Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA DE SEGOVIA

Grado en Ingeniería Informática de Servicios y Aplicaciones

Wearable Heart Beat Register-Apps: Registro automatizado de la frecuencia cardiaca en reposo a través de un smartwatch

---

Alumno: Alberto Ruiz Alejandro

Tutores: José Vicente Álvarez, Mario Corrales Astorgano

Índice general

Índice de figuras

Índice de cuadros

Agradecimientos

Me gustaría dar las gracias en primer lugar a mi familia, por haberme apoyado y acompañado durante todo el grado y sin la cual nunca podría haber llegado hasta aquí. Siempre han querido lo mejor para mí y han sabido darme los mejores consejos, en los buenos y no tan buenos momentos.

En segundo lugar, agradecer a los tutores D. José Vicente Álvarez Bravo y D. Mario Corrales Astorgano, por haberme dado la oportunidad de realizar este proyecto, en el que he aprendido multitud de cosas, seguro que muy útiles para mi futuro. Gracias por haber estado ahí para cualquier duda que me surgía.

Por último, dar las gracias a todos mis amigos y amigas que me han apoyado durante la carrera. Algunos de ellos han sido esenciales, sobre todo al inicio de ella, y me han permitido llegar hasta donde estoy hoy, razón por la cual lo menos que puedo hacer es dejar constancia de ello aquí.

Resumen

Los dispositivos wearables están cada vez más presentes en el día a día. Relojes, pulseras, auriculares o gafas son ya parte de nuestras vidas y nos ofrecen una mayor comodidad en distintos aspectos de nuestras vidas. Uno de ellos, y quizá el más importante, es la salud. Con ellos se pueden medir diferentes parámetros biométricos y llevar un registro de ellos. Además, determinadas patologías pueden incluso ser detectadas de manera precoz con bastante fiabilidad.

En este TFG se pretende crear una aplicación para smartwatches Wear OS, que sea capaz de monitorizar la frecuencia cardiaca en estado de reposo. Además, se podrá transmitir dicha medición al teléfono para llevar un registro de ello y algunas funcionalidades más, presentes todas ellas en una aplicación Android para smartphones. Con ello se pretende explorar estas nuevas tecnologías e integrar en el proyecto los componentes básicos de toda aplicación Wear OS.

Palabras claves: Wear OS, Android, Java, frecuencia, smartwatch, wearable.

Abstract

Wearable devices are increasingly present in everyday life. Watches, bands, headphones or glasses are already part of our lives and offer us greater comfort in different aspects of our lives. One of them, and perhaps the most important, is health. With them you can measure different biometric parameters and keep a record of them. In addition, certain pathologies can even be detected early with considerable reliability.

In this TFG it is intended to create an application for Wear OS smartwatches, which is capable of monitoring the heart rate at rest. In addition, said measurement can be transmitted to the phone to keep track of it and some other features, all present in an Android application for smartphones. This is intended to explore these new technologies and integrate into the project the basic components of any Wear OS application.

Key words: Wear OS, Android, Java, heart rate, smartwatch, wearable.

Parte I

Memoria del Proyecto

Capítulo 1

Descripción del proyecto

1.1. Introducción

En los últimos años, hay un tipo de dispositivos móviles que ha experimentado un gran crecimiento en comparación con el resto: los dispositivos wearables. Auriculares, relojes, pulseras, o incluso gafas que, además de desempeñar la función para la que fueron diseñados originalmente, cuentan con muchas otras que facilitan la vida cotidiana. Entre ellas, se encuentran la posibilidad de hacer llamadas, medir el ritmo cardiaco o localizarnos mediante GPS.

Los wearables más conocidos son los relojes inteligentes o smartwatches y las pulseras de actividad o smartbands. Hay un campo que se ha visto especialmente beneficiado por este tipo de dispositivos, el sanitario. En España más de 10 millones de personas padecen enfermedades y patologías relacionadas con el corazón, de las cuales más de 120.000 personas fallecen cada año según datos del INE [16]

Con estos dispositivos no solo se puede medir el ritmo cardiaco, como se comentaba anteriormente, sino que existen múltiples funciones a mayores como son la monitorización del sueño, los niveles de glucosa o la saturación de oxígeno. Además, muchos de los relojes que hay actualmente en el mercado poseen algoritmos de inteligencia artificial capaces incluso de detectar de manera temprana una patología cardiovascular o crisis epilécticas [14].

En este contexto surge la motivación de este trabajo fin de grado. El objetivo es desarrollar una aplicación Wear OS que sea capaz de monitorizar la frecuencia cardiaca en situaciones de reposo. Para ello, se utilizaran los sensores de movimiento, como el acelerómetro o el GPS. A mayores, se implementará una aplicación Android para smartphones, la cual tendrá un registro de esas medidas que hemos hecho con el reloj, a partir de las cuales podría plantearse en un futuro el análisis de dichos datos en busca de alguna anomalía, como por ejemplo la detección de arritmias.

1.2. Objetivos del trabajo

La finalidad principal de este trabajo es la implementación de una aplicación que monitorice la frecuencia cardiaca en situación de reposo y guarde esa información en una base de datos. Para ello se deberán llevar a cabo las siguientes tareas:

- Implementación de la app para el reloj, la cual estará programada con el sistema operativo Wear OS de Google, y tendrá las siguientes funciones:
  - Monitorizar si la persona se encuentra en una situación de reposo.
  - En caso de estar en estado de reposo, monitorizar la frecuencia cardiaca para obtener los valores medios durante esos intervalos.
  - Almacenar esos valores en una base de datos.
- Implementación de una app Android para smartphones con lenguaje de programación Java, que sea capaz de vincularse con la app del reloj y ofrezca una serie de funcionalidades a mayores, tales como las siguientes:
  - Un sencillo tutorial de manejo tanto de la app para teléfonos como para la de relojes.
  - Visualización del historial de los datos de frecuencia cardiaca en resposo a lo largo del día, semana, mes o año (mediante gráficos de dichas mediciones).

1.3. Entorno de aplicación

En esta sección introduciremos dos aplicaciones para dispositivos wearables que utilizan tecnologías similares. Actualmente, aunque hay más apps Wear OS que hace tres o cuatro años en Google Play, siguen siendo pocas comparadas con otros tipos de dispositivos, como los smartphones. Por ello, es difícil hacer una comparación realista entre la que se presenta en este trabajo y las que están en el mercado.

Además, estas apps suelen pasar por los dos extremos, algunas son muy sencillas y solo miden una característica concreta, como la frecuencia cardiaca, y otras hacen uso de casi todas las tecnologías del reloj. Este es el caso de las aplicaciones que vienen preinstaladas en el smartwatch, o alguna otra que se encuentra en la tienda de google.

Por ello, tras una reflexión, se optó por comparar este proyecto con otros trabajos fin de grado cuyo objetivo también fuese desarrollar una aplicación para un smartwatch. Haciendo una búsqueda rápida en internet, se encuentran un par de proyectos muy interesantes, debido a que no coinciden totalmente con las tecnologías que se han usado en este proyecto, por lo que muestran más formas de desarrollar aplicaciones para wearables. Se describirán de manera resumida a continuación, y finalizaré la sección con una comparativa de los tres. Las memorias de ambos proyectos se pueden ver respectivamente en [7] y [17]

1.3.1. Aplicación de asistencia remota para personas con necesidades especiales

Este TFG fue realizado por Tomás Acosta Almeda, estudiante del grado en Ingeniería de las Tecnologías Industriales de la Universidad de Sevilla, en 2015. El objetivo del proyecto era desarrollar una aplicación que fuese capaz de detectar una caída de una persona que lleve puesto un smartwatch, y que acto seguido se mandara un correo electrónico indicando las coordenadas donde supuestamente podía habérsele caido.

La particularidad por la que fue elegido este proyecto ha sido el reloj elegido, junto con el lenguaje de programación que utiliza. El smartwatch se conoce como Pebble, y muchos lo consideran como uno de los promotores de la industria de los relojes inteligentes. En 2012 se llegaron a recaudar 10 millones de dolares en Kickstarter, una plataforma que ayuda a impulsar proyectos creativos mediante donaciones. En los siguientes dos años, el reloj tuvo éxito y consiguieron 230 millones de dolares en ventas. Sin embargo, la empresa fracasó y se vio obligada a cerrar cuando sacaron a la venta en 2015 la segunda edición del smartwatch: el Pebble Time. Éste tuvo unas ventas por debajo de las esperadas y esto desencadenó que en 2016 tuviese que cerrar la compañía. En la figura 1.1 se puede ver una imagen del reloj Pebble.

Figura 1.1: Pebble Watch en diferentes colores [20]

Tras este breve resumen del auge y caida del pebble, veamos las tecnologías usadas

en el proyecto. El lenguaje sobre el que se sustenta la programación en Pebble es C, lo cual ya difiere bastante de nuestro caso. No es C puro, pero comparte con él la mayoría de características. Para la detección de la caida, se utiliza exclusivamente el acelerómetro del reloj. Finalmente, comparte con nuestro TFG el lenguaje en el que se programó en el móvil: Android con Java.

La aplicación del móvil se utiliza para detectar la ubicación y mandar el correo electrónico con la ubicación. Esto hace que realmente se obtenga la ubicación del móvil, pero dado que debería estar conectado con el Pebble para recibir los mensajes, la ubicación real del reloj debe de estar a unos metros de distancia supuestamente. De todas formas, claramente no es la solución más óptima, ya que obliga a tener siempre en la misma ubicación tanto al reloj como al smartphone.

La comunicación con el reloj es mediante un sistema definido en el propio entorno de programación de Pebble. Cuando el reloj detecta una caída, deja un margen de 15 segundos, y entonces conecta con el teléfono. Éste debe tener conexión a internet, para que posteriormente mande el correo electrónico con las coordenadas de la caída. Merece la pena destacar que este smartwatch no es táctil y está hecho enteramente de plástico. Esto se eligió así debido a la finalidad que tenía, y que para hacer las pruebas aguantaría mucho mejor que uno con pantalla de cristal.

1.3.2. Monitorización y recomendación de ejercicio físico a través de pulsera/reloj inteligente

Este proyecto lo hizo Miguel Pardal Martín, estudiante del grado en Ingeniería del Software de la Universidad de Málaga, en 2021. Por lo tanto, es bastante más reciente que el que acabamos de ver. En este caso, como bien indica el nombre, el objetivo era desarrollar una aplicación para el smartwatch que monitorizase ciertos datos, mandarlos y almacenarlos en una base de datos. Además, con una aplicación móvil que también fue implementada, el usuario es capaz de visualizar dichos datos mediante gráficos, y además recibir recomendaciones de rutinas de ejercicio físico. Como vemos, este TFG se asemeja bastante más a lo que se desarrolla en este proyecto: tenemos dos aplicaciones (una en el wearable y otra en el smartphone), y la del reloj monitoriza datos, mientras que el móvil los recibe y muestra por pantalla.

En cuanto a los lenguajes de programación, este TFG es bastante completo. Ninguno de ellos coincide con los que se han utilizado en este proyecto, y por ello de nuevo sirve para que exploremos un poco más otras posibilidades de desarrollo de aplicaciones móviles, y en concreto dispositivos wearables.

Para el lado del servidor, el lenguaje principal utilizado es JavaScript, junto con el entorno de ejecución de código Node.js. Para el almacenamiento de los datos utiliza tanto firebase como MongoDB, dos de las bases de datos no relacionales más utilizadas actualmente. Para el lado cliente utiliza flutter. Tanto la app para el smartwatch como la del

smartphone están programadas en flutter.

Figura 1.2: Interfaz gráfica de la pantalla principal de la app

Flutter es un framework del lenguaje Dart, el cual fue lanzado por google en el 2011 como una alternativa más moderna a JavaScript. La primera versión de flutter salió al mercado en el 2015, por lo que es mucho más moderna que Java. Esto hace que haya mucha menos documentación y sea más difícil de aprender, sobre si todo si no estás familiarizado con Dart. Sin embargo, Flutter tiene una serie de ventajas bastante notables que hace que su popularidad esté creciendo últimamente. La más importante es que permite desarrollar aplicaciones multiplataforma para Android, IOs y la web desde una misma base de código. Además es mucho más fácil crear aplicaciones más estéticas, y sus apps tienen un alto rendimiento en cuanto al renderizado de las vistas.

Aparte de las funciones ya comentadas, la aplicación móvil tiene un par de funciones más que merece la pena mencionar. La primera es la gestión de usuarios, lo cual permite que aunque el usuario se cambie de dispositivo siga podiendo acceder al histórico de sus datos monitorizados. La segunda es la capacidad de vincular mediante bluetooth tu reloj con la app del móvil. Aunque esto no sea necesario en la mayoría de casos, ya que se puede hacer de manera externa, es una función interesante para hacer que la aplicación sea completamente autosuficiente. En la figura 1.2 se ven capturas de la interfaz de la aplicación móvil. El smartwatch se encarga de monitorizar exclusivamente, por lo que no tiene una interfaz propia.

1.3.3. Comparativa

Para finalizar la sección, se presenta a continuación una comparación entre este trabajo y los dos analizados anteriormente. Para ello, la tabla 1.1 muestra un resumen de las características de cada uno de ellos, que reúne todo lo que ha sido discutido y sirve para rápidamente entender las partes en común y las que difieren de cada uno de los trabajos.

Característica	**HeartRest**	**TFG US**	**TFG UMa**
Año	2023	2015	2021
Lenguaje de programación	Wear OS y Android (Java)	Pebble (C) y Android (Java)	JavaScript y Flutter (Dart)
App para reloj y móvil	Sí	Sí, pero sin interfaz la del móvil	Sí, pero sin interfaz la del reloj
Sensores utilizados	Acelerómetros	Acelerómetros	Pasos, Calorías
GPS utilizado	Sí	Sí (el del teléfono)	Sí
Dispositivos compatibles	Relojes Wear OS y móviles Android	Relojes Pebble y móviles Android	Relojes y móviles tanto IOs como Android

Cuadro 1.1: Tabla comparativa de los proyectos descritos en esta sección

1.4. Entorno y tecnologías utilizadas

Esta sección tiene como objetivo exponer las tecnologías principales que se han usado en el trabajo fin de grado. Para ello, se resume a continuación los dos sistemas operativos utilizados, Android y Wear OS, y el lenguaje de programación utilizado, Java.

1.4.1. Android

Android es un sistema operativo móvil basado en una versión modificada del kernel de Linux y otro software de código abierto, diseñado principalmente para dispositivos móviles con pantalla táctil, como teléfonos inteligentes y tabletas. Android es desarrollado por un consorcio de desarrolladores conocido como Open Handset Alliance, aunque su versión más utilizada es desarrollada principalmente por Google.

En esencia, el sistema operativo se conoce como Proyecto de código abierto de Android (AOSP) y es un software gratuito y de código abierto (FOSS) con licencia principalmente

bajo la Licencia Apache. Sin embargo, la mayoría de los dispositivos se ejecutan en la versión patentada de Android desarrollada por Google, que se entrega con software patentado adicional de código cerrado preinstalado, en particular, Google Mobile Services (GMS), que incluye aplicaciones principales como Google Chrome, el plataforma de distribución digital Google Play, y la plataforma de desarrollo de Google Play Services asociada.

Figura 1.3: Logo de Android [14]

El código fuente de Android se ha utilizado para desarrollar variantes de Android en una variedad de otros dispositivos electrónicos, como consolas de juegos, cámaras digitales, reproductores multimedia portátiles y PC, cada uno con una interfaz de usuario especializada. Algunos derivados bien conocidos incluyen Android TV para televisores y Wear OS para dispositivos portátiles, ambos desarrollados por Google. Wear OS es el que se ha utilizado para este proyecto y será explicado más adelante.

Android ha sido el sistema operativo más vendido en todo el mundo en teléfonos inteligentes desde 2011 y en tabletas desde 2013. En mayo de 2021, tenía más de tres mil millones de usuarios activos mensuales, y en enero de 2021 , Google Play Store presentó más de 3 millones de aplicaciones. El logo de Android se puede ver en la figura 1.3. Como curiosidad, fue creado en 2007 por Irina Block, una diseñadora gráfica de Google [13].

Respecto a su arquitectura, se puede ver en la figura 1.4. De arriba a abajo y de manera resumida se divide en las siguientes capas:

- **Aplicaciones:** las aplicaciones básicas de Android incluyen herramientas como un correo electrónico, mensajes de texto, calendario, mapas, navegador, contactos, y numerosas aplicaciones más. Todas estas aplicaciones están escritas en el lenguaje de programación Java, que será explicado un poco más adelante, pues también es el lenguaje usado en este proyecto.

- **Frameworks de las aplicaciones:** es un conjunto de herramientas de software que permite a los desarrolladores de aplicaciones armar un producto terminado que cumpla con los requisitos de su propietario.

Figura 1.4: Arquitectura de Android [1]

- **Bibliotecas:** Android incluye un conjunto de bibliotecas de C/C++ que son utilizadas por diversos componentes del sistema. Estas características son expuestas a los desarrolladores a través del framework de aplicaciones de Android. Por ejemplo, la de SQLite se ha utilizado en este proyecto. Se explicará con más detalle en el capítulo de implementación.

- **Runtime de Android:** es un entorno de ejecución de aplicaciones utilizado por el sistema operativo móvil Android. ART reemplaza a Dalvik, que es la máquina virtual utilizada originalmente por Android, y lleva a cabo la transformación de la aplicación en instrucciones de máquina, que luego son ejecutadas por el entorno de ejecución nativo del dispositivo.

- **Kernel de Linux:** Android depende de Linux para los servicios base del sistema como seguridad, gestión de memoria, gestión de procesos, pila de red y modelo de controladores. El núcleo también actúa como una capa de abstracción entre el hardware y el resto de la pila de software.

1.4.2. Wear OS

Wear OS (también conocido como Wear y anteriormente como Android Wear) es una versión del sistema operativo Android de Google diseñado para relojes inteligentes y otros dispositivos portátiles. Se puede vincular con teléfonos móviles con Android versión 6.0

"Marshmallow" o superior, o con una versión IOs superior a 10.0 con soporte limitado de la aplicación de emparejamiento de Google. Su logo se puede ver en la figura 1.5.

Figura 1.5: Logo de Wear OS [24]

Wear OS admite conectividad Bluetooth, NFC, Wi-Fi 3G y LTE, así como una variedad de funciones y aplicaciones. Las formas de las pantallas de los relojes pueden ser cuadrada, circular o rectangular. Diversas marcas poseen relojes con este sistema operativo, entre las que se encuentran Samsung, Oppo o Fossil. Respecto a la arquitectura, es esencialmente la misma al ser Android, pero existen algunas diferencias, que se comentan brevemente a continuación.

Wear OS está diseñado para ejecutarse en dispositivos con menos recursos que Android, por lo que su arquitectura está optimizada para funcionar con limitaciones de hardware que no existen en los teléfonos móviles. Además, tiene una mayor dependencia de la comunicación con un smartphone emparejado para acceder a datos y servicios en línea. Por lo tanto, la arquitectura de Wear OS está diseñada para trabajar de manera más estrecha con un dispositivo móvil en comparación con Android. Por último, Wear OS y Android tienen diferentes características requeridas para su funcionamiento óptimo. Por ejemplo, Wear OS no necesita características como una gran pantalla, cámaras o puertos de entrada/salida, ya que estos dispositivos están diseñados para ser portátiles y pequeños. Esto hace que difieran a mayores en otros aspectos de la arquitectura para adaptarse a las necesidades de los dispositivos wearables.

1.4.3. Java

El lenguaje Java es lo único que ya era familiar de antes debido a las múltiples asignaturas que ha habido durante el grado. Sin embargo, en ninguna de ellas se usó para programar Android. Java es un lenguaje de programación de alto nivel, basado en clases y orientado a objetos que está diseñado para tener la menor cantidad posible de dependencias de implementación.

Java es el lenguaje oficial de desarrollo Android. Hay una gran comunidad creada en torno a él, y debido a ello y a la familiaridad con el lenguaje se optó por este lenguaje frente a otros como Kotlin o el framework Flutter. Una característica importante al utilizar Java

para el desarrollo de aplicaciones Android es que éste se basa en el modelo de programación orientada a eventos (se explica con más detalle en el capítulo de implementación), donde los eventos son generados por el usuario o por el sistema operativo.

Estos eventos son manejados por los componentes de la aplicación, como las activities, fragments, servicios y broadcast receivers, que son los bloques de construcción básicos de una aplicación Android. No se entrará más en detalle sobre Java en este apartado, sino que a lo largo de la memoria se irán explicando los componentes que han sido necesarios en el proyecto para cada una de las funcionalidades implementadas.

