación se encuentre en los idiomas inglés y español permiten que la aplicación pueda llegar a tener un mayor alcance, pues sería más grande el rango de usuarios puedan hacer uso de ella.

Este proyecto de tesis ha permitido dominar la mayoría de los componentes fundamentales del sistema operativo Android así que la planeación y creación de aplicaciones futuras será un proceso mucho más simple, rápido y eficiente a comparación de lo que era antes de la realización de este trabajo.

Capítulo 7: Conclusiones

La metodología de desarrollo nos permitió crear un código fuente escalable al cual podremos añadir actualizaciones, correcciones y cambios con mayor facilidad según los nuevos requerimientos que vayan apareciendo dada la retroalimentación de los usuarios que se tengan, así como de los cambios que se vayan presentando en las nuevas versiones del sistema operativo Android y los dispositivos que hagan uso de este.

La metodología Extreme Programming nos permitió crear nuestro proyecto de una forma modular tratando de que las funciones fueran independientes unas de otras, pudiendo probarlas de forma independiente para tratar de encontrar todos los bugs y fallos posibles antes de fusionarlas con otros módulos para formar un solo producto, pero sobre todo nos permitió adaptarnos fácilmente a los cambios en los requerimientos.

El sistema de control de versiones fue una metodologías y herramientas que más apoyaron al desarrollo de este proyecto pues ayudaron crear el proyecto de una forma escalonada, a solucionar bugs detectados en poco tiempo y en caso de que sucediera un error critico poder dar un paso a atrás a una versión que funcione sin problemas.

En general podemos decir que la finalidad detrás de crear esta aplicación se ha cumplido y esperamos poder seguir adquiriendo conocimiento en el desarrollo de aplicaciones para el sistema operativo Android cada vez más complejas.

Los sistemas operativos y aplicaciones para dispositivos móviles han y continúan evolucionando día a día a tal punto que es posible crear aplicaciones que satisfagan un gran rango de necesidades cotidianas, rango que seguirá creciendo con el paso del tiempo, así como también veremos nuevos dispositivos donde podrán correr todas estas nuevas aplicaciones.

Bibliografía

[1] Dan Frommer. (2011). HISTORY LESSON: How The iPhone Changed Smartphones Forever. 10 de agosto del 2019, de Business Insider Sitio web: https://www.businessinsider.com/iphone-android-smartphones-2011-6

[2] Industry Leaders Announce Open Platform for Mobile Devices . Open Handset Alliance. 5 de noviembre de 2007. Consultado el 17 de febrero de 2012. Sitio web: http://www.openhandsetalliance.com/press_110507.html

[3] StatCounter. (2019). Mobile Operating System Market Share Worldwide. Septiembre 4, 2019, de StatCounter Sitio web: https://gs.statcounter.com/os-market-share/mobile/worldwide

[4] Google. (2019). Supported devices. Septiembre 29, 2019, de Google Sitio web: https://support.google.com/googleplay/answer/1727131?hl=en

[5] Oracle. (2006). Java ME Overview. Marzo 10, 2019, de Oracle Corporation Sitio web: http://www.oracle.com/technetwork/java/embedded/javame/index.html

[6] Stephen Colebourne. (2011). Kotlin and the search for a better Java. Noviembre 20, 2018, de Stephen Colebourne's blog Sitio web: https://blog.joda.org/2011/07/kotlin-and-search-for-better-java_9066.html

[7] Dmitry Jemerov and Svetlana Isakova. (2017). Kotlin in Action. USA: Manning Publications.

[8] Antonio Leiva. (2018). Kotlin for Android Developers 6th Edition. USA: Leanpub.

[9] Frederic Lardinois. (2017). Google makes Kotlin a first-class language for writing Android apps. Septiembre 12, 2019, de Tech Crunch Sitio web: https://techcrunch.com/2017/05/17/google-makes-kotlin-a-first-class-language-for-writing-android-apps/

[10] Claudia Juárez Escalona. (2018). Penetración de smartphones no se detiene. Enero 5, 2019, de El Economista Sitio web: Penetración de smartphones no se detiene

[11] Alex Genadinik. (2014). Mobile App Marketing and Monetization: How to Promote Mobile Apps Like A Pro: Learn to promote and monetize your Android or iPhone app. Get hundreds of thousands of downloads & grow your app business. England: Createspace Independent Pub.

[12] Jason Wei. (2012). Android Database Programming. California, USA: Packt Publishing Limited.

[13] Schach, S., Fernández, E., Guerrero, E., Ramírez, R. & Betancourt, S. (2006). Ingeniería de software clásica y orientada a objetos. México: McGraw-Hill.

[14] Mike van Drongelen, Adam Dennis, Richard Garabedian, Alberto Gonzalez, Aravind Krishnaswamy. (2017). Lean Mobile App Development: Apply Lean startup methodologies to develop successful iOS and Android apps. Berkeley, California, USA: Packt Publishing.

[15] Javier Cuello & José Vittone. (2013). Designing Mobile Apps. Barcelona, España: CreateSpace Independent Publishing Platform.

[16] Ashish Belagali, Hardik Trivedi, Akshay Chordiya. (2017). Kotlin Blueprints. United States: Packt Publishing.

[17] Kenneth S. Rubin. (2012). Essential Scrum: A Practical Guide to the Most Popular Agile Process (Addison-Wesley Signature Series (Cohn)) 1st Edition. Massachisetts, USA: Addison-Wesley Professional.

[18] Andrew Stellman, Jennifer Greene. (2013). Learning Agile: Understanding Scrum, XP, Lean, and Kanban 1st Edition. United States: O'Reilly Media.

[19] Google. (2017). Guía de arquitectura de apps. Enero 5, 2019, de Google Sitio web: https://developer.android.com/jetpack/docs/guide

[20] David González Verdugo. (2019). Componentes de arquitectura de Android, de MVC a MVVM. Septiembre 15, 2019, de Solid Gear Sitio web: https://solidgeargroup.com/componentes-de-arquitectura-de-android-de-mvc-a-mvvm/?lang=es

[21] Roger Dudler. (2011). git - la guía sencilla una guía sencilla para comenzar con git. sin complicaciones ;). Octubre 20, 2018, de GitHub Sitio web: https://rogerdudler.github.io/git-guide/index.es.html