1.5. Estructura de la memoria

La memoria del proyecto está dividida en tres partes, las cuales a su vez contienen varios capítulos. A continuación se resume brevemente cada uno de ellos:

- **Parte I:** Memoria del proyecto. En esta parte se encuentran los siguientes capítulos:
  - **Introducción:** se presenta en él una introducción al tema del TFG, así como los objetivos del proyecto y el estado del arte, comparándolo con otros dos trabajos similares cuyo objetivo era desarrollar una app para dispositivos wearables.
  - **Metodología:** en este capítulo se explica como se ha llevado a cabo el desarrollo, que metodología se ha utilizado y las herramientas utilizadas durante la realización del mismo.
  - **Planificación:** este capítulo contiene las estimaciones de tiempo y presupuesto que ha requerido el trabajo. Para ello se incluye una planificación inicial, realizada previamente al desarrollo, una planificación final, en la que se comenta cual ha sido realmente el esfuerzo dedicado en comparación a lo que se estimó, y el presupuesto necesitado en el proyecto.
- **Parte II:** Documentación técnica. Contiene los siguientes capítulos:
  - **Análisis:** una parte fundamental de todo proyecto software es la ingeniería de requisitos. Esto es lo que se encuentra en este capítulo. En concreto, contiene los requisitos del proyecto, los diagramas de casos de uso, las historias de usuario derivadas de ellos, el modelo de dominio y los diagramas de actividad.
  - **Diseño:** contiene tanto la arquitectura y patrones de diseño utilizados en las apps creadas como distintos diagramas, incluyendo el diagrama de la base de datos, los de paquetes y el de despliegue. También se expone en este capítulo la estructura final del proyecto desarrollado.
  - **Implementación:** este capítulo resume como se han implementado ambas apps. Incluye como se ha detectado el reposo, cuales son las opciones que hay para monitorizar la ubicación y los acelerómetros (junto con la opción elegida),

la comunicación entre el reloj y el teléfono y las librerías usadas para visualizar los datos de las frecuencias almacenadas. Se termina el capítulo comentando cuales fueron las mayores dificultades que surgieron durante el desarrollo.

- **Pruebas:** el último capítulo de esta parte resume las pruebas software realizadas. Se trata de pruebas unitarias utilizando la librería JUnit en Android Studio.

- **Parte III:** Manuales de la aplicación: se incluyen dos manuales con distinto propósito:

  - **Manual de Instalación:** este manual explica los pasos necesarios desde cero para la instalación de ambas apps. Requiere la instalación de Android Studio, clonar el proyecto de Github y activar la depuración en los dispositivos. Se explica de manera detallado todo esto en él.
  - **Manual de Usuario:** una vez instaladas las apps, este manual explica como utilizarlas y las diferentes funcionalidades que tienen.

Por último, la memoria finaliza con la bibliografía necesaria durante todo el proyecto.

Capítulo 2

Metodología

2.1. Proceso de desarrollo

Tras analizar los requisitos y objetivos de este trabajo, se decidió en su momento seguir una metodología iterativa e incremental. Las metodologías ágiles, aunque poseen muchas ventajas, no son óptimas en este caso, ya que se cuenta con un número reducido de personas, y dichas metodologías se aplican cuando el número de personas es mayor.

La mayor ventaja que tiene esta metodología es que, con cada iteración, se obtiene un producto funcional. De esta manera, al cliente se le da la oportunidad de probar el producto desde muy temprano en el proyecto. Además, en cada iteración, no solo se va añadiendo funcionalidad a lo que ya se tenía, sino que se pueden corregir defectos o cambiar alguna funcionalidad con la que el cliente no esté del todo satisfecho. Por tanto, al cliente se le da un rol activo durante todo el proceso de desarrollo.

En la figura 2.1 se puede ver un esquema con cada una de las fases que tienen lugar en cada iteración del desarrollo. Como se puede apreciar, cada iteración consta de una planificación y unos requisitos. Luego, se lleva a cabo el análisis y el diseño del sistema de acuerdo a los requisitos fijados y la planificación hecha. Finalmente, se prueba y evalúa, y se vuelve a empezar una nueva iteración. Cuando ya no hagan falta más iteraciones pasamos al despliegue o distribución del proyecto.

Figura 2.1: Modelo de desarrollo iterativo [8]

2.2. Herramientas utilizadas

Durante el transcurso del trabajo fin de grado ha sido necesario utilizar múltiples tecnologías, cada una de las cuales cumplía una función determinada. A continuación, se mencionan y describen brevemente las razones de la elección de dichas herramientas, en caso de que sea necesario.

Entorno de trabajo

1. El proyecto ha sido realizado enteramente en un ordenador portátil cuyo sistema operativo es windows 10.

2. Para probar el proyecto se han utilizado dos dispositivos móviles: un smartphone (el del autor del TFG), que tiene sistema operativo Android y, por tanto, es compatible con la aplicación creada, y el smartwatch Samsung Galaxy Watch 4, que fue adquirido al principio del proyecto. Tiene sistema operativo Wear OS con la capa de personalización de Samsung, y en él hemos probado la app Wear OS que hemos creado.

Lenguaje de programación

1. El lenguaje de programación utilizado en la totalidad del proyecto ha sido Java orientado a objetos. Las apps Android se pueden programar en múltiples lenguajes, como Dart, Kotlin o Java. Kotlin es un lenguaje desarrollado por JetBrains que es muy popular actualmente, pero debido a que se disponían más conocimientos en Java, se optó finalmente por hacerlo con él y descartar las otras opciones. Con Java se pueden hacer aplicaciones Android y Wear OS de manera similar.

2. El IDE utilizado ha sido Android Studio. Con él se tiene la opción de trabajar tanto con Android como Wear OS de manera sencilla. Además, esta herramienta permite programar tanto en Java como en Kotlin, que ya describimos anteriormente.

Documentación

1. La memoria ha sido realizada en el procesador de textos latex. En particular, se ha utilizado su versión online overleaf. Con bastante facilidad se pueden hacer documentos que queden realmente bien.

2. Para la realización de los diferentes diagramas UML que se han incluido, se ha utilizado la web Draw.io [9], la cuál ya sabía utilizar por asignaturas previas en el grado.

3. Para realizar gráficos de la memoria, se ha utilizado también Microsoft Excel, debido a su simplicidad y fácil manejo.

4. Para editar las fotos que se han incluido en la memoria, se ha utilizado el editor online Canva [3].

Control de versiones

Para no perder los datos y llevar un seguimiento de lo que se iba realizando, se ha utilizado Github. Se integra de manera sencilla con Android Studio y también se ha utilizado en diversas asignaturas a lo largo de la carrera.

Comunicación

La comunicación con los tutores se ha llevado a cabo durante todo el proyecto mediante el uso conjunto de un canal de teams y el correo electrónico. El canal de teams ha sido útil para compartir toda la documentación que se iba realizando, mientras que la comunicación se ha llevado a cabo tanto por dicho canal como por correo electrónico.

Capítulo 3

Planificación

3.1. Planificación temporal

3.1.1. Planificación inicial

En esta sección se detalla la organización prevista al inicio del TFG para llevar a cabo todos los objetivos y cumplir todos los requisitos dentro del marco de tiempo propuesto. Se utilizará una metodología ágil basada en iteraciones o incrementos que irán sumando funcionalidad hasta obtener el producto final. A continuación se expone tanto el reparto de horas previsto para cada una de las iteraciones, como un análisis de los riesgos que pueden demorar el desarrollo del trabajo.

El reparto de horas se ha hecho basándose en proyectos similares que se han realizado durante la carrera. Por ello, una vez repartidas las funcionalidades previstas para cada iteración, se ha calculado según lo anterior una duración en horas estimada de cada una de ellas.

Distribución temporal

El TFG tiene previsto realizarse entre enero y abril de 2023. La primera iteración comienza el 16 de enero de 2023, y la última iteración finalizará el 16 de abril de 2023. Por tanto, se cuenta con tres meses para llevarlo a cabo. Los días de trabajo serán de lunes a domingo, es decir, todos los días de la semana se avanzará con el TFG.

El ritmo de trabajo durante todo este tiempo será el mismo. Por ello, se dedicarán cuatro horas todos los días. Teniendo en cuenta que entre el 16 de enero y el 16 de abril hay un total de 91 días, se tendrían 364 horas para invertir en el TFG. Dado que la normativa actual contempla 300 horas, tendría un margen de 64 horas para posibles retrasos o mejoras del proyecto. Además, aunque en principio pueda trabajar todos los días, a lo largo de estos cuatro meses seguro que algún día no puedo dedicarle cuatro horas (y otros días seguramente dedique más), por lo que finalmente las horas que realmente haga se ajustarán más a las 300 horas que están puestas de referencia.

A mayores de lo comentado, ha habido una etapa previa a la reunión de inicio de enero. Dado que no se disponian de conocimientos previos en las tecnologías que se manejan, se han empleado los meses de septiembre de 2022 a enero de 2023 en aprender los conocimientos básicos que permitiesen afrontar el TFG de manera correcta. Además, desde mediados de diciembre, se ha estado leyendo documentación y haciendo alguna que otra aplicación sencilla orientada a lo que sería mi TFG. Por ello, la etapa de familiarización ya considero que ha sido en su gran mayoría realizada, y paso directamente al desarrollo en sí del TFG.

Dicho esto, se detalla a continuación las iteraciones previstas, con su duración estimada, y un breve resumen de lo que se realizará en ellas:

- **Iteración 1 (100 horas):** Esta primera iteración tiene 2 objetivos principales. El primero es la implementación de una aplicación que sea capaz de usar los servicios GPS y los acelerómetros para así saber si estamos en reposo o no. El segundo es realizar una aplicación que mida la frecuencia cardiaca cada cierto tiempo, que será fijo en un principio. Todo ello se juntará en una sola aplicación que muestre todos esos datos por pantalla.

- **Iteración 2 (100 horas):** Una vez que funcionen las aplicaciones realizadas del GPS y la frecuencia cardiaca, el siguiente paso es integrar las dos. Antes de empezar esta iteración tendremos dos aplicaciones, que aportan dos funcionalidades independientes. El objetivo ahora es juntarlas y que la aplicación sea capaz de monitorizar la frecuencia cardiaca cada cierto tiempo, siempre que nos encontremos en estado de reposo.

- **Iteración 3 (100 horas):** A estas alturas se tendrá una aplicación funcional que realiza justamente lo que buscábamos en este trabajo. Sin embargo, nos queda una parte muy importante, que se lleva a cabo en esta iteración: la transmisión y almacenamiento de las frecuencias cardiacas que monitorizamos. A lo largo del TFG, se decidirá si los datos los almacenamos en un servidor, o si implementamos una app para smartphones en Android que reciba esos datos y los almacene en una base de datos local.

- **Iteración 4 (64 horas):** Por último, las últimas semanas del TFG se dedicarán a crear una interfaz de usuario amigable tanto en la app del reloj como en la del móvil (si decidimos hacerla finalmente en vez de usar un servidor). La app del móvil es en la que se centrará casi toda la iteración, pues el usuario deberá ser capaz de ver un registro de las frecuencias cardiacas medidas a lo largo del tiempo.

Como se habrá podido observar, no se ha mencionado la realización de la memoria en ninguna de las iteraciones. En un principio, la memoria se irá confeccionando a medida que avanzamos en el TFG, por lo que todas las iteraciones tendrán su parte correspondiente de memoria, pero no se indica ni cuantas horas ni las partes de la memoria que se hacen en cada una de las iteraciones.

En la figura 3.1 se muestran las horas que se deican a cada iteración, frente al número de horas que se van acumulando hasta llegar a las 364 que se comentaban anteriormente.

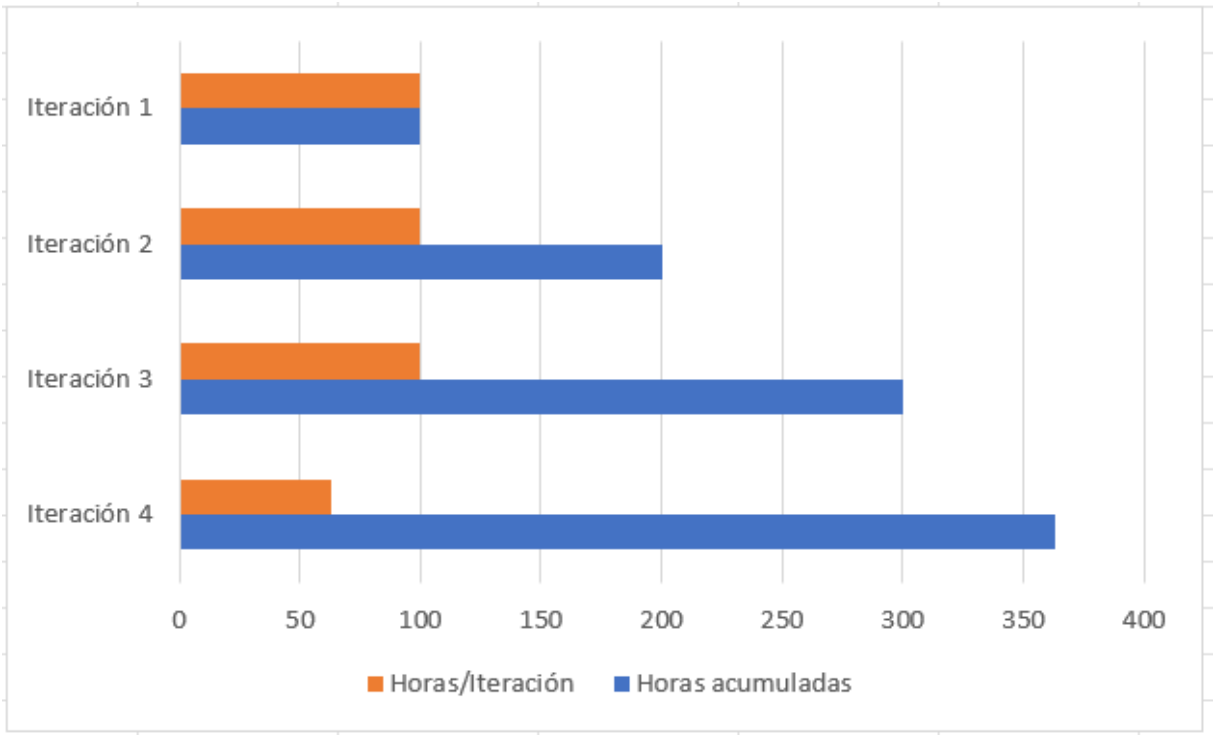

Figura 3.1: Distribución del número de horas de desarrollo de cada iteración

Análisis de riesgos

A lo largo de cuatro meses propuestos para realizar el TFG, pueden surgir problemas que retrasen lo que se había planificado inicialmente. Por ello, conviene que se identifiquen y determinar de manera estimada que tan probable es que ocurra alguno de ellos y cuanto retraso supondría. En la tabla 3.1 se pueden ver los riesgos que se consideran más probables en el TFG. Cada uno de ellos tiene un identificador, su nombre, la probabilidad de que ocurra, una breve descripción y el retraso en horas que supondría. Debido a que el ritmo que iba a seguir era el de 4 horas por día, se entiende que cada 4 horas supondría un retraso de 1 día.

Una vez identificados cuales son los riesgos que tienen más probabilidad de ocurrir, se deben establecer una serie de medidas que, por un lado, eviten que dicho riesgo ocurra, y que en caso de ocurrir, se minimice el impacto sobre el proyecto. Esto se conoce respectivamente como plan de reducción y plan de mitigación. Hay un último factor a tener en cuenta en el análisis de riesgos. Dada la tabla 3.1, hay riesgos que son mucho más probables que otros, por lo que el retraso estimado en horas no da una información tan útil como cabría esperar. Por ello, si se multiplica la columna de probabilidad por la del

ID	Riesgo	Probabilidad	Descripción	Retraso
1	Salud	0.05	Contraer alguna enfermedad de gravedad media o alta	25
2	Falta de experiencia en Android	0.15	Tardar más tiempo en implementar las funcionalidades	30
3	Falta de experiencia en Wear OS	0.3	Tardar más tiempo en implementar las funcionalidades	50
4	Planificación incorrecta	0.2	No ajustarse a la planificación inicial que estaba prevista	25
5	Problemas con el ordenador	0.05	El ordenador se pierde o se estropea	15
6	Pérdida de datos	0.05	Durante el transcurso del TFG se borra parte del trabajo que se llevaba hecho de manera accidental	20
7	Añadir/Eliminar funcionalidades	0.2	Cambiar algunos requisitos de los establecidos al principio del TFG	20

Cuadro 3.1: Riesgos a los que estamos expuestos durante la realización del proyecto

retraso se obtiene lo que se conoce como exposición al riesgo, el cual da una medida más realista del retraso que supone en el proyecto el riesgo en cuestión. Toda esta información se encuentra en la tabla 3.2.

ID	Riesgo	Exposición	Plan de reducción	Plan de mitigación
1	Salud	1.25	Intentar no acudir a sitios donde puedo contagiarme de algo y no coger frío	Cuidarme para intentar estar bien lo antes posible, así como echar más horas en los días posteriores para compensar los días perdidos
2	Falta de experiencia en Android	4.5	Leer documentación de lo que voy a implementar antes de ponerme a ello	Dedicarle más horas de las que en un principio había planeado
3	Falta de experiencia en Wear OS	15	Leer documentación de lo que voy a implementar antes de ponerme a ello	Dedicarle más horas de las que en un principio había planeado
4	Planificación incorrecta	5	Dar suficiente margen en los plazos establecidos para intentar no salirnos de ellos	Hacer un esfuerzo extra cuando considere que vaya por detrás de lo que había planificado
5	Problemas con el ordenador	0.75	Cuidar bien el ordenador	Arreglar el ordenador o comprar uno nuevo lo antes posible
6	Pérdida de datos	1	Ir haciendo copias periódicamente de mi progreso tanto locales como en la nube	Dedicar más horas para compensar el progreso perdido
7	Añadir/Eliminar funcionalidades	4	Ajustarme a los requisitos iniciales e intentar cumplirlos	Tener reuniones para saber que requisitos cambian y como se implementarían

Cuadro 3.2: Información adicional sobre los riesgos del proyecto.

3.1.2. Planificación final

Una vez ha sido realizado el proyecto casi en su totalidad, es necesario analizar como ha sido finalmente el desarrollo del mismo, para ver en que medida se ha ajustado la realidad a lo que inicialmente se estimó. Esto es lo que se discutirá en esta sección. Para ello, a continuación se describirán cada una de las iteraciones en las que se dividió inicialmente el proyecto, describiendo lo que ha llevado cada una de ellas y, en caso de una gran diferencia de tiempo entre lo real y lo estimado, una justificación de este desvío temporal, apoyándose en los riesgos que se expusieron anteriormente.

Descripción de las iteraciones

A continuación se detalla lo que se ha desarrollado en cada iteración:

- **Iteración 1:** esta iteración tenía como objetivo familiarizarse con los APIs de la ubicación y los sensores en Android, para realizar una sencilla app que monitorizase el estado de reposo en función de ellos. En cuanto a la funcionalidad, se ha hecho lo que predijo inicialmente. Sin embargo, en cuanto al tiempo que llevó se encuentra algo de diferencia. Mientras que inicialmente se estimaron 100 horas, la realidad es que bastaron 3 semanas para tener lo que se pedía en esta funcionalidad. Esto quiere decir que fueron necesarias únicamente 84 horas para esta iteración. Esto se debe principalmente a los riesgos número 2 y 3 respecto a la falta de experiencia tanto en Android como Wear OS. Lo que inicialmente puede parecer complicado, finalmente no lo es tanto, y viceversa, como veremos en posteriores iteraciones.

- **Iteración 2:** el objetivo de esta iteración era conseguir una primera versión funcional de la principal función de este proyecto: monitorizar la frecuencia cardiaca en estado de reposo. Los componentes se tenían ya gracias a la iteración 1, y solo quedaba juntarlos. De nuevo, lo desarrollado se ha ajustado a lo que en un principio se estableció. En cuanto al tiempo que se ha tardado, en este caso se han dedicado un número de horas ligeramente superior al estimado, de nuevo debido a los riesgos 2 y 3. En concreto se han dedicado 4 semanas a esta iteración, o equivalentemente, 112 horas. Aún así, no es una diferencia muy pronunciada respecto a lo que se planificó inicialmente.

- **Iteración 3:** en esta iteración hemos desarrollado las funcionalidades de transmisión y almacenamiento de las frecuencias que se monitorizan en reposo. Decidimos que lo mejor sería guardarlo en una base de datos local del teléfono. En esta iteración es donde la diferencia entre lo planificado y lo real ha sido mayor, de nuevo debido a los riesgos 2 y 3. Esto se ha debido a dos factores principalmente: que la API DataLayer es sencilla de utilizar y facilita mucho esta tarea, y que se disponían de conocimientos previos en cuanto a SQLite, la base de datos local de Android que se ha utilizado. Por ello, en dos semanas o, equivalentemente, 56 horas, se ha conseguido implementar todo lo previsto en esta iteración.

- **Iteración 4:** la última iteración se ha dedicado a la implementación de la app del teléfono junto con retoques de interfaz también en la app del reloj. En este caso, se han vuelto a dedicar ligeramente más horas que las estimadas, 84 (3 semanas). Las causas principales de esto han sido los riesgos número 2 y 7. Respecto al 7, mientras se desarrollaba la app se han incluido algunas funcionalidades, no muy complejas, pero que han supuesto esa variación en la estimación.

Por último, fuera de lo que se planificó inicialmente se ha retrasado a mayores una semana el proyecto debido al riesgo 7. Mientras que en un principio el reposo se iba a medir en función de los acelerómetros y la ubicación, finalmente la ubicación no se ha considerado, y lo que se hace es almacenarla junto con la frecuencia para mostrar esta información en la app. Aparte, aunque en general si se ha seguido la planificación inicial, durante el transcurso del TFG se iban detectando fallos, pero sin suponer un gran retraso debido a que eran fallos menores.

Si se suman las horas que se han dedicado, se ve que sí que se ajusta al número de horas prevista (con una diferencia de una semana aproximadamente), aunque el reparto de horas no haya sido exactamente el mismo. En la figura 3.2 se muestra un gráfico que compara las horas que se han estimado frente a las reales en todas las iteraciones, para tener un resumen visual de todo el apartado de planificación.

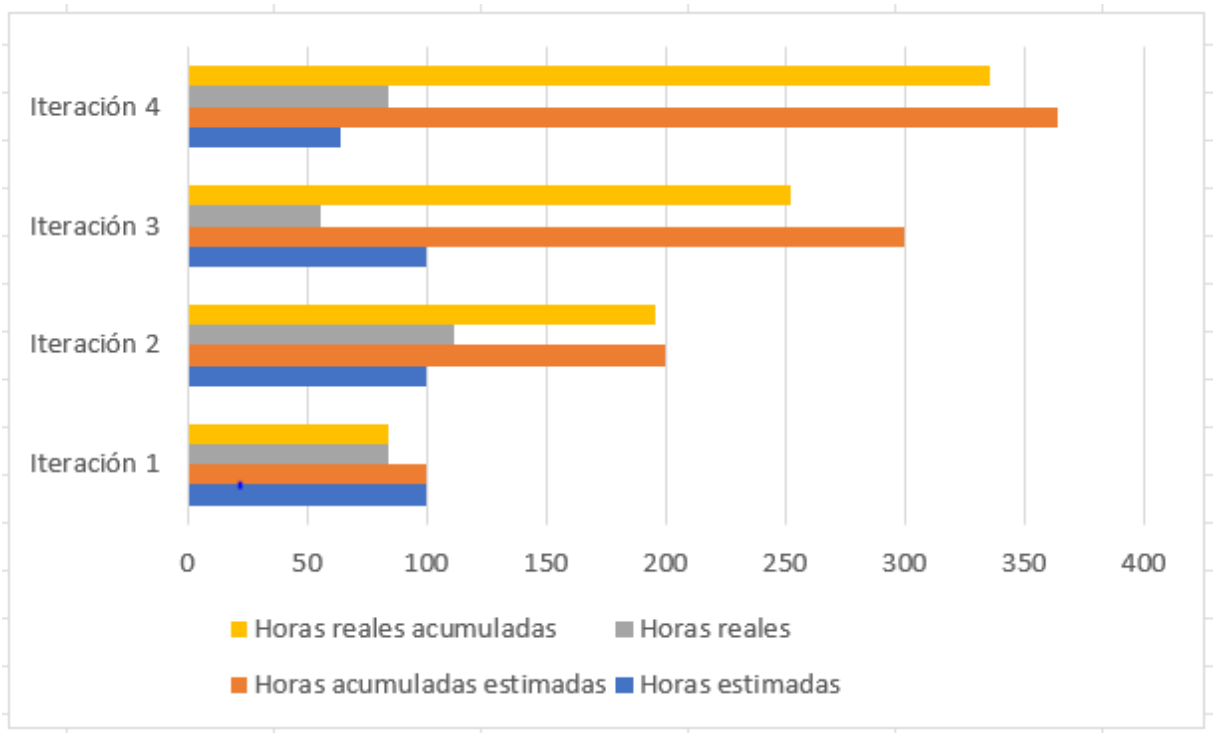

Figura 3.2: Comparación de las horas estimadas frente a las reales en cada iteración

3.2. Presupuesto económico

Para la realización del proyecto se han necesitado varios dispositivos de trabajo, algunos de los cuales ya se tenían y otros que no. Además, calcularé el coste que podría suponer un proyecto como este, basado en el salario medio de un desarrollador Android. A continuación, se detallan cada uno de los presupuestos que componen el trabajo.

3.2.1. Hardware y software

Hardware

En este apartado se incluyen los dispositivos que se han utilizado durante el desarrollo del TFG, al ser el único hardware necesario en este caso. Aunque ya se disponía tanto del ordenador como del móvil, se incluye aún así el precio de dichos dispositivos para dar una visión más realista en cuanto a todo lo que se necesita. El smartwatch utilizado (Samsung Galaxy Watch 4) fue necesario adquirirlo, y es el único que se tuvo que comprar ya que no se tenía ninguno. En cuanto al software, todos los programas que se han utilizado han sido gratuitos, por lo que no se ha gastado nada en este apartado.

Para calcular los costes de los dispositivos, se ha llevado a cabo el cálculo de las horas de uso de los dispositivos. Esto se ha hecho así ya que es más justo que considerar el coste total del dispositivo cuando solo se van a utilizar cuatro meses. Para ello, partimos del precio por el que fueron adquiridos los dispositivos, y calculamos el coste en función del tiempo medio de vida del dispositivo. En el caso del ordenador, su coste fue de 750 horas y tiene un tiempo medio de vida de cinco años. Haciendo una regla de tres sencilla, se obtiene que para las 363 horas de trabajo que se preveen en este TFG, el coste del ordenador para el proyecto es de $\frac{363}{43800} * 750 = 6{,}21$ euros. De manera análoga, se calcula este valor para el smartphone y el smartwatch. El smartphone tiene un tiempo medio de vida de 4 años y fue adquirido por 200 euros, mientras que el smartwatch tiene un tiempo de vida de 3 años y fue adquirido por 156 euros. La tabla 3.3 muestra los costes finales aplicando lo que se acaba de describir. En la tabla se muestra el coste del dispositivo cuando fue adquirido en la columna coste y el coste amortizado del dispositivo de acuerdo a este proyecto. Lo que interesa en este caso es el coste amortizado, y es el que se toma para el cálculo del presupuesto total, que está detallado en una sección posterior.

Dispositivo	Coste amortizado	Coste
Ordenador portátil	6.21 €	750 €
Smarphone Xiaomi	2.07 €	200 €
Smartwatch Samsung	2.15 €	156 €
Coste total	10.43 €	1106 €

Cuadro 3.3: Presupuesto hardware

3.2.2. Recursos humanos

En este apartado se considerará únicamente el salario de la persona que ha desarrollado el proyecto. Haciendo una búsqueda rápida en internet, se encuentra que el sueldo medio de un desarrollador Android en España se encuentra en torno a los 32000 - 33000 euros brutos al año [21]. Debido a que la jornada se ha realizado ha sido de unas 28 horas semanales, se supondrá que se trabaja a tiempo parcial y que se cobra la mitad que eso (aunque la media jornada suele ser en torno a 20 horas, se considera así por simplicidad). En conclusión, dado que el proyecto ha tenido una duración de tres meses, el coste de recursos humanos es 5300 euros aproximadamente. Esto se calcula dividiendo el sueldo anterior entre tres, ya que se trabaja cuatro meses, y entre dos porque se trabaja a tiempo parcial.

3.2.3. Presupuesto total

En este apartado se calcula el coste total estimado que ha tenido el proyecto. Se basa mayoritariamente en los gastos asociados al hardware, software y los recursos humanos, que ya han sido discutidos. Sin embargo, hay otra serie de gastos que también hay que considerar: los correspondientes al espacio de trabajo.

En el espacio de trabajo se incluyen los gastos que son comunes a la mayoría de proyectos. Se considerarán aquí la luz y el internet. Otro gasto que podría incluirse es el lugar de residencia, pero en este caso no se aplica al estar en mi residencia habitual durante la realización del proyecto.

Respecto a la la luz, su precio medio varía notablemente de un día a otro. Atendiendo a los datos de la web [22] , a día 25 de enero de 2023, el precio de la luz medio en España estaba a 0.16857 €/h. Multiplicándolo por las 363 horas estimadas para el proyecto, se obtiene un total de 61.2 euros aproximadamente. La tarifa de internet que tenemos nos cuesta aproximadamente 30 euros al mes, por lo que en total gastamos 120 euros aproximadamente. La tabla 3.4 resume todos los gastos del proyecto de acuerdo a todo lo visto en la sección.

Tipo de gasto	**Coste**
Hardware	10.43 €
Recursos humanos	5300 €
Luz	61.2 €
Internet	120 €
Coste total	**5491.63** €

Cuadro 3.4: Presupuesto total del proyecto

Parte II

Documentación técnica

Capítulo 4

Análisis

4.1. Requisitos

En esta sección se detallan todos los requisitos que han sido necesarios para este proyecto. Se clasificarán en tres grandes grupos: requisitos funcionales, no funcionales y de información. A su vez, se dividirán en cada una de las iteraciones en las que han sido necesarios cumplirlos, de acuerdo a la planificación inicial que se había previsto al inicio del TFG.

De manera breve, se definen a continuación cada uno de los tipos de requisitos que consideramos. Los requisitos funcionales son aquellos que describen lo que hace el sistema. Son los que el desarrollador tendrá que implementar para satisfacer los requisitos de usuario. Éstos no se consideran aparte y, por ello, se incluyen dentro de los funcionales. El como llevar a cabo lo descrito en los requisitos funcionales es lo que se especifica en los requisitos no funcionales. Éstos pueden ser de diversos tipos: rendimiento, usabilidad, seguridad, etc. No se considerará esta distinción y se englobarán todos bajo la etiqueta de requisitos no funcionales. Por último, se tienen los requisitos de información, que se encargan de describir los datos con los que operará el sistema.

Todos los requisitos vendrán resumidos en tablas, con un identificador, un nombre, y una breve descripción del requisito para entenderlo mejor.

4.1.1. Iteración 1:

En la primera iteración, el objetivo es realizar una sencilla aplicación que muestre los datos por pantalla, para luego poder hacerlo más complejo en iteraciones posteriores. Los requisitos funcionales que se deben cumplir son los que vienen en la tabla 4.1. Los requisitos no funcionales determinados para esta iteración son los que vienen en la tabla 4.2. Respecto a los requisitos de información, en esta iteración no se almacenan los datos, por lo que ya se considerarán en otra iteración posterior.

ID	Nombre	Descripción
RF_01	Inicio monitorización	El sistema debería de permitir que el usuario pueda empezar a monitorizar tanto la ubicación como los sensores.
RF_02	Pausa monitorización	El sistema debería de permitir que el usuario pueda pausar la monitorización tanto de la ubicación como los sensores.
RF_03	Mostrar ubicación	El sistema debería de mostrar la ubicación actual, así como monitorizarla periódicamente.
RF_04	Mostrar aceleración	El sistema debería de mostrar la aceleración actual, así como monitorizarla periódicamente.
RF_05	Mostrar frecuencia	El sistema debería de mostrar la frecuencia actual, así como monitorizarla periódicamente.

Cuadro 4.1: Requisitos funcionales de la iteración 1

ID	Nombre	Descripción
RNF_01	Entorno desarrollo	El sistema debe desarrollarse para el sistema operativo Wear OS, con el lenguaje Java.
RNF_02	Periodo de monitorización	El sistema debe de monitorizar la ubicación cada 5 segundos.
RNF_03	Ahorro de batería	El sistema debe de monitorizar la frecuencia cardiaca y los acelerómetros en el modo que priorice el bajo consumo de batería.

Cuadro 4.2: Requisitos no funcionales de la iteración 1

4.1.2. Iteración 2

La segunda iteración tiene como objetivo integrar los acelerómetros y la ubicación para crear una aplicación que sea capaz de detectar el estado de reposo, y en ese caso, monitorizar la frecuencia cardiaca. De nuevo no se tienen en esta iteración requisitos de información al no almacenar la información todavía. Además, los requisitos que coincidan con los de la anterior iteración no se pondrán de nuevo. Esto se aplica también a las siguientes iteraciones. Las tablas 4.3 y 4.4 muestran los requisitos funcionales y no funcionales de esta iteración respectivamente.

ID	Nombre	Descripción
RF_06	Comunicación interna	El sistema debe ser capaz de comunicar a los acelerómetros la ubicación GPS.
RF_07	Empezar monitorización	El sistema debe de empezar la monitorización de la frecuencia cardiaca una vez se ha detectado un estado de reposo.
RF_08	Parar monitorización	El sistema debe de parar la monitorización de la frecuencia cardiaca cuando el usuario no esté en reposo, en caso de que se estuviese monitorizando.
RF_09	Mostrar estado de reposo	El sistema debería de informar mientras se monitoriza si el usuario se encuentra o no en reposo.
RF_10	Mostrar frecuencia	El sistema debería de mostrar la frecuencia actual, en caso de estar en reposo.

Cuadro 4.3: Requisitos funcionales de la iteración 2

ID	Nombre	Descripción
RNF_04	Condiciones inicio monitorización	El sistema debe empezar a monitorizar siempre y cuando los acelerómetros estén por debajo de un límite (3 m/s) y hayan pasado más de 30 minutos desde la última medida.
RNF_05	Condiciones pausa monitorización	El sistema debe parar de monitorizar cuando los acelerómetros detecten un valor mayor al límite mencionados en el RNF_04.
RNF_06	Periodo de monitorización	El sistema debe de monitorizar la frecuencia en intervalos de 30 minutos durante 30 segundos, es decir, se monitoriza durante 30 segundos, se para 30 minutos, se vuelve a monitorizar, y así sucesivamente.

Cuadro 4.4: Requisitos no funcionales de la iteración 2

4.1.3. Iteración 3

En esta iteración, lo que se pretende conseguir es transmitir y almacenar las frecuencias que han sido previamente monitorizadas en estado de reposo. Por ello, esta la única iteración que tiene requisitos de información. Los requisitos funcionales, no funcionales y de información se encuentran respectivamente en las tablas 4.5, 4.6 y 4.7.

ID	Nombre	Descripción
RF_11	Transmitir frecuencias	El sistema debería de transmitir al teléfono las frecuencias que se monitorizan.
RF_12	Sincronización	El sistema debería de ser capaz de almacenar las frecuencias cuando no estén conectados el móvil y el reloj.

Cuadro 4.5: Requisitos funcionales de la iteración 3

ID	Nombre	Descripción
RNF_07	Número de frecuencias	El sistema debe de mandar y almacenar todas las frecuencias monitorizadas durante 30 segundos si el usuario permanece en reposo durante todo ese tiempo
RNF_08	Periodo de sincronización	El sistema debe de actualizar de manera inmediata las frecuencias que se monitorizaron mientras no estaban conectados, una vez el usuario inicie la app en el móvil y esté conectado de nuevo al reloj.

Cuadro 4.6: Requisitos no funcionales de la iteración 3

ID	Nombre	Descripción
RI_01	Frecuencia	Se almacenarán las frecuencias medidas.
RI_02	Fecha	Se almacenarán las fechas de las frecuencias medidas.
RI_03	ID smartwatch	Se almacenará el ID único del smartwatch en el que fue medida la frecuencia.
RI_04	Latitud	Se almacenará (si está disponible) la latitud donde fue medida la frecuencia.
RI_05	Longitud	Se almacenará (si está disponible) la longitud donde fue medida la frecuencia.

Cuadro 4.7: Requisitos de información de la iteración 3

4.1.4. Iteración 4

La última iteración se centra en la interfaz de usuario, mediante la visualización de diferentes gráficos de las frecuencias monitorizadas, así como otras funcionalidades a mayores

como el tutorial de uso y la interfaz del reloj. Los requisitos funcionales y no funcionales se encuentran respectivamente en las tablas 4.8 y 4.9.

ID	Nombre	Descripción
RF_13	Ver última frecuencia	La app del móvil debe de mostrar la última frecuencia monitorizada.
RF_14	Ver historial semana	La app del móvil debe de mostrar un gráfico de las frecuencias medidas durante la última semana.
RF_15	Ver historial día	La app del móvil debe de mostrar un gráfico de las frecuencias medidas durante el día que elija el usuario.
RF_16	Ver historial mes	La app del móvil debe de mostrar un gráfico de las frecuencias medidas durante el mes que elija el usuario.
RF_17	Ver historial año	La app del móvil debe de mostrar un gráfico de las frecuencias medidas durante el año que elija el usuario.
RF_18	Ver tutorial	La app del móvil debe de tener un tutorial de manejo de la app tanto del móvil como la del reloj.
RF_19	Ver frecuencia actual	La app del reloj debe de mostrar la frecuencia actual que se monitorice.

Cuadro 4.8: Requisitos funcionales de la iteración 4

ID	Nombre	Descripción
RNF_09	Frecuencias en el historial	Las frecuencias mostradas en las gráficas deben de ser un promedio de las frecuencias durante una hora, día o mes determinado, dependiendo de la gráfica que consideremos.
RNF_10	Cálculo de la última frecuencia	La última frecuencia debe mostrarse como un promedio de las frecuencias medidas durante el último periodo de monitorización.

Cuadro 4.9: Requisitos no funcionales de la iteración 4

4.2. Historias de usuario

En esta sección se muestran las diferentes historias de usuario presentes en el proyecto. Para ello, se incluye una historia de usuario por cada caso de uso. Por ello, también incluimos el diagrama de casos de uso. Trataremos tanto el caso de la app del teléfono como la del smartwatch.

4.2.1. Smartwatch

En el smartwatch tenemos únicamente los casos de uso relativos al inicio y parada de la monitorización. Se puede ver el diagrama de casos de uso en la figura 4.1.

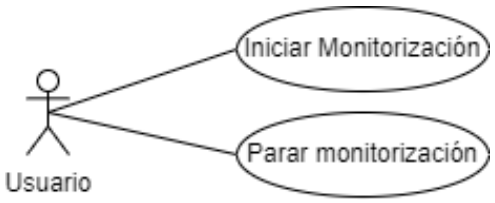

Figura 4.1: Diagrama de casos de uso de la app del smartwatch

Respecto a las historias de usuario se tienen las siguientes:

ID	HU_01
Nombre	Iniciar monitorización
Prioridad	Alta
Riesgo	Alto
Descripción	Como usuario quiero poder iniciar la monitorización de la frecuencia cardiaca en estado de reposo siempre que lo desee.
Validación	Quiero iniciar la monitorización en cualquier momento

Cuadro 4.10: Historia de Usuario iniciar la monitorización

ID	HU_02
Nombre	Parar la monitorización
Prioridad	Alta
Riesgo	Alto
Descripción	Como usuario quiero poder parar la monitorización de la frecuencia cardiaca en estado de reposo siempre que lo desee. Precondición: el usuario debe haber iniciado la monitorización.
Validación	Quiero parar la monitorización en cualquier momento

Cuadro 4.11: Historia de Usuario parar la monitorización

4.2.2. Teléfono móvil

En el teléfono móvil es donde se encuentran la mayor parte de las funcionalidades. Por ello se tiene un mayor número de casos de uso. Estos se detallan en la figura 4.2.

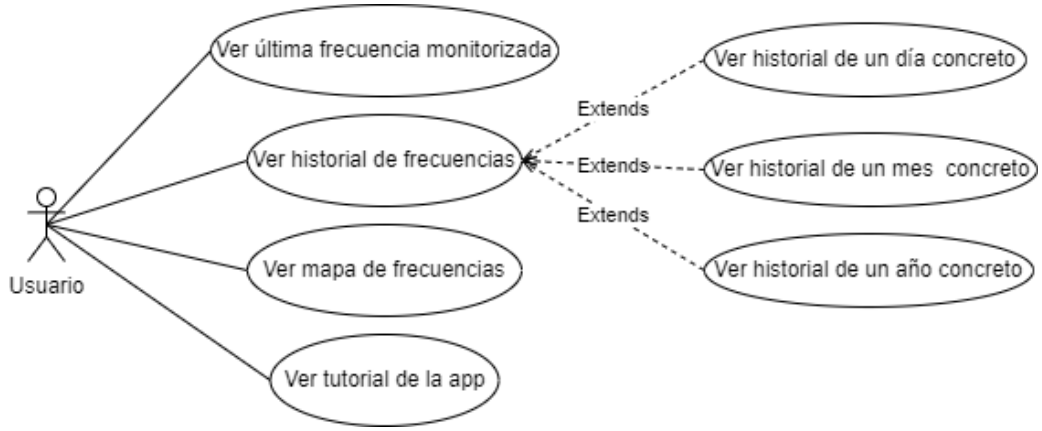

Figura 4.2: Diagrama de casos de uso de la app del móvil

Las historias de usuario correspondientes a dichos casos de uso son las siguientes:

ID	HU_03
Nombre	Ver última frecuencia
Prioridad	Media
Riesgo	Bajo
Descripción	Como usuario quiero poder ver la última frecuencia que fue monitorizada en estado de reposo.
Validación	- Quiero ver el valor de la última frecuencia que fue monitorizada - Quiero ver la fecha en la que fue monitorizada la frecuencia - Quiero ver (si está disponible) la ubicación de la frecuencia monitorizada

Cuadro 4.12: Historia de Usuario ver última frecuencia

ID	HU_04
Nombre	Ver historial
Prioridad	Alta
Riesgo	Bajo
Descripción	Como usuario quiero poder ver las frecuencias medidas en la última día, semana, mes y año
Validación	- Quiero ver las frecuencias medidas en la última semana - Quiero ver las frecuencias medidas en el último día - Quiero ver las frecuencias medidas en el último mes - Quiero ver las frecuencias medidas en el último año

Cuadro 4.13: Historia de Usuario ver historial

ID	HU_05
Nombre	Ver historial de un día concreto
Prioridad	Media
Riesgo	Bajo
Descripción	Como usuario quiero poder ver las frecuencias medidas en un día concreto que desee
Validación	- Quiero poder elegir el día - Quiero poder ver las frecuencias del día elegido

Cuadro 4.14: Historia de Usuario ver historial de un día concreto

ID	HU_06
Nombre	Ver historial de un mes concreto
Prioridad	Media
Riesgo	Bajo
Descripción	Como usuario quiero poder ver las frecuencias medidas en un mes concreto que desee
Validación	- Quiero poder elegir el mes - Quiero poder ver las frecuencias del mes elegido

Cuadro 4.15: Historia de Usuario ver historial de un mes concreto

ID	HU_07
Nombre	Ver historial de un año concreto
Prioridad	Media
Riesgo	Bajo
Descripción	Como usuario quiero poder ver las frecuencias medidas en un año concreto que desee
Validación	- Quiero poder elegir el año - Quiero poder ver las frecuencias del año elegido

Cuadro 4.16: Historia de Usuario ver historial de un año concreto

ID	HU_08
Nombre	Ver mapa de frecuencias
Prioridad	Media
Riesgo	Medio
Descripción	Como usuario quiero poder ver un mapa con las ubicaciones donde fueron medidas las frecuencias
Validación	- Quiero ver un mapa con mi ubicación actual - Quiero ver un marcador en cada ubicación donde se monitorizaron las frecuencias - Quiero que en el marcador se indique el valor de la frecuencia medida

Cuadro 4.17: Historia de Usuario ver mapa de frecuencias

ID	HU_09
Nombre	Ver tutorial
Prioridad	Media
Riesgo	Bajo
Descripción	Como usuario quiero ver un tutorial que indique como utilizar tanto la app del reloj como la del móvil.
Validación	- Quiero ver un tutorial de la app del reloj - Quiero ver un tutorial de la app del móvil

Cuadro 4.18: Historia de Usuario ver tutorial

4.3. Modelo de dominio

A continuación, se presenta el modelo de dominio del problema. Se trata de un modelo conceptual de todos los temas relacionados en el proyecto. En él se describen las distintas entidades, sus atributos, papeles y relaciones, además de las restricciones que rigen el dominio del problema. Se puede ver en la figura 4.3.

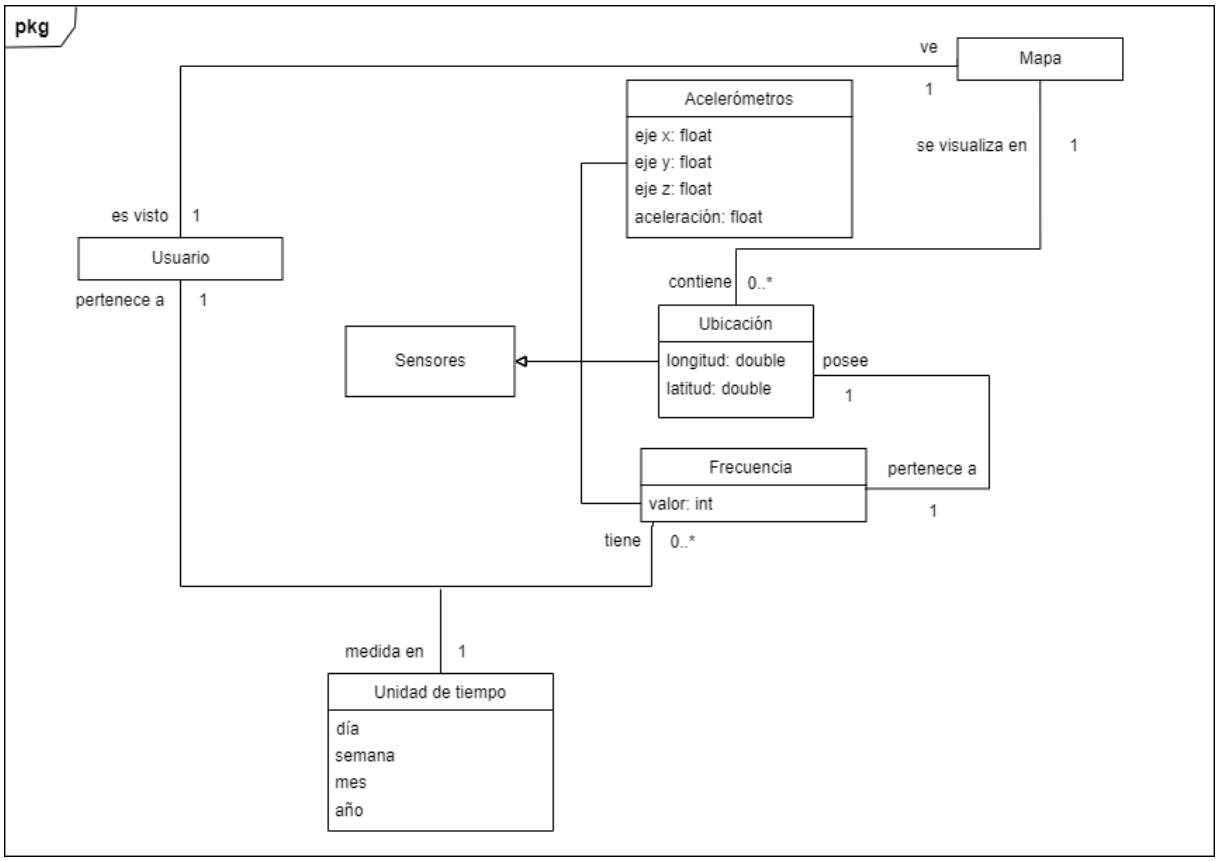

Figura 4.3: Modelo de dominio del problema

El modelo se interpreta como sigue. El **Usuario** inicia la monitorización del estado de **Reposo**, el cual se determina por medio de los **Acelerómetros**. Éstos, junto la **Frecuencia**, son los **Sensores** presentes en la aplicación. Una vez detectado un estado de reposo, se mide la frecuencia, y se almacena junto con la **Ubicación** donde fue medida. Estas ubicaciones pueden ser visualizadas por el usuario en un **Mapa**. A mayores, el usuario puede ver un **Historial de frecuencias** de las que ha medido, y un **Tutorial** de uso de las aplicaciones del reloj y el móvil.

Respecto a los atributos que se encuentran en cada concepto, se tiene lo siguiente. Los acelerómetros miden la aceleración, en función de los valores de los ejes x,y,z. La aceleración es lo que determina el estado de reposo. La frecuencia almacena el valor como un entero, mientras que la ubicación posee la latitud y longitud correspondientes almacenados como double. Por último, en el mapa se muestra tanto la ubicación actual como cada una de las frecuencias medidas en un marcador. Ambos son objetos de la clase location.

4.4. Diagramas de actividad

En esta sección se exponen los diferentes diagramas de actividad resultantes del proyecto. Éstos son los correspondientes a cada una de las historias de usuario que se identificaron anteriormente. En resumen, un diagrama de actividades es un diagrama de flujo que muestra actividades ejecutadas por un sistema. En este caso, se mostrará el flujo de cada una de las historias de usuario, y acompañados todos ellos de una breve explicación del flujo del diagrama.

HU_01: Iniciar monitorización

Esta historia de usuario corresponde a la funcionalidad con el que el usuario es capaz de iniciar la monitorización de la frecuencia cardiaca en estado de reposo. En el diagrama se muestra el flujo de la aplicación para una monitorización completa.

Para ello, una vez el ususario inicia la aplicación del reloj y pulsa en iniciar monitorización, lo primero que ocurre es la comprobación de la última medida que tuvo éxito. En caso de haber pasado al menos media hora desde la última medida, se inicia la monitorización de los acelerómetros y la ubicación (si no se indican los minutos restantes para monitorizar de nuevo).

Cada vez que se recibe una nueva aceleración, se comprueba que esté por debajo de un determinado valor, en cuyo caso se considera que el usuario está en reposo. Para empezar a monitorizar la frecuencia cardiaca, el usuario debe permanecer en reposo durante 10 segundos seguidos, de lo contrario no se monitoriza. Cuando esto se cumple, se monitoriza la frecuencia cardiaca durante 30 segundos. De nuevo, si durante esos 30 segundos se detecta que el usuario no está en reposo se resetea y vuelve a empezar todo (acumular los 10 segundos de nuevo). En caso de monitorizarse las medidas correctamente durante los 30 segundos, se mandan al teléfono y se almacenan.

Cuando no haya conexión por bluetooth con el teléfono o de internet, las frecuencias se almacenarán localmente en el reloj o en servidores de Google (se explica más esto más detalladamente en el capítulo de implementación). Una vez haya conexión de nuevo se mandan automáticamente al teléfono. Todo esto lo gestiona la API que se ha utilizado en el proyecto. Por último, cuando se ha realizado una medida de una tanda correctamente, se actualiza el valor de la última medida. El diagrama se muestra en la figura 4.4.

HU_02: Parar la monitorización

Suponiendo que la monitorización está iniciada, el usuario puede pararla. El diagrama de actividad resultante se puede ver en la figura 4.5. Al parar de monitorizar se para de medir la frecuencia cardiaca, los acelerómetros y la ubicación.

HU_03: Ver última frecuencia

Esta funcionalidad es la que por defecto se muestra al iniciar el usuario sin que el usuario necesite hacer nada más. Se muestra el valor de la frecuencia, junto con la fecha en la que fue monitorizada (el día), y la ubicación donde fue monitorizada si existe. Puede ser que no exista por haber medido la frecuencia cuando no había señal gps, como por ejemplo el caso de interiores. El diagrama se muestra en la figura 4.6.

HU_04 - HU_07: Ver historial

Las historias de usuario comprendidas entre la 4 y la 7, ambas inclusive, se agrupan todas bajo la funcionalidad común de ver el historial de las frecuencias medidas. Además, las historias de usuario 5, 6 y 7 son completamente análogas cambiando la unidad de tiempo que se quiere ver (día, mes o año). Por ello, incluimos simplemente dos diagramas de actividad, los correspondientes a la historia de usuario 4 y al resto de las historias, que se agrupan en uno solo. Esto se ve en la figura 4.7.

HU_08: Ver mapa de frecuencias

Cada medida que es monitorizada, se almacena junto con la ubicación en la que fue monitorizada, si ésta está disponible. El objetivo es mostrar las ubicaciones en un mapa junto con el valor medido. Esto es la funcionalidad de esta historia de usuario. Cuando el usuario inicia la app y selecciona el mapa, se le carga un mapa con la ubicación actual donde está, si está disponible. Además, se le muestran marcadores indicando el valor de la frecuencia medida. El diagrama de actividad se muestra en la figura 4.8

HU_09: Ver tutorial

Por último, tenemos la historia de usuario correspondiente al breve tutorial que está disponible en la aplicación del móvil. El diagrama de actividad es muy sencillo y se puede ver en la figura 4.9.

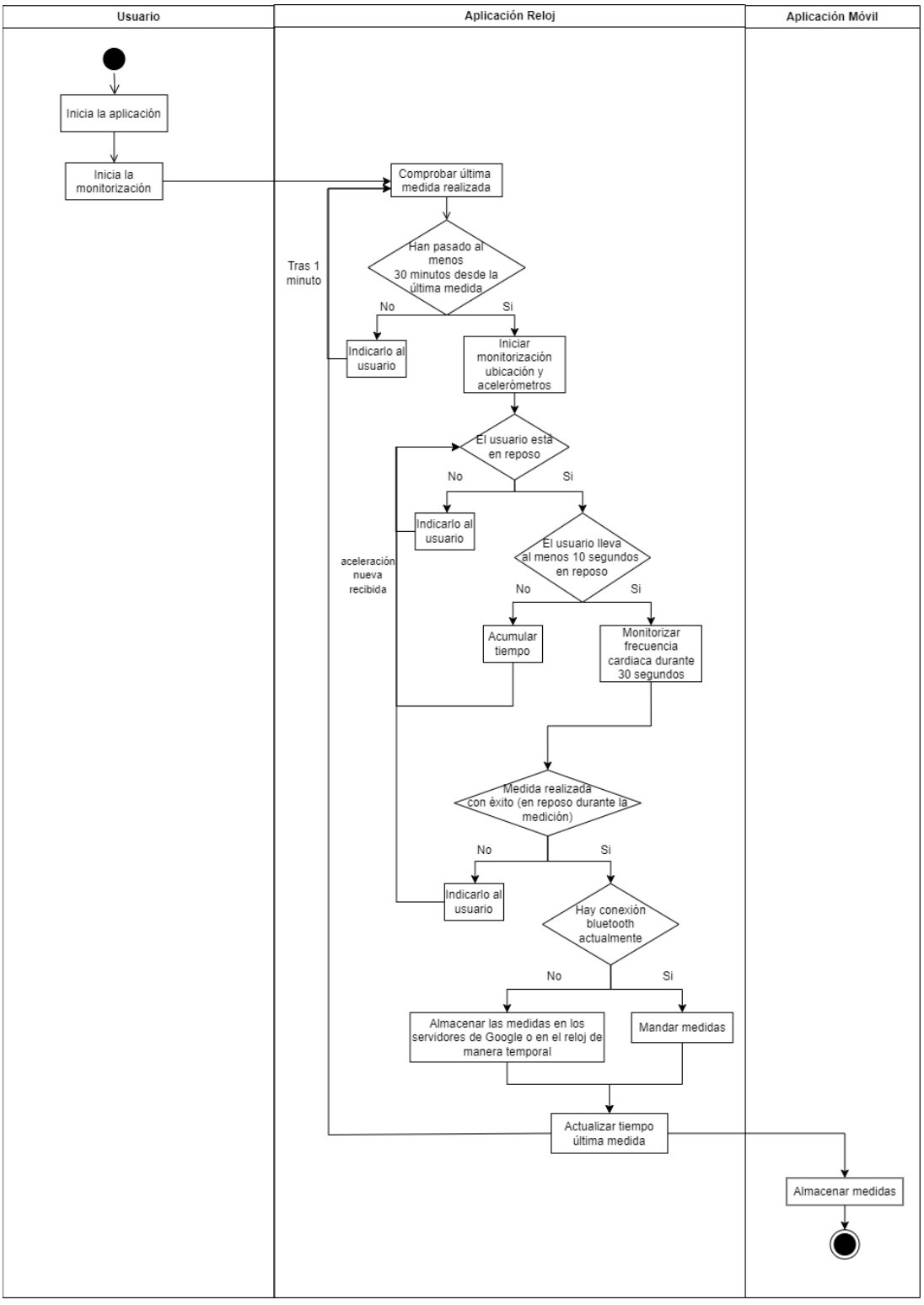

Figura 4.4: Diagrama de actividad de HU_01

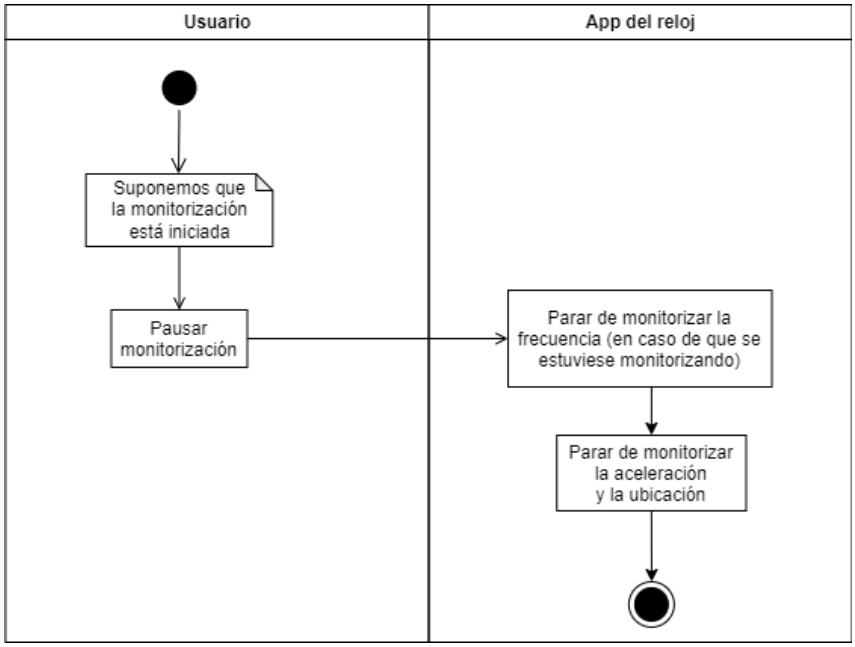

Figura 4.5: Diagrama de actividad de HU_02

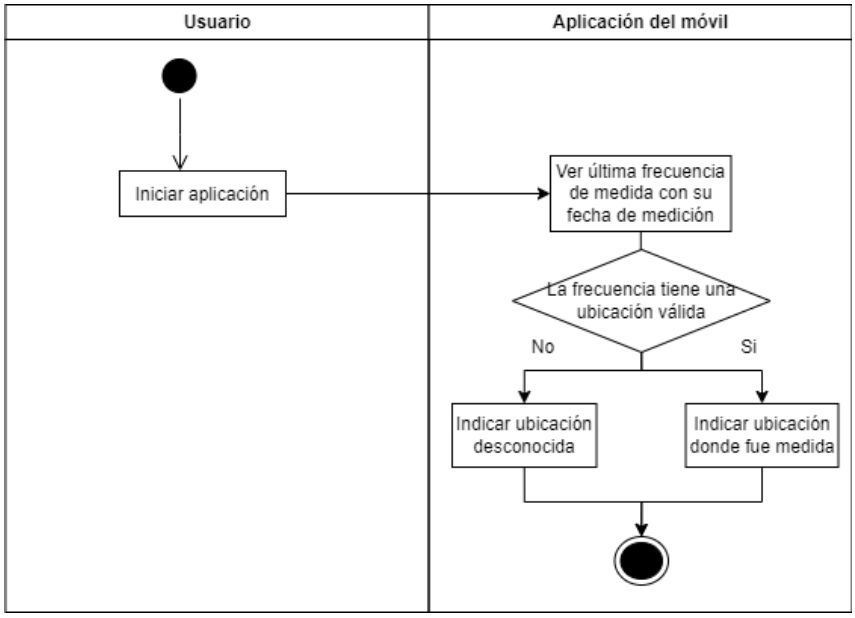

Figura 4.6: Diagrama de actividad de HU_03

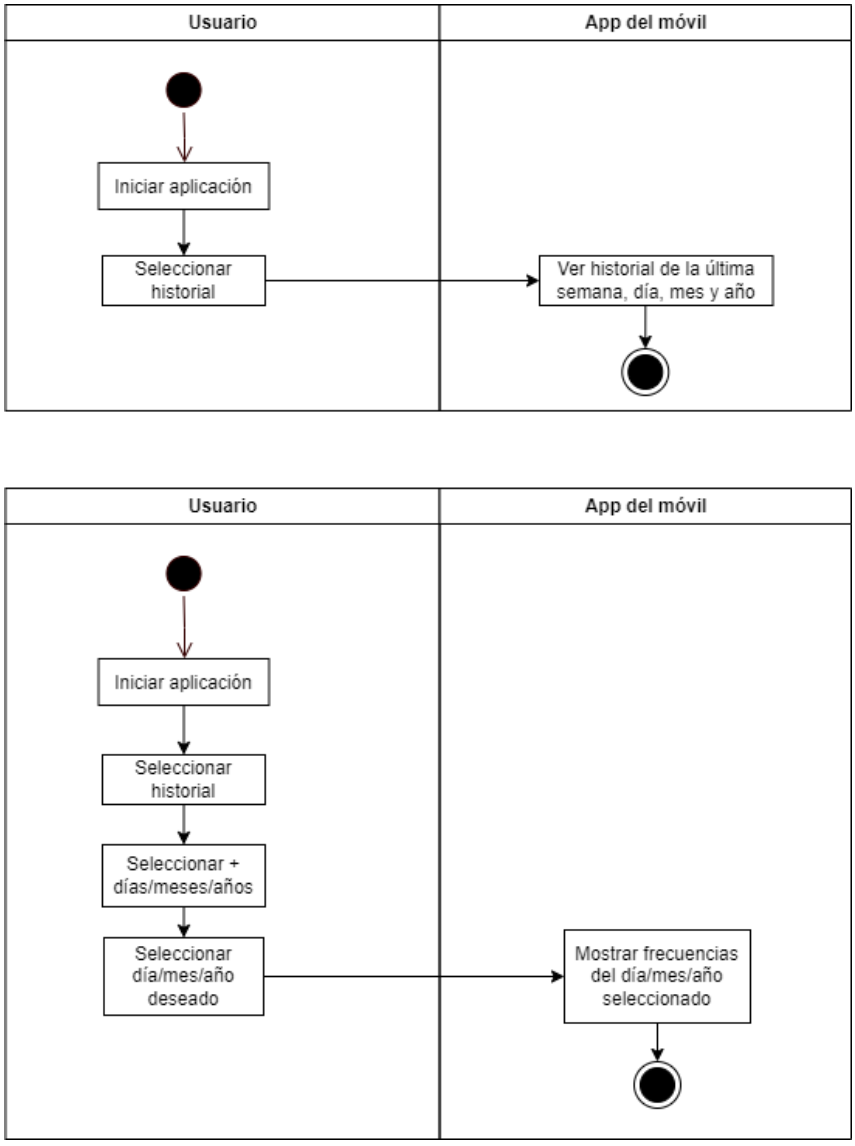

Figura 4.7: Diagrama de actividad de HU_04 y HU_05 - HU_07

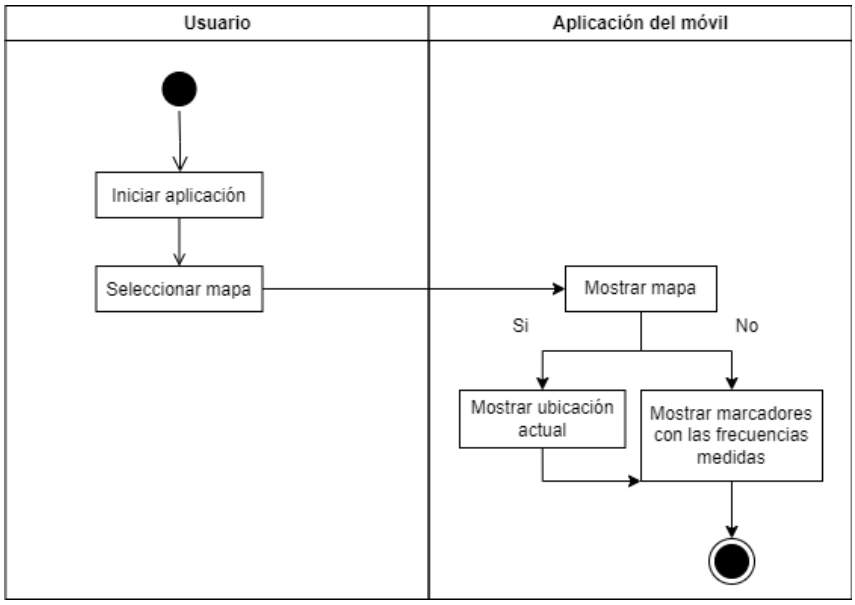

Figura 4.8: Diagrama de actividad de HU_08

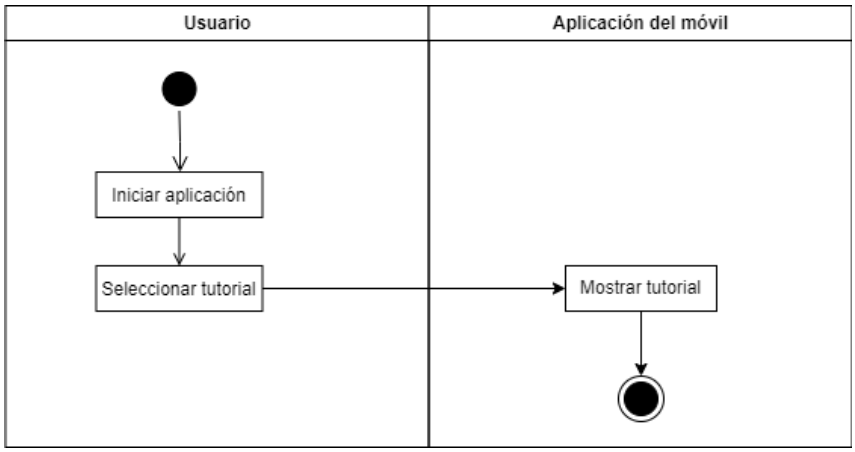

Figura 4.9: Diagrama de actividad de HU_09

Capítulo 5

Diseño

A lo largo de este capítulo se expondrán distintos diagramas del proyecto, explicando cada uno de ellos. También se explicará la arquitectura seguida en cada una de las apps, así como los patrones de diseño utilizados. Por último, se mostrará la estructura del proyecto final, comentando como se organiza el código y lo que contienen las distintas carpetas del proyecto de Android Studio.

5.1. Diseño de datos

En la figura 5.1 se puede ver la única entidad presente en la base de datos del proyecto. Para los propósitos que se pretendían lograr no era necesario considerar una base de datos más compleja que lo que se ha realizado. Se ve en ella que se almacena el valor de la frecuencia cardiaca como un número entero, la fecha en la que fue medida, el ID del reloj (necesario por si se conecta más de un reloj al teléfono), y la ubicación de la medida (latitud y longitud). La clave primaria de la tabla es un ID entero.

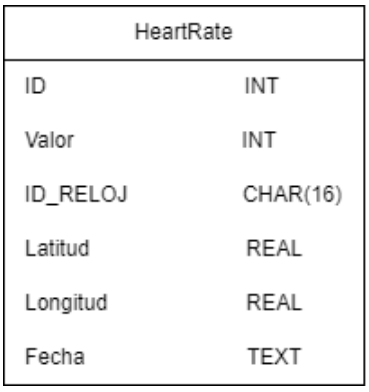

Figura 5.1: Diagrama Entidad-Relación del proyecto

5.2. Arquitectura

Esta sección expone cual ha sido la arquitectura seguida tanto en la app del reloj como la del teléfono móvil. También se comentarán cuales son los patrones de diseño principales que se han utilizado en el proyecto.

5.2.1. App del reloj

La arquitectura seguida en el smartwatch es la recomendada por la documentación, y está basada en 3 capas: la capa de usuario, la capa de dominio y la capa de datos. Separar las capas tiene varias ventajas, entre las que se encuentran una escalabilidad más fácil, separación de componentes de acuerdo a su funcionalidad, una mayor flexibilidad y modularización del código, posibilidad de reutilización del código y una mayor seguridad al separar la capa de usuario de la de datos. A continuación se explica brevemente como se ha implementado en el proyecto cada una de las capas:

- **Capa de usuario:** en Android, esta capa corresponde a las diferentes activities o fragments de la aplicación. En este caso se tiene una activity, *MainActivity*, con un contenedor dentro de ella en la cual se renderizan cada uno de los tres fragments correspondientes a la pantalla de inicio, la de monitorización y la de pausa/stop.

- **Capa de dominio:** esta capa es opcional, pero permite una mayor abstracción y flexibilidad en la aplicación. En el caso del proyecto, es muy importante separar adecuadamente las funcionalidades, pues tenemos varios hilos y ciclos de vida a la vez funcionando, y juntarlos todos o hacer una mala arquitectura puede dar lugar a múltiples fallos en la ejecución. En el proyecto, el servicio y los hilos correspondientes a los sensores y ubicación pertenecen a esta capa.

- **Capa de datos:** por último tenemos la capa de datos, que contiene a las clases que sirven como modelo de datos, así como clases dedicadas al almacenamiento en una base de datos cuando sea necesario. En la app del reloj no se tiene una base de datos, pero si tenemos clases modelo. Estas son las que se encuentran en el paquete común de la app, *utilities*. A través del servicio y los hilos se accede a dichas clases y se recuperan o actualizan sus atributos.

El esquema de esta arquitectura se puede ver en la figura 5.2. Se puede ver como cada capa accede a la inmediatamente inferior a ella.

5.2.2. App del móvil

En cuanto a la app del móvil, la arquitectura es exactamente la misma que en Wear OS, con una única diferencia, en este caso no se usa la capa de dominio. Como se había comentado, se trata de una capa opcional que es recomendable en la mayoría de casos. Sin embargo, debido a que no es una aplicación muy grande, y que no existen los problemas

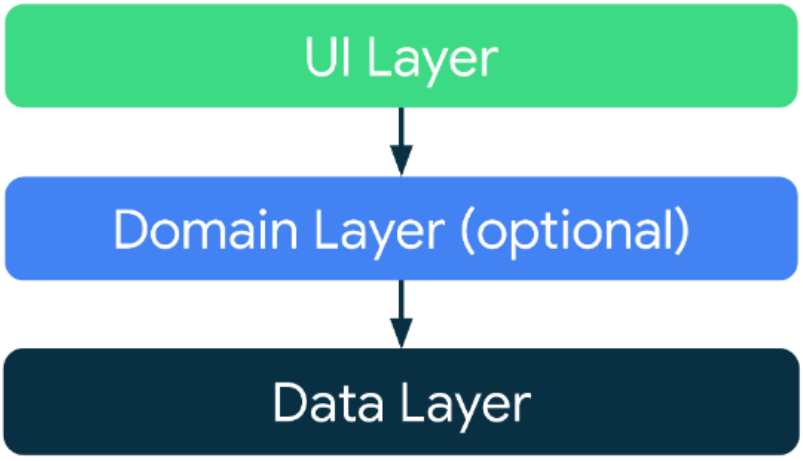

Figura 5.2: Arquitectura en 3 capas de una aplicación Android [11]

de sincronización de hilos y de ciclos de vida que había en el reloj, no es estrictamente necesario implementarla. No obstante, se podría una futura mejora la organización más óptima del código.

A continuación se resume lo que se incluye en cada una de las dos capas presentes en la aplicación:

- **Capa de usuario:** se tiene de nuevo una actividad, *MainActivity*, con un contenedor y una barra de navegación inferior. A través de dicha barra se navega entre los distintos fragments de la aplicación.
- **Capa de datos:** aquí se tiene tanto el paquete *utilities*, que ya fue comentado anteriormente, como la clase correspondiente al almacenamiento de los datos. Se usa el almacenamiento en una base de datos local, usando SQLite.

5.3. Patrones de diseño

Los patrones de diseño son soluciones generales y reutilizables a problemas que se encuentran con relativa frecuencia en la programación. Esta sección se dedica a resumir cuales han sido los principales patrones de diseño usados en el proyecto. En general, los patrones de diseño se dividen en las siguientes categorías:

- **Creacionales:** se refieren a la forma de crear los objetos. Este patrón de diseño se utiliza cuando se tiene que tomar una decisión en el momento en el que se va a instanciar una clase.
- **Estructurales:** se basa en como los objetos y clases se componen y relacionan para crear estructuras complejas.

- **De comportamiento:** son aquellos que tratan con algoritmos y/o asignación de responsabilidades entre los diferentes objetos.

5.3.1. Singleton

Singleton es un patrón de diseño creacional que garantiza que tan solo exista un objeto de su tipo y proporciona un único punto de acceso a él para cualquier otro código.

Este patrón ha sido utilizado en la aplicación del reloj, en concreto instanciando una única vez la clase *SharedData*. Ésta tiene distintos atributos sobre el flujo del programa, como por ejemplo un indicador de si se ha iniciado la aplicación o si se está monitorizando la frecuencia. Para ello, se instancia una única vez el objeto y se accede a él cuando se necesita mediante el método getInstance(), que devuelve siempre el mismo objeto, por lo que desde cualquier parte del código se accede a los mismos atributos del objeto.

5.3.2. Observer

El patrón observer es un patrón de diseño de comportamiento en el que un objeto, llamado *observable*, mantiene una lista de objetos dependientes, llamados *observers*, y notifica automáticamente a estos observers cuando el *observable* experimenta algún cambio de estado.

En la aplicación del reloj, los sensores (en este caso, los acelerómetros y la frecuencia cardiaca) implementan este patrón implícitamente. El objeto "SensorManager" actúa como el observable, y mantiene una lista de objetos dependientes, los "SensorListeners" u observers, que se registran para recibir actualizaciones de los sensores.

Cuando un sensor en el dispositivo cambia su estado, el SensorManager notifica automáticamente a todos los SensorListeners registrados para ese sensor, invocando su método "onSensorChanged". En este método, los SensorListeners pueden leer los nuevos datos del sensor y actualizar la aplicación según sea necesario.

Por lo tanto, en este caso, el patrón observador se utiliza para permitir que la aplicación Android reciba actualizaciones de los sensores de manera asincrónica, sin tener que estar constantemente verificando el estado de los sensores en un bucle de ejecución. En su lugar, el SensorManager notifica automáticamente a los oyentes cuando hay un cambio de estado en el sensor. Una explicación más detallada de la API de los sensores será explicada en un capítulo posterior. La figura 5.3 muestra un esquema del funcionamiento de este patrón.

5.3.3. Factory

Factory es un patrón de diseño creacional que se usa cuando se tiene una superclase con múltiples subclases y, según la entrada, se necesita devolver una de las subclases. Este patrón elimina la responsabilidad de la creación de instancias de una clase desde el programa cliente a la clase factory.

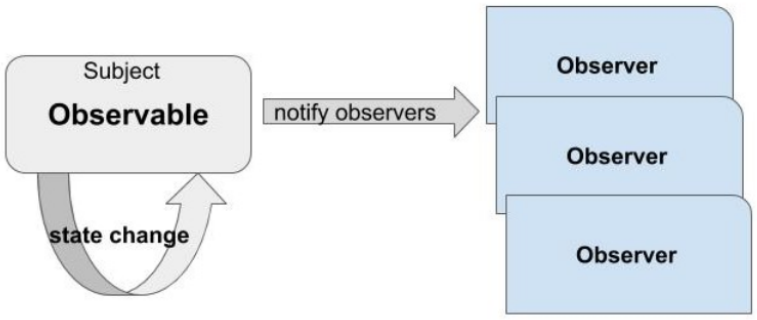

Figura 5.3: Esquema del patrón observer [19]

En el proyecto, este patrón lo utiliza internamente la API LocationManager. En concreto, para la gestión de los providers. LocationProvider es una clase abstracta que define una interfaz común para todos los providers de la ubicación. Los providers de la ubicación son clases que dan acceso a datos de ubicación de diferentes fuentes, como GPS o Wi-Fi.

La clase LocationManager es responsable de crear y administrar instancias de la subclase LocationProvider adecuada según los criterios especificados, como la precisión y el ahorro de batería. Lo hace usando el patrón factory.

5.4. Diagramas de paquetes

Los diagramas de paquetes representan las dependencias entre los paquetes que componen un proyecto. En este caso, se tiene un diagrama de paquetes para la aplicación del móvil y otro para la del reloj. Ambos se muestran en las figuras 5.4 y 5.5 respectivamente.

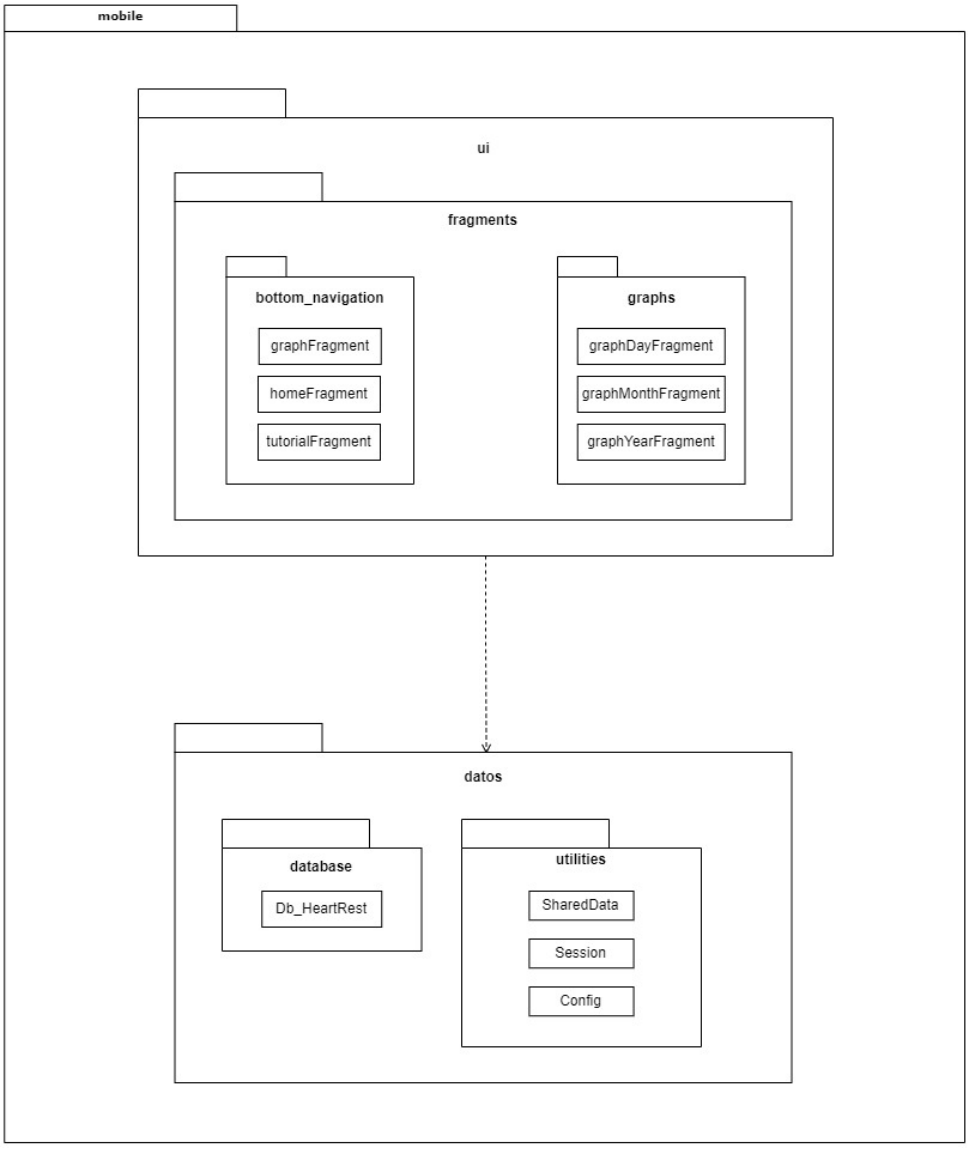

Figura 5.4: Diagrama de paquetes de la aplicación del móvil

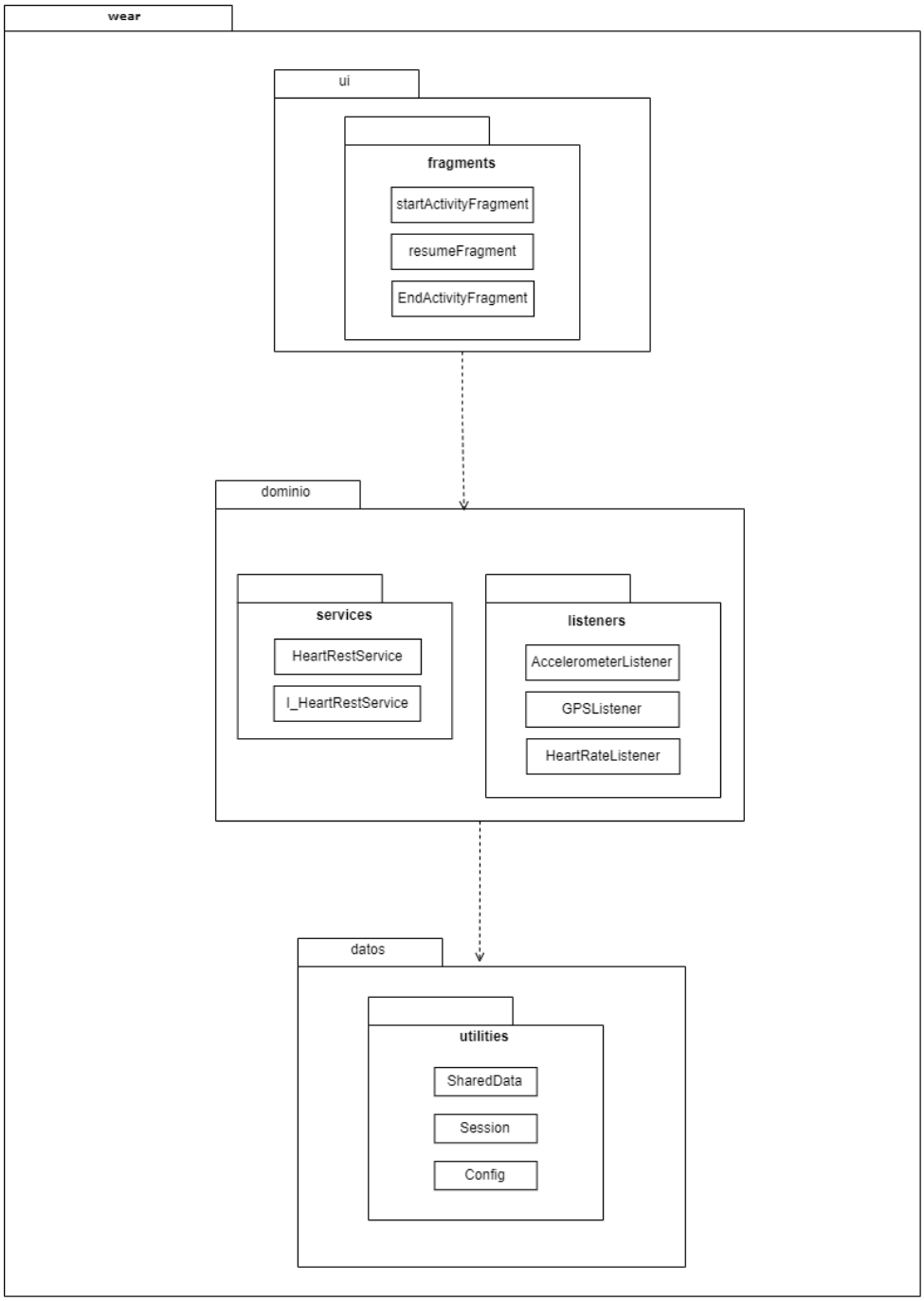

Figura 5.5: Diagrama de paquetes de la aplicación del reloj

En ellos se ve claramente la arquitectura en 3 capas (2 en el caso del teléfono), que se explicaró en una sección anterior. Dentro de cada una de las capas se tienen los distintos paquetes, y éstos a su vez contienen las clases de la aplicación.

En la aplicación del móvil se tiene el paquete *fragments*, que contiene todo lo relativo

a la interfaz de usuario. Dentro de él, se tienen a su vez dos paquetes: *bottom_ navigation* y *graphs*. El primero contiene los fragmentos correspondientes a la barra de navegación. El segundo contiene los fragmentos de las gráficas que muestran las frecuencias en un día, mes o año concreto. Respecto a la capa de datos, en ella se incluye la clase con la lógica de SQLite, dentro del paquete *database*, y las clases comunes a ambas apps dentro del paquete *utilities*.

En la aplicación del móvil, se tiene de nuevo en *fragments* las clases con la lógica de la interfaz de usuario. En la capa de dominio se tiene la lógica de la aplicación. Contiene las clases que monitorizan la frecuencia, los acelerómetros y ubicación bajo el paquete *listeners* y las clases del servicio bajo el paquete *services*. Por último, en este caso en la capa de datos se tiene el paquete común *utilities* que también está en la app del móvil.

5.5. Estructura del proyecto

Esta sección tiene como objetivo mostrar como han quedado finalmente distribuidos los componentes de la aplicación. Se explicará cada uno de las estructuras de ambas apps así como el paquete que comparten ambas. En la figura 5.6 se puede ver la estructura del proyecto global, en la que se ven 3 subproyectos que se explicarán en las siguientes subsecciones. Además, se pueden observar los ficheros gradle del proyecto y otros ficheros de configuración.

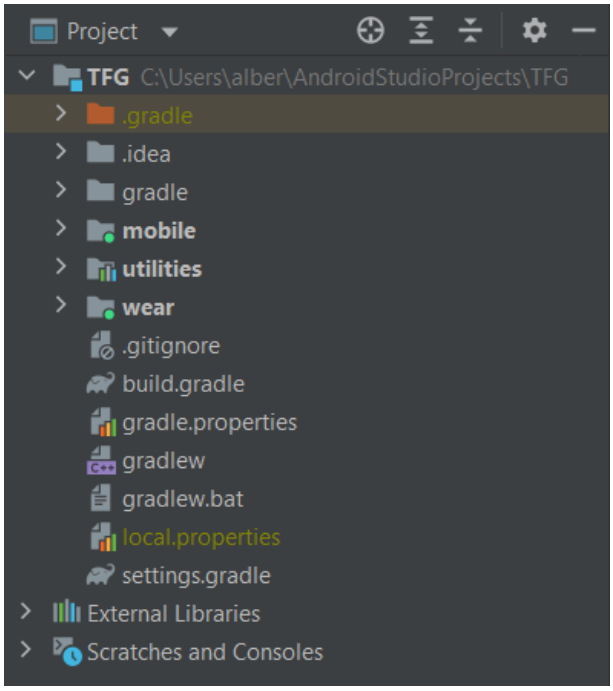

Figura 5.6: Estructura a más alto nivel de todo el proyecto

5.5.1. App del reloj

Las figuras 5.7. y 5.8 muestran respectivamente la estructura de la app Wear OS entera y la carpeta res, la cual almacena los ficheros XML de la aplicación.

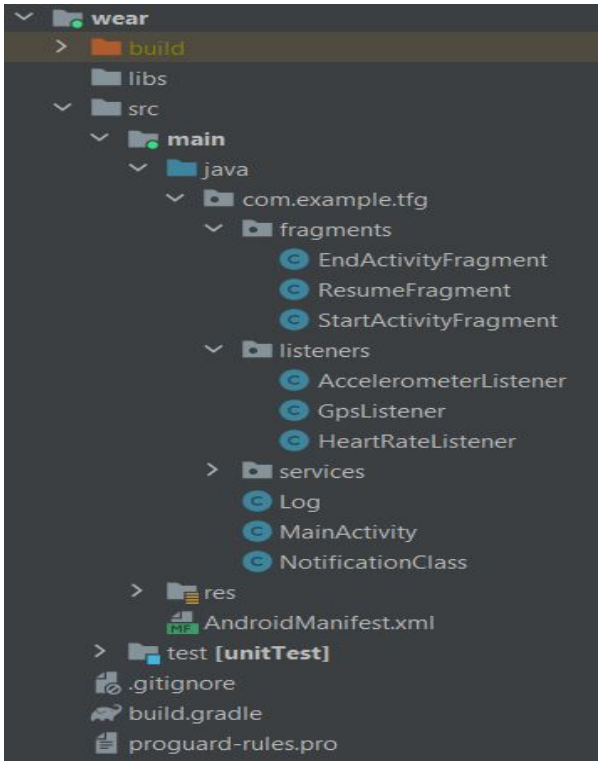

Figura 5.7: Estructura general

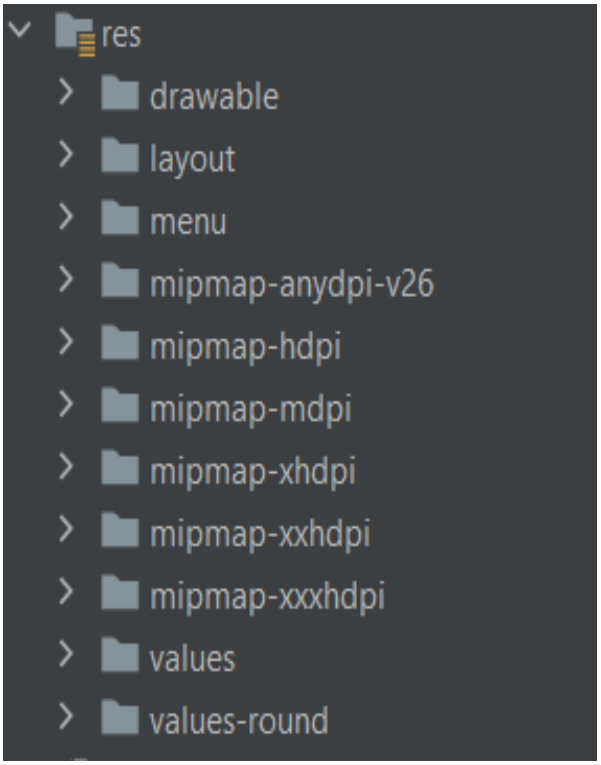

Figura 5.8: Carpeta res

Figura 5.9: Estructura de la app Wear OS

Respecto a la estructura del proyecto, el código Java de la aplicación se guarda bajo la ruta *src/main/java*, mientras el manifiesto Android se encuentra bajo *src/main*, al igual que la carpeta res. Se ve también como se ha organizado el código en los diferentes paquetes. Además, la carpeta *test* es donde se encuentran las pruebas unitarias de la aplicación. En cuanto a la carpeta res, ésta contiene a su vez diversas carpetas. Todas ellas contienen archivos XML o imágenes, con distintos propósitos. Se describen brevemente a continuación:

- Drawable: contiene imágenes y descriptores de imágenes en XML.
- Layout: contiene los ficheros XML correspondientes a las diferentes actividades o fragments de nuestra aplicación. Son los que contienen la interfaz de usuario de la app.
- Mipmap: todas las carpetas que comienzan por mipmap contienen ficheros XML también describiendo imágenes. La diferencia es que en este caso un mismo recurso

está en todas las carpetas mipmap, pero con una resolución diferente. El sistema carga automáticamente la que se adapte mejor en cada caso.

- Values: las carpetas que comienzan por la palabra values almacenan valores y constantes de la aplicación. En ella se encuentran strings o colores que se utilizan de manera repetida en la aplicación. Además, en el caso de Wear OS, values-round indica valores específicos que se toman cuando la pantalla del reloj tiene forma circular. También se puede crear una carpeta con valores que la aplicación debe tomar cuando el dispositivo tiene activado el modo oscuro. Esto se ha utilizado en la app del móvil.

5.5.2. App del móvil

La estructura de la app del móvil es muy similar a la de Wear OS. Se comentará únicamente los aspectos que difieren de la anterior. El código Java, así como el manifiesto de Android o la carpeta res se encuentran en la misma ruta. En las figuras 5.10 y 5.11 muestran respectivamente la estructura general y la carpeta res de la app.

En la estructura general de la app, a mayores de lo que había en el caso de Wear OS, se tiene la carpeta assets, la cual contiene ficheros .ttf correspondientes a diferentes tipos de letra. Respecto a la carpeta res, se ven varias carpetas que no aparecían anteriormente:

- Anim: contiene ficheros XML que corresponden a las diferentes animaciones de la app. Se ha utilizado para las animaciones entre los diferentes fragments de la aplicación.
- Font: son ficheros XML que usan los archivos .ttf de la carpeta assets que mencionamos anteriormente.
- Menu: para la barra de navegación inferior, se necesita un fichero XML que lo defina. Se encuentra en esta carpeta.
- Values-night: como se comentaba anteriormente, se utilizan ficheros XML para diferenciar la interfaz que se usa dependiendo de si el dispositivo está en modo oscuro o no.
- Xml: en ella se guardan archivos xml de configuración, que vienen por defecto.

5.5.3. Paquete común: Utilities

Por último, para el desarrollo del proyecto han sido necesarias clases que compartían ambos proyectos. La que más se ha utilizado es la clase *SharedData*, que contiene atributos correspondientes a la ubicación, frecuencia o fecha, es decir, lo que se monitoriza. Como se utiliza en ambas apps se decidió hacer este paquete común. Además de dicha clase, se tienen las clases Config y Session, usadas en el flujo de la aplicación, ya que contienen

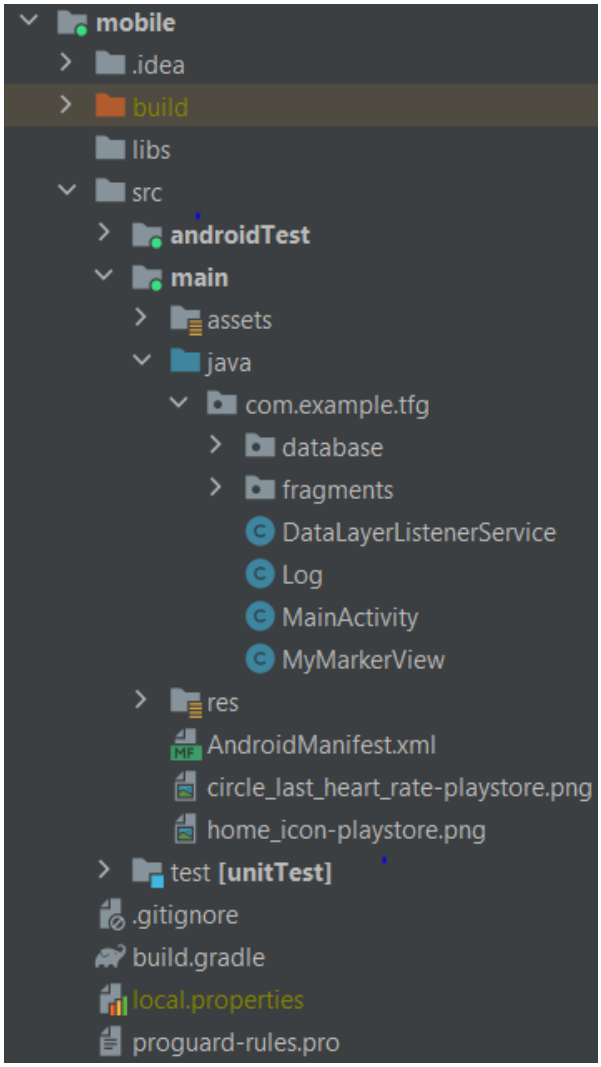

Figura 5.10: Estructura general

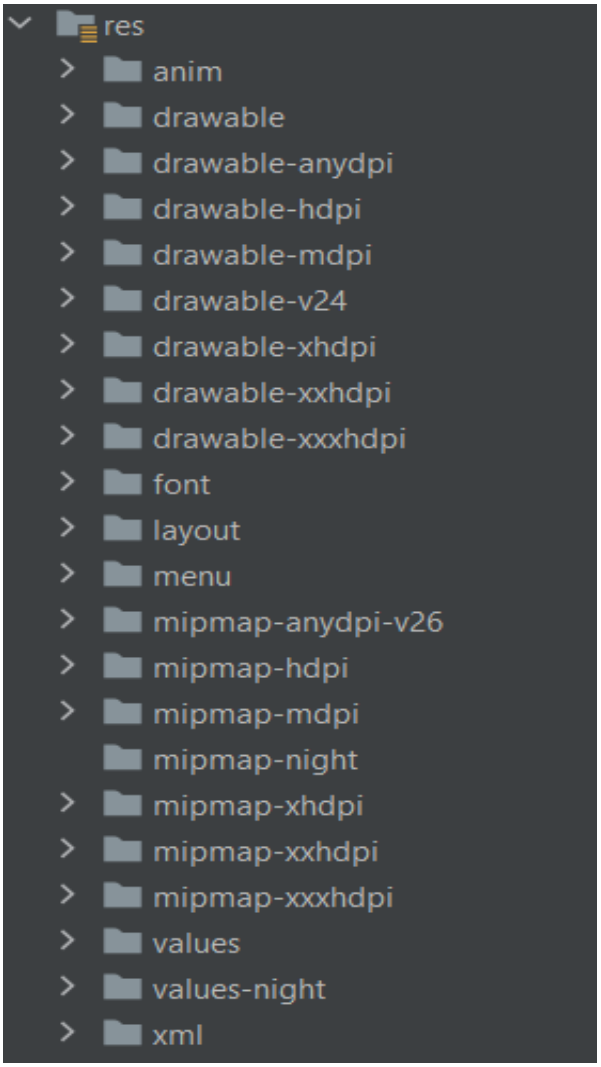

Figura 5.11: Carpeta res

Figura 5.12: Estructura de la app móvil

handlers comunes y variables de sesión respectivamente. La estructura se puede ver en la figura 5.13.

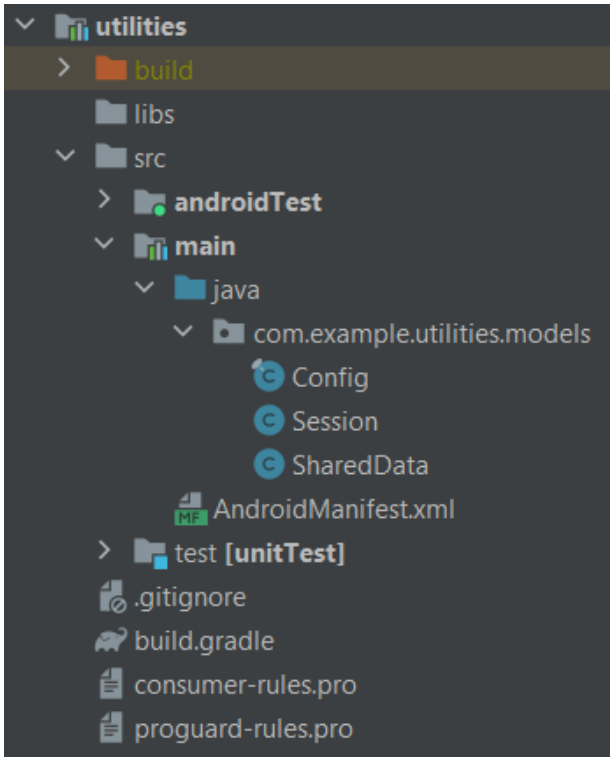

Figura 5.13: Estructura del paquete utilities

5.6. Diagrama de despliegue

El diagrama de despliegue se utiliza para representar la distribución física de los componentes software en los distintos nodos físicos de la red. En nuestro caso, tenemos 4 componentes juntando ambas aplicaciones: los dos dispositivos móviles (teléfono y reloj) y los servidores de Google que sirven para la sincronización de los datos que se envían del reloj al teléfono. La base de datos es local y se encuentra en el teléfono como se puede ver en la figura 5.14. No se ha considerado separar para cada aplicación en este caso los diagramas debido a la sencillez de éstos. Básicamente, los dos componentes a la izquierda formarían el diagrama de despliegue de la aplicación Wear OS, y todos excepto el reloj formarían el de la aplicación Android del teléfono.

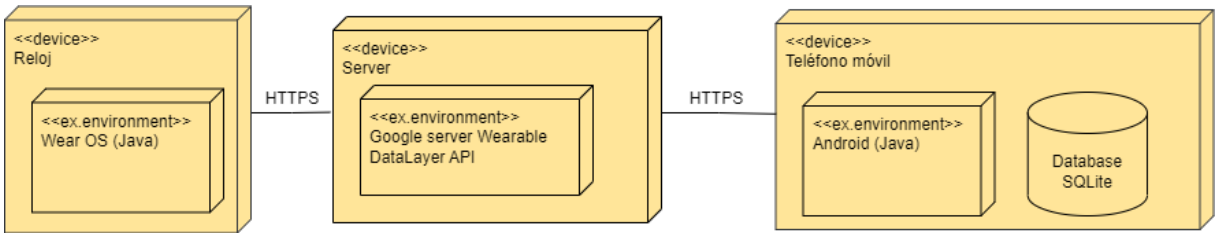

Figura 5.14: Diagrama de despliegue del proyecto

Capítulo 6

Implementación

6.1. Opciones disponibles de monitorización

En esta sección se estudiarán las posibilidades que hay a la hora de monitorizar tanto los sensores de la frecuencia cardiaca y los acelerómetros como la ubicación. Para cada uno de ellos hay diversos parámetros que se describirán. Finalmente se justificará cual es la opción elegida para conseguir nuestros objetivos de la manera más adecuada posible.

Merece la pena mencionar el tipo de programación que se ha utilizado durante todo el proyecto. Se trata de lo que se conoce como **programación orientada a eventos**, un paradigma de la programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen. En ella, es el propio usuario el que dirige el flujo del programa, a diferencia de la programación secuencial.

6.1.1. Sensores

Para monitorizar los sensores (en este caso el acelerómetro y la frecuencia cardiaca), se usan las clases *SensorManager*, *Sensor*, y se debe implementar la interfaz *SensorEventListener*, que consta de dos métodos principales: *onSensorChanged* y *onAccuracyChanged*. De ellos, el que tiene la lógica principal es el onSensorChanged, el cual se dispara cada vez que se recibe una actualización del sensor que estemos monitorizando.

Para que empiece en sí la monitorización, y que empiece a llamarse al método onSensorChanged cada cierto intervalo de tiempo, lo que se hace es llamar al método registerListener de la clase SensorManager. Éste recibe una serie de parámetros que detallamos a continuación:

- **SensorEventListener**: el listener que recibirá los datos monitorizados por el sensor.
- **Sensor**: el sensor que se va a monitorizar. En nuestro caso este será el sensor del acelerómetro o el de la frecuencia cardiaca.

- **Periodo de muestreo**: es un entero que determina la frecuencia con que se llamará al método onSensorChanged del listener (también conocido como *delay*). Viene expresado en microsegundos ($1\ \mu s = 10^{-6}\ s$). Sin embargo, hay una serie de valores predeterminados que se encuentran en la tabla 6.1 . En ella se especifican cada uno de ellos con su delay aproximado en microsegundos.

Valor	Delay (en microsegundos)
SENSOR_DELAY_NORMAL	200000
SENSOR_DELAY_UI	60000
SENSOR_DELAY_GAME	20000
SENSOR_DELAY_FASTEST	0

Cuadro 6.1: Valores predeterminados de muestreo de los sensores

- **Máxima latencia** (opcional): este parámetro es un entero, de nuevo en microsegundos, que indica la máxima latencia que puede tener la monitorización del sensor. Una vez monitorizado el sensor y creado el *SensorEvent* correspondiente, éste puede estar un tiempo determinado almacenado antes de ser enviado al onSensorChanged. El máximo tiempo que puede estar almacenado es lo que indica este parámetro. Los eventos se almacenan en una cola FIFO, y cuando un evento alcanza la máxima latencia, todos los eventos se mandan de manera secuencial al onSensorChanged. Llamar al registerListener sin este parámetro es lo mismo que llamarlo poniendo un 0 en él, es decir, por defecto es 0.

- **Handler** (opcional): el handler al cual se enviarán los eventos del sensor. Esto es útil cuando se quiera mandar los eventos a un hilo que corra en segundo plano, ya que por defecto se mandarían al hilo principal o main thread.

En este caso no se necesitan los parámetros opcionales, y simplemente se especificarán los parámetros obligatorios. Se establecerá un ratio de muestreo de 1000000, para recibir actualizaciones cada segundo. Sin embargo hay que tener en cuenta que, independientemente del ratio de muestreo, la aplicación sigue monitorizando. Para no consumir mucha batería del reloj, lo que se hará es monitorizar cada cierto intervalo de tiempo.

6.1.2. Ubicación

La otra característica importante que monitorizamos en este trabajo se trata de la ubicación, lo cual se hace mediante el GPS del reloj. Aún así, dentro de la ubicación existen dos posibilidades de implementación, que describimos de manera resumida a continuación:

- **API FusedLocationProvider**: esta API ha sido creada por google y es una solución que ha ganado popularidad últimamente. La mayor ventaja de ella es que para

monitorizar la ubicación utiliza los datos de los sensores GPS e internet de manera inteligente y, por tanto, no tiene que gestionar el programador que señal utilizar.

- **API LocationManager**: esta es la alternativa a la anterior y su diferencia principal es que es independiente de los servicios de google. Además, el desarrollador debe de gestionar que señales utilizar en cada caso según sea más conveniente.

Tras investigar y ver tutoriales y ejemplos de ambas opciones, se eligió la segunda opción. La razón principal fue que había más ejemplos de aplicaciones Wear OS que la utilizasen, y además hay bastante documentación al ser anterior a la API de google. Realmente, las funciones que ofrecen ambas son similares, así que lo importante es utilizar la ubicación de manera correcta independientemente de por lo que se opte.

Esta API se basa principalmente en la clase *LocationManager*. Ésta contiene todos los métodos y variables necesarias para monitorizar la ubicación. Similar a lo que ocurría en el caso de los sensores, necesitamos un listener que se encarga de recibir las ubicaciones actualizadas. El método principal del listener es el *onLocationChanged*, que se dispara cada vez que una nueva ubicación es recibida.

En la ubicación hay una serie de parámetros importantes, relacionados con la precisión y el tiempo entre dos mediciones consecutivas. El método *requestLocationUpdates* de la clase LocationManager es el que se encarga de actualizar la ubicación. Recibe una serie de parámetros que listamos y describimos brevemente a continuación:

- **Provider**: el provider es el que indica que señales se van a utilizar para monitorizar la ubicación. Esto es sumamente importante porque dependiendo de las condiciones que se tengan, se tendrá que usar uno u otro (esto se explica un poco más adelante). La red o los satélites son ejemplos de providers que se pueden indicar en este campo. Las opciones disponibles se detallan en la tabla 6.2.
- **Tiempo mínimo**: indica el tiempo mínimo entre actualizaciones de la ubicación, en milisegundos.
- **Distancia mínima**: indica la distancia mínima entre actualizaciones de la ubicación, en metros.
- **Location listener**: listener que escuchará las actualizaciones de ubicación. Contiene la mayoría de la lógica de la aplicación en cuanto a la ubicación.

Para elegir un provider tenemos que tener en cuenta diversos factores, como la disponibilidad de los satélites o el internet. En interiores los satélites suelen funcionar mal, por lo que es más recomendable usar el network provider. Sin embargo, cuando se está en el exterior, posiblemente no se disponga siempre de Wifi o de datos móviles, y por tanto la mejor opción será el gps provider. Además, muchos relojes (como el de este trabajo) ni siquiera admiten una tarjeta SIM, por lo que el network provider se reduce a considerar la red Wifi.

Provider	Descripción
GPS_PROVIDER	Determina la ubicación usando satélites GNSS.
NETWORK_PROVIDER	Determina la ubicación basándose en las antenas de telefonía móvil y el Wifi.
FUSED_PROVIDER	Combina datos de los otros providers para ofrecer la ubicación más precisa posible. Se utiliza implícitamente en las actualizaciones de ubicación que se hacen con la clase Criteria (lo vemos más adelante).
PASSIVE_PROVIDER	Es un provider especial que se utiliza para recibir actualizaciones de ubicación de forma pasiva cuando otras aplicaciones o servicios las soliciten sin que las solicitemos nosotros de manera explícita. Este provider devuelve ubicaciones generadas por otros providers.

Cuadro 6.2: Providers disponibles para la monitorización de la ubicación

Para saber que provider es más conveniente en cada caso hay un método de la clase LocationManager llamado *getBestProvider*. Éste recibe un objeto de la clase *Criteria*, que indica diversas configuraciones de como se quiere monitorizar la ubicación, y un booleano indicando si se quieren permitir providers solo si están activados. Este método es el que se usará, y es el más indicado, pues de manera automática escoge el provider que ofrece una mejor precisión en cada situación.

6.2. Detección del reposo

Recordemos que el objetivo principal del proyecto era la monitorización de la frecuencia cardiaca en situaciones de reposo. Por ello, tenemos que encontrar una manera lo más precisa y fiable posible de detectar dicha situación. En esta sección se explicará de manera resumida cual es el algoritmo que hay detrás para detectar cuando estamos en reposo.

6.2.1. Comunicación en la aplicación del smartwatch

El componente involucrado en la detección del reposo son únicamente los acelerómetros. En un principio, se pensó en utilizar la ubicación también para detectar el reposo, pero se acabó descartando debido a la imprecisión y fiabilidad de ésta. No obstante, la ubicación se monitoriza igualmente ya que almacenamos las ubicaciones donde se mide la frecuencia.

A su vez, en esta parte de la aplicación hay una clase que ha sido indispensable para el desarrollo de la aplicación: la clase Handler. Se describe brevemente un poco más adelante.

En el smartwatch, la aplicación necesita tener varios hilos corriendo a la vez en segundo plano. Esto es así porque si se delegase todo el trabajo sobre el hilo principal, éste se sobrecargaría y no funcionaría correctamente la aplicación. En el caso del proyecto, se tiene un solo servicio, y es éste el que se encarga de comunicarse con los sensores para que se inicie o pare la monitorización.

Para cada uno de los sensores y la ubicación, hay un hilo que corre en segundo plano. Por ello, se necesita una forma de comunicación entre los distintos componentes e hilos de la aplicación. Esto se consigue gracias a dos cosas. La primera viene dada por como se ha implementado el servicio. Se trata de lo que se denomina un *bound service*. En resumidas cuentas, este tipo de servicios implementan la interfaz Cliente-Servidor. Un componente de nuestra aplicación, como por ejemplo una actividad, se vincula al servicio y actúa como cliente, mientras que el servicio actúa como servidor. Esto permite que se envíen peticiones y se respondan y, por tanto, que sean capaces de efectuar una comunicación entre procesos (IPC).

La segunda es lo que se comentaba en un párrafo anterior, la clase *Handler*. Esta clase es propia de Android, y el descubrirla y entenderla ha facilitado mucho el desarrollo del proyecto. Un handler tiene dos funciones principales, que son las siguientes:

- Programar mensajes e hilos o *runnables* para que sean mandados o ejecutados respectivamente en un futuro.
- Mandar un mensaje a un hilo distinto al que estamos trabajando.

Ambas funciones han sido usadas en el proyecto, pero la segunda es la que tiene un mayor peso en esta parte de la aplicación. Para iniciar o parar la monitorización de la frecuencia cardiaca usamos distintos handlers y pasamos dichos mensajes. El flujo de la detección del reposo es lo que se explica a continuación.

6.2.2. Flujo de detección del reposo

Para detectar el reposo, lo que se hace es utilizar los acelerómetros de una manera adecuada. Además, la ubicación se monitoriza periódicamente para guardar en todo momento la ubicación más actualizada posible. El objetivo de la aplicación es medir la frecuencia cada media hora siempre que estemos en reposo. Dicho esto, una vez se pulsa en iniciar la monitorización el flujo es como sigue:

1. Lo primero es comprobar que la última monitorización exitosa se llevó a cabo hace más de media hora. Para ello, se accede a una *SharedPreference* que guarda la fecha de esto. Si no pasó media hora, entonces se indica al usuario cuantos minutos quedan para una nueva monitorización. Si pasó el tiempo necesario, entonces se inicia la monitorización de la ubicación y los acelerómetros. En consecuencia, realmente solo se monitorizan tanto la ubicación como los acelerómetros en momentos concretos, para evitar un consumo excesivo de batería. Durante la media hora entre medidas no están activas dichas monitorizaciones.

2. Una vez iniciada la monitorización, para que se considere que el usuario está en reposo deben pasar 10 segundos seguidos en los que el usuario esté en reposo. Se considera que se está en reposo si la aceleración está debajo de una determinado valor. A lo largo de los 10 segundos, si se recibe una aceleración por encima de este valor, entonces se indica que el usuario no está en reposo y se resetean esos 10 segundos. Suponiendo que el usuario permanece en reposo 10 segundos, se pasa a la siguiente fase.

3. Cuando estamos en reposo, la frecuencia se monitoriza indicando al usuario en tiempo real su frecuencia. De nuevo, si durante los 30 segundos se detecta que el usuario deja de estar en reposo, entonces volvemos al punto anterior y debemos esperar 10 segundos de nuevo en reposo. Hasta que no esté el usuario en reposo durante los 30 segundos no se considera que la medida ha sido realizada con éxito.

4. Cuando la medida ha sido realizada con éxito, se actualiza el valor de la última medida, se deja de monitorizar todo (frecuencia, aceleración y ubicación) y se transmiten las medidas realizadas junto con la fecha actual y la última ubicación disponible.

Todo lo anterior se realiza mediante la comunicación entre los distintos hilos de la aplicación, como coméntabamos anteriormente, con la clase Handler. Ésta permite enviar y escuchar mensajes procedentes de otros hilos, y ejecutar una determinada acción dependiendo del mensaje. Por ello, las diferentes acciones comentadas son ejecutadas de acuerdo a lo anterior.

6.3. Transmisión y almacenamiento de los datos

En esta sección se resume como se ha llevado a cabo la transmisión de las frecuencias monitorizadas al smartphone y el método de almacenamiento de dichas frecuencias.

6.3.1. Transmisión de los datos

Una vez se ha detectado un estado de reposo y medido una tanda de frecuencias, proceso el cual ha sido explicado en las secciones anteriores, es necesario implementar una forma fiable de pasar estos datos al teléfono, para su posterior almacenamiento y visualización. La mejor manera de hacer esto es utilizar una API que está diseñada específicamente para este propósito: la API **Wearable Data Layer**. En las apps Wear OS, no se recomienda en ningún caso utilizar sockets de bajo nivel para comunicarse. Se describe brevemente la API a continuación.

La API Wearable Data Layer es parte de los servicios de Google, y solo es compatible con relojes Wear OS que se conecten a teléfonos Android. Éste es el caso de este proyecto, y en estas condiciones es mucho mejor opción que cualquiera de las otras que hay disponibles. A su vez, Wearable Data Layer está constituida por diversas APIs para mandar información entre diferentes nodos. Las principales son las siguientes:

- **Message Client**: su función principal es mandar mensajes a nodos que esten conectados actualmente. Sin embargo, la gran limitación de esta API es que los nodos deben de estar conectados para que se manden y reciban los mensajes de manera correcta. Además, no soporta que los datos que se envíen sean de un tamaño mayor a 100kb.

- **Channel Client**: esta API solo está disponible a partir de Wear OS 2.0, y la diferencia principal con la anterior es que con ella se pueden mandar datos de más de 100kb (y de diversos tipos, no solo mensajes). Sin embargo, de nuevo está creada con el propósito de establecer una comunicación en tiempo real entre los dispositivos. Por ello, necesita que ambas apps esten iniciadas para que la comunicación sea efectiva.

- **Data Client**: esta tercera y última API, es la que ha sido elegida para este proyecto. El motivo principal es que soporta el envío de datos de manera asíncrona, es decir, que no hace falta que ambos nodos esten conectados para que se puedan mandar los datos. Además, al igual que en el caso anterior el tamaño de éstos puede ser grande. Cuando se da esta situación lo recomendable es utilizar un objeto de la clase Asset y vincularlo a un objeto de la clase DataItem. No se entrará más en detalle en esto ya que no ha sido necesaria en este caso.

El uso de la API DataClient se puede resumir en lo siguiente:

- Crear un objeto de la clase *PutDataMapRequest* al que le pasamos un path específico que determina el canal de comunicación. Este canal debe de coincidir con el que se use en el móvil.

- Añadir a dicho objeto los datos que queramos transmitir. Dependiendo del tipo de datos que se transmitan se usará un método distinto.

- Crear un objeto de la clase *PutDataRequest* que recibe el objeto anterior y lo formateará como un request.

- Crear una lista de objetos de la clase *Task<DataItem>*, a la cual se añade el request que se acaba de crear. Esto automáticamente hace que se mande el objeto a los nodos que estén conectados. Además se pueden añadir dos listeners para indicar si se han mandado de manera exitosa o no los datos.

En la figura 6.1 se puede ver cada uno de los anteriores pasos. En este caso se le pasa un objeto de la clase SharedData del cual se extraen los datos que se mandan.

Una vez que se tiene implementado la API del lado del reloj, es necesario implementar una forma de recuperar dichos mensajes en la app del móvil. Para ello, en el móvil se tienen dos alternativas:

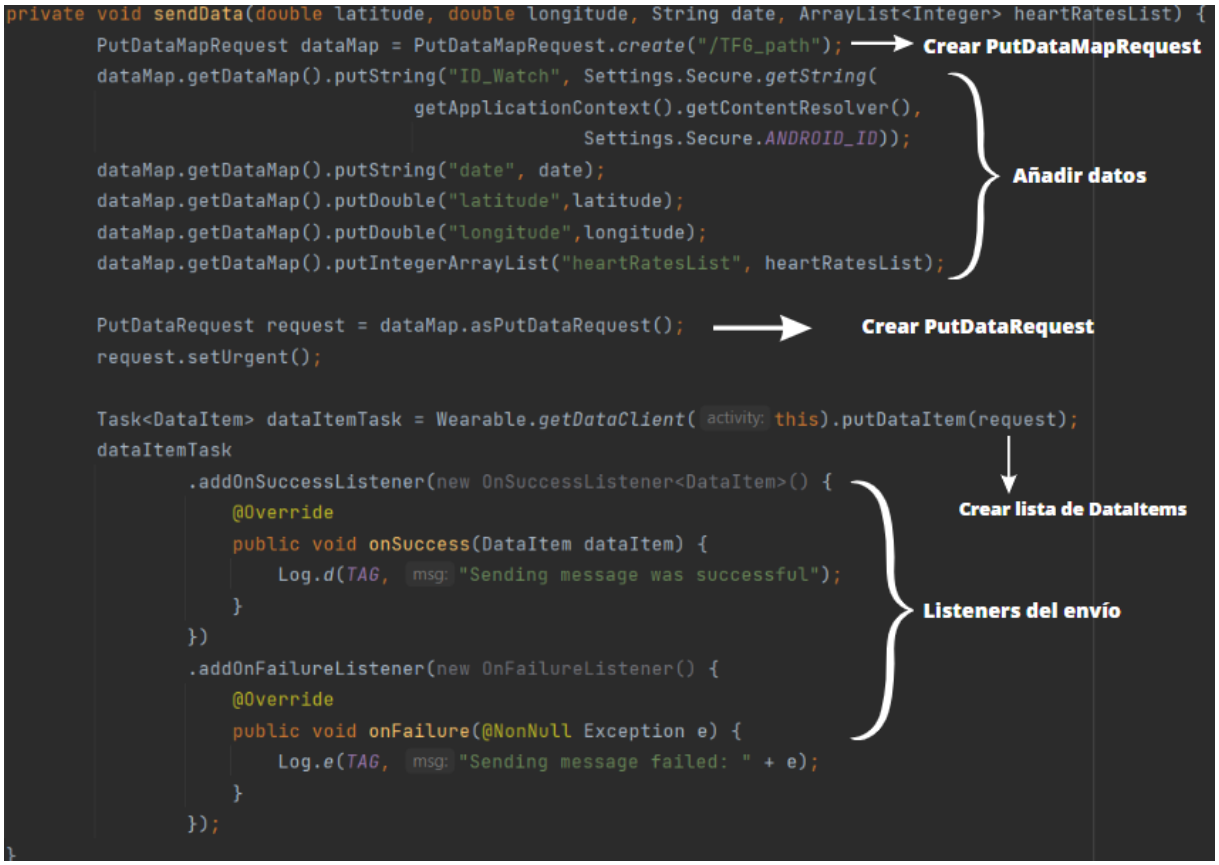

Figura 6.1: Método para mandar los datos del reloj al móvil

- Implementar el método **DataClient.onDataChanged**. Este método escucha los datos que se mandan desde el reloj y los puede extraer para su posterior procesamiento y almacenamiento.
- Extender la clase **WearableListenerService** e implementar el método onDataChanged. Este método hace exactamente lo mismo que en el caso anterior. La diferencia radica en la sincronización de los datos. Mientras que en el caso anterior solo se escuchan eventos mientras la app del móvil está activa, usar el WearableListenerService permite recibir datos que fueron mandados antes de que se iniciase la app. Esto es sumamente importante, porque así no se tienen que estar usando las dos apps a la vez para que todo funcione correctamente. Por ello se se elige esta opción en el proyecto.

Sincronización

Al hablar del WearableListenerService y de la API DataClient, se ha mencionado el concepto de la sincronización entre el reloj y el móvil. Cuando el reloj no está conectado al teléfono, es necesario garantizar que las mediciones se transmitan y almacenan de forma

correcta. Por suerte, la API DataClient junto con el uso del WearableListenerService hacen que este proceso sea realizado de manera automátizada.

Cuando las apps no están ambas activas, o el reloj no está conectado al teléfono, los datos que se mandan al teléfono realmente se almacenan en unos servidores de Google de manera temporal hasta que se conecta de nuevo el teléfono y se corre la app del móvil. Cuando además ni siquiera tiene internet el reloj, entonces las medidas se almacenan localmente hasta tener internet o conectarse al teléfono de nuevo, para mandar en ese momento los datos. Esto hace que no se pierdan esos datos y que se pueda usar tranquilamente la app en el reloj sin necesidad de llevar encima el teléfono.

6.3.2. Almacenamiento de los datos

Durante el proyecto, se barajarón varias opciones de almacenamiento, como el almacenamiento de los datos en un servidor o de manera local. Tras una reflexión, se concluyó que sería más conveniente hacerlo de manera local para así no depender de terceros, siendo además no muy grande la cantidad de datos a almacenar. Para ello, se ha utilizado la base de datos **SQLite** que proporciona varias funcionalidades y su uso y manejo es sencillo.

El smartphone es donde se almacenarán todos los datos de las frecuencias, junto con su fecha, latitud, longitud y un identificador del reloj. Esto es necesario ya que si se conecta un móvil a otro reloj, se tienen que mostrar solo los datos del reloj que esté conectado o el último que se hubiese conectado.

Respecto a SQLite, no se entrará en detalle sobre su funcionamiento, pero en resumen, se basa principalmente en el uso de dos clases, **SQLiteOpenHelper** y **SQLiteDatabase**. Para usarlo, se crea una nueva clase que extienda a SQLiteOpenHelper, y se implementa el método onCreate, que se encarga de crear (si no lo estaba ya, sino simplemente se accede) la base de datos que se defina en él. Este método se ejecuta cada vez que se instancia un objeto de dicha clase. También se debe implementar el método onUpgrade, que se llama cuando la estructura de la base de datos cambia. En este caso no se gestiona esta posibilidad así que simplemente es un método vacío. A partir de esos dos métodos, se pueden ir creando más métodos de manera específica de acuerdo a lo que se quiera lograr.

6.4. Visualización de los datos

Esta sección resume como se han implementado las funcionalidades correspondientes a mostrar los datos por pantalla. Por un lado, para el historial de frecuencias se ha utilizado la librería MpAndroidChart [18] y para el mapa el SDK de Google Maps [12].

6.4.1. MpAndroidChart

MPAndroidChart es una librería de gráficos para Android que permite crear gráficos y diagramas en aplicaciones Android. Fue creada por el usuario PhilJay en github, es de

código abierto y está escrita en Java, con una API flexible que facilita la personalización y configuración de los gráficos para satisfacer requisitos específicos.

MPAndroidChart admite una amplia variedad de tipos de gráficos, incluidos gráficos de línea, de barras, de sectores, entre otros. También proporciona funciones como el zoom, el desplazamiento y el resaltado, lo que facilita la interacción de los usuarios con los gráficos.

Una de las principales ventajas de MPAndroidChart es su rendimiento. La biblioteca está diseñada para ser altamente optimizada, con características como la aceleración por hardware y la caché para garantizar que los gráficos se rendericen rápidamente y de manera fluida incluso con conjuntos de datos grandes. Además, destaca frente a otras es su buena documentación y comunidad creada en torno a ella.

Para utilizar MPAndroidChart en el proyecto de Android, lo primero que hay que hacer es agregar la biblioteca como una dependencia en el archivo gradle a nivel de app. Una vez hecho esto, primero se agrega el elemento XML correspondiente al archivo de layout y, a continuación, se configura con los datos y personalizaciones correspondientes en la activity o fragment al que está vinculado el XML. En este proyecto se han creado cuatro gráficos distintos: dos de líneas o *LineChart* y dos de barras o *BarChart*. Los de líneas muestran los datos de las frecuencias medidas en un día o un mes, mientras que los de barras muestran la información de la última semana o un año concreto.

6.4.2. Maps SDK

Cada frecuencia medida se almacena junto con su ubicación en la base de datos. Esta información se muestra en la aplicación del móvil en un mapa, siendo cada frecuencia un marcador en él. Para realizar lo anterior se ha optado por utilizar el SDK de Maps, que es la actualmente la mejor opción para dispositivos Android.

Con el SDK de Maps para Android, se pueden agregar mapas a una app Android (incluso también para apps Wear OS) que utilizan datos, reproducciones de mapas y respuestas gestuales de Google Maps. También se puede ofrecer información adicional sobre las ubicaciones del mapa y facilitar la interacción con el usuario agregando marcadores, polígonos y superposiciones al mapa. El SDK es compatible con los lenguajes de programación Kotlin y Java y ofrece bibliotecas y extensiones adicionales para funciones avanzadas y técnicas de programación.

Lo primero que hay que hacer es obtener una API Key de Google. Para ello, es necesario crear un proyecto en la consola de google y seguir los pasos descritos en [4]. Acto seguido, en Android Studio se debe configurar el proyecto para añadir la SDK de Maps [6]. Una vez hecho esto, ya se pueden agregar mapas a la aplicación. En [5] se detallan los pasos para conseguir lo anterior. No se entrará más en detalle, ya que ésta no es la finalidad principal del proyecto.

6.5. Dificultades en el desarrollo

La última sección del capítulo se dedica a comentar a grandes rasgos cuales han sido las mayores dificultades que han surgido durante el proyecto, y como se han solucionado. La mayoría de ellas se debían a la falta de experiencia en Android o Wear OS. Se listan a continuación:

- **Modo ambiente:** se trata de una característica particular de Wear OS. Para consumir menos batería, los dispositivos wearables disponen de un modo ambiente que se activa cuando el usuario permanece un tiempo sin interactuar con la pantalla. Al principio del proyecto, cuando todavía no se disponía de experiencia suficiente, la aplicación se cerraba automáticamente al intentar monitorizar cualquier tipo de sensor. Tras unos días investigando se descubrió la razón, y se implementó en la app el modo ambiente para que funcionase todo de acuerdo a lo esperado.

- **Problemas con los hilos:** la parte del proyecto que suponía un mayor reto a priori era como unificar distintos componentes (ubicación y sensores) para lograr detectar el reposo y empezar a monitorizar la frecuencia en ese momento. No se tenía claro cuantos servicios hacían falta, se producían errores por intentar instanciar un mismo hilo dos veces y no se sabía como organizar el código. De nuevo, hicieron falta varios días de familiarización y aprendizaje de la gestión de runnables, handlers y executors para finalmente intentar algo que funcionase e ir ajustándolo al objetivo del proyecto. En resumen, se optó por tener un hilo para cada sensor y para la ubicación, junto con un handler propio. Fue muy importante también la clase Looper, cuya instancia se asocia a un handler y permite escuchar los mensajes procedentes de otros hilos, para ejecutar distintas acciones en consecuencia.

- **Ubicación en interiores:** al principio del proyecto, la aplicación no funcionaba al ejecutarse en un dispositivo real, pero sí en el emulador de Android Studio. Se probaron las dos formas de monitorizar la ubicación (discutidas en una sección anterior) y aún así no se conseguía nada. Tras investigar, resulta que tenía que ver con los providers que se escogían al monitorizar la ubicación. En interiores no funcionan los satélites, y esta era la razón por la que no funcionaba en el reloj pero sí en el emulador.

Precisamente el último punto de los anteriores es el que motivó un gran cambio que tuvo lugar cuando el proyecto estaba bastante avanzado. En primera instancia, la ubicación iba a ser utilizada para detectar el reposo, pero debido a problemas de fiabilidad y lo comentado sobre los interiores, se acabó descartando la ubicación y esa tarea se delegó completamente a los acelerómetros. No obstante, se pensó que podía ser útil almacenar la ubicación para agregar alguna funcionalidad a mayores en la aplicación del teléfono. Por ello, se tuvo que volver a reajustar parte del código para satisfacer los nuevos requisitos que surgieron.

Capítulo 7

Pruebas

Una parte esencial de todo proyecto de software es el de las pruebas. Cuando se está trabajando en un gran proyecto que involucra a un gran número de personas, y requiere una gran inversión en tiempo y dinero, es obligatorio implementar diferentes pruebas a lo largo del desarrollo del mismo. Los errores se detectan difícilmente cuando el código es grande, y por ello la importancia de las pruebas.

Las pruebas software se suelen clasifican en las siguientes categorías:

- **Pruebas unitarias:** Comprueban que cada una de las piezas o unidades más pequeñas del software en el que se está trabajando funcione correctamente. Estas pruebas se aplican de manera individual y son las primeras que deben realizarse durante todo el proceso de desarrollo.

- **Pruebas de integración:** El objetivo de estas pruebas es comprobar que los diferentes componentes operan bien juntos. En ocasiones sucede que un módulo que funciona perfectamente de manera aislada, pero una vez se integra con el resto provoca algún tipo de fallo en el sistema, por lo que es necesario supervisar esa integración para obtener los mejores resultados. Es un proceso más complejo y debe realizarse después de las pruebas unitarias.

- **Pruebas funcionales:** en ellas se comprueban las funciones del software creado para establecer la usabilidad y las características de cara al mercado. Se consideradan pruebas de caja negra porque lo que se verifica es el comportamiento del sistema, si todo funciona tal y como está establecido en el documento de software *Software Requirement Specification* (SRS)

- **Pruebas de aceptación:** antes de comenzar a trabajar en el proyecto, el equipo responsable debe definir cuáles son los criterios de aceptación. Por otra parte, si durante la fase de desarrollo se toma cualquier decisión que añada o disminuya criterios, también debe dejarse constancia. Durante las pruebas unitarias y de integración se realizarán este tipo de pruebas para corroborar que todo el sistema funciona según lo esperado.

- **Pruebas de rendimiento:** el objetivo es comprobar la respuesta del software ante cargas de trabajo diferentes y en condiciones reales. Sirven para definir cuestiones como la estabilidad o la velocidad de una aplicación de software y están directamente relacionadas con la experiencia del usuario y la tasa de conversión.
- **Pruebas de estrés:** antes de dar por finalizado el proceso de desarrollo de un proyecto software es necesario comprobar cuánta tensión se puede soportar antes de que se produzca algún error. Para ello, se envía mucha más información de lo habitual para comprobar en qué punto se satura el sistema. Esto es lo que se hace en estas pruebas.
- **Pruebas de regresión:** en ocasiones, el desarrollador decide modificar alguna funcionalidad en el sistema. Este tipo de pruebas tienen como objetivo comprobar que los cambios en un componente del software no provocan reacciones no deseadas o fallos en otros elementos no modificados. No tendría sentido introducir una nueva función si esto va a tener consecuencias negativas sobre el resto.
- **Pruebas de humo:** son pruebas funcionales que ayudan a determinar si el conjunto del software funciona bien y si está preparado para ser sometido a pruebas más exhaustivas. Están pensadas para garantizar la funcionalidad principal, así que si no se superan no se avanza ni se programa ninguna prueba más compleja.

Tras esta introducción para conocer cuales son las pruebas típicas que se realizan en un proyecto software, en esta memoria se mostrarán únicamente las pruebas unitarias de alguna funcionalidad del sistema. Recalcar que lo suyo sería hacerlas para cada fragmento del código, pero por falta de tiempo se harán únicamente para dos funcionalidades, indicando como se han llevado a cabo.

7.1. Pruebas unitarias: JUnit

Para realizar las pruebas unitarias en Android Studio, se ha utilizado JUnit. JUnit es un conjunto de bibliotecas para hacer pruebas unitarias de aplicaciones Java. JUnit es un framework que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

En Android Studio, el proceso para realizarlas es como sigue:

1. Abrir la clase la cual se quiere probar.
2. Pulsar Alt + enter sobre el nombre de la clase. Pulsar en "Create test" a continuación.

3. Se abrirá un menú donde se pueden seleccionar las librerías que usar para el test, así como los métodos que se desean probar. En la librería, seleccionar JUnit5. Una vez se hayan seleccionado los métodos que se quieren testear, pulsar en ok.

4. Se crea entonces una clase Java con los métodos que fueron seleccionados donde ya se pueden escribir los tests. Los archivos test se encuentran en la ruta *app/src/test* por defecto. También se puede utilizar la carpeta *androidTest*, cuando los test requieren algún tipo de intrumentación Android.

A continuación se muestra y explica brevemente el código de las pruebas unitarias para dos funcionalidades. Estas son actualizar la interfaz de la pantalla indicando si estamos en reposo o no y mostrar la frecuencia en caso de estarlo, y la de parar de monitorizar loa acelerómetros (esto es análogo para la frecuencia y el GPS). La primera funcionalidad comprende dos métodos de la clase *ResumeFragment*, mientras que la segunda es un método de la clase *AccelerometerListener*.

Para ejecutar las pruebas, se puede ejecutar cada método por separado o toda la clase del test entera. Ambas formas se llevan a cabo mediante unos botones verdes a la izquierda del nombre de los métodos y de la clase respectivamente.

7.1.1. Indicar estado de reposo y frecuencia

Cuando se está en reposo y monitorizando la frecuencia, el método **updateHeartRate** recibe el valor de la frecuencia y modifica la interfaz indicando el estado de reposo y el valor de la frecuencia. El código se muestra en la figura 7.1. En ella se ve que se crean dos TextViews mediante el método *mock*. En pruebas unitarias un mock es un objeto que reemplaza al de verdad para realizar las pruebas. En este caso lo único que se necesitan son TextViews, pero puede ser cualquier otro tipo de objeto. A continuación se instancia la clase y se llama al método que actualiza la frecuencia. Finalmente se llama al método *verify* que comprueba que coincida lo que se ha ejecutado con lo esperado.

El método **updateNoRest** se encarga de actualizar la interfaz cuando se detecta que el usuario no está en reposo. Indica por pantalla que el usuario no está en reposo y quita el valor de la frecuencia que hubiese anteriormente, estableciéndola a "–". El código es análogo al del anterior método, y se muestra en la figura 7.2.

7.1.2. Parar de monitorizar los acelerómetros

En el proyecto, cada elemento que se monitoriza (frecuencia cardiaca, aceleración y GPS) tiene una clase que se encarga de comenzar a monitorizarlo y pararlo. El método **stopListeners** se encuentra en las tres clases y sirve para parar la monitorización. La figura 7.3 muestra el código del test de dicho método. El código es similar a los anteriores, usando los mocks y comprobando que coincidan los valores con lo que se espera.

```java
@Test
public void testUpdateHeartRate() {

    TextView mockRestIndicator = mock(TextView.class);
    TextView mockHeartRate = mock(TextView.class);

    ResumeFragment testUpdateHeartRateFragment = new ResumeFragment();

    testUpdateHeartRateFragment._restIndicator = mockRestIndicator;
    testUpdateHeartRateFragment._heartRate = mockHeartRate;

    testUpdateHeartRateFragment.updateHeartRate(80);

    verify(mockRestIndicator).setText("Estás en reposo");
    verify(mockRestIndicator).setTextSize(20);
    verify(mockHeartRate).setText("80");
}
```

Figura 7.1: Test del método updateHeartRate

```java
@Test
void updateNoRest() {

    TextView mockRestIndicator = mock(TextView.class);
    TextView mockHeartRate = mock(TextView.class);

    ResumeFragment resumeFragment = new ResumeFragment();
    resumeFragment._restIndicator = mockRestIndicator;
    resumeFragment._heartRate = mockHeartRate;

    String expectedRestIndicatorText = "No estás en reposo";
    String expectedHeartRateText = "--";

    resumeFragment.updateNoRest();

    verify(mockRestIndicator).setText(expectedRestIndicatorText);
    verify(mockRestIndicator).setTextSize(20);
    verify(mockHeartRate).setText(expectedHeartRateText);
}
```

Figura 7.2: Test del método updateNoRest

```java
@Test
public void stopListeners() throws InterruptedException{

        AccelerometerListener accelerometerListener = new AccelerometerListener();
        SensorManager sensorManager = mock(SensorManager.class);
        SensorEventListener eventListener = mock(SensorEventListener.class);
        Sensor accelerometerSensor = mock(Sensor.class);
        accelerometerListener.setSensorManager(sensorManager);
        accelerometerListener.setEventListener(eventListener);
        accelerometerListener.setAccelerometerSensor(accelerometerSensor);

        Thread thread = new Thread(accelerometerListener);
        thread.start();
        accelerometerListener.stopListeners();

        verify(sensorManager).unregisterListener(eventListener, accelerometerSensor);
        assertFalse(accelerometerListener.getIsResting());
        assertEquals( expected: 0, accelerometerListener.getRestAccumulatedTime());
        assertEquals( expected: 0, accelerometerListener.getRestStartTime());
        SharedData singleton = SharedData.getInstance();
        assertEquals( expected: 0, singleton.getAcceleration());

}
```

Figura 7.3: Test del método stopListeners

Capítulo 8

Conclusiones

Una vez finalizado el trabajo fin de grado, conviene analizar si se han conseguido los objetivos que en un principio se establecieron, además de comentar en líneas generales como de satisfactorio ha sido el desarrollo del TFG, de acuerdo a lo que se esperaba de él. Por último, se propondrán diferentes funcionalidades que podrían mejorar notablemente esta aplicación de cara al futuro.

El trabajo que ha sido desarrollado tenía un componente relativamente alto de incertidumbre. Si bien es cierto que al final el desarrollo coincidía en su mayoría con el lenguaje Android, usado para desarrollar apps de teléfonos móviles, la realidad es que cuando se propuso y se decidió aceptar este proyecto tampoco se disponía de conocimientos en este ámbito. Por ello, se ha necesitado una gran etapa de aprendizaje previa con el objetivo de aprender a programar en Java para Android. Una vez hecho esto, también supuso un tiempo entender Wear OS, para saber en que era igual a Android y en que difería. Con todo lo anterior, ya se tenía una base suficientemente sólida para empezar el desarrollo del TFG.

Durante el desarrollo del TFG, se han ido aprendiendo múltiples funciones disponibles del lenguaje Android, y con gran satisfacción he de decir que, aunque al principio quizá tenía algo de miedo debido a lo comentado anteriormente, he conseguido superar los obstáculos con mayor o menor dificultad y me siento bastante cómodo programando en Android tras terminar el proyecto. Por ello, considero que tocar algo que no sabía me ha ayudado a ser un futuro ingeniero informático un poco más versátil y con más herramientas que poder usar.

Respecto a los objetivos, el principal era la monitorización de la frecuencia cardiaca en estado de reposo, y se ha conseguido implementar de manera satisfactoria. Igualmente, la comunicación, almacenamiento y visualización de dichas frecuencias eran también objetivos bastante importantes necesarios para hacer una aplicación funcional y acorde a lo que se espera de un trabajo fin de grado. Todo ello ha sido realizado con éxito. Y además, se ha implementado una funcionalidad a mayores de las que se consideraban en un principio: el almacenamiento y visualización en un mapa las frecuencias monitoriza-

das. Esto puede tener una futura aplicación, como se comentará en las líneas de trabajo futuras a continuación.

Como conclusión general de lo que ha sido este trabajo para mí personalmente, diría que me ha servido para aumentar la seguridad en mí mismo y ayudarme a poder afrontar mejor futuros retos que me proponga, ya que con dedicación y ganas prácticamente todo se puede llegar a lograr, independientemente de las condiciones en las que te encuentres.

8.1. Líneas de trabajo futuras

En esta sección se presentan una serie de funcionalidades que mejorarían las apps. Algunas de ellas son totalmente nuevas, mientras que otras son mejoras de lo que está actualmente implementado. Éstas son las siguientes:

- **Análisis de los datos:** esta nueva funcionalidad sería muy conveniente de cara al mercado y a la experiencia de usuario. Se podrían implementar algoritmos de machine learning sobre las frecuencias, lo cual podría ayudar a la detección de patologías cardiovasculares como las arritmias. También se podría crear un mapa de calor donde se viese en que sitios se monitoriza un valor más alto de la frecuencia cardiaca en reposo. Esto podría indicar en que lugares se tiene un mayor estrés (como por ejemplo en casa, en el trabajo, etc). Éstos son solo algunos ejemplos de lo que se podría hacer al añadir este campo de aplicación a las apps actuales.

- **Aviso a emergencias:** esta funcionalidad consiste en incluir en las apps un acceso directo al teléfono de emergencias, para llamar si se detecta alguna anomalía. Esto va de la mano con lo anterior, ya que para recomendar que el usuario llamase a emergencias se necesitaría un análisis prolongado de las frecuencias que fuese monitorizando.

- **Login:** una mejora importante sería incluir un formulario de registro y login para que el usuario viese solo los datos de su cuenta.

- **Almacenamiento en un servidor externo:** actualmente los datos se almacenan en una base de datos local, como ya fue explicado en un capítulo anterior. Esto podría ser mejorado con relativa sencillez. Mediante el uso de un servidor externo se aseguraría que no se perdiesen los datos. Aunque el usuario borrase los datos de la app o la desinstalase, los datos seguirían disponibles y al volver a iniciarla seguirían ahí.

- **Saturación de oxígeno:** este otro indicador puede ser muy útil desde el punto de vista sanitario, para ampliar el rango de enfermedades que se pudiesen detectar, de nuevo con análisis de datos. Otro parámetro muy interesante se trata de la presión arterial, capaz de ser monitorizada en algunos smartwatches del mercado actual. Sin embargo, actualmente las APIs disponibles para estos indicadores no son igual

de fáciles de utilizar que las de la frecuencia cardiaca, ya que la mayoría de marcas poseen su propio SDK, que en la mayoría de ocasiones no es público. Seguramente en un futuro será más fácil utilizar estos sensores.

Lo anterior resume las principales mejoras que podrían incluirse en las apps para ser más funcionales, y ser capaz de competir con las que desarrollan las grandes empresas tecnológicas. No obstante, implementar cada una de ellas necesitaría mucha inversión en tiempo y dinero, y disponer de un gran equipo de desarrolladores, a un nivel ya más empresarial. Otras mejoras que no se han puesto incluirían la internacionalización de la app o una interfaz de usuario lo más amigable y adaptada posible al presente.

Parte III

Manuales de la Aplicación

Capítulo 9

Manual de Instalación

En este capítulo se verá como instalar ambas apps desde cero. Para ello se deberán seguir una serie de pasos que se describen a continuación.

9.1. Instalación del entorno y clonar el repositorio

Lo primero que se tiene que hacer para poder instalar las apps es instalar Android Studio, el IDE con el que se ha trabajado durante todo el proyecto y donde se importará posteriormente el proyecto. Para ello, se debe disponer de un ordenador con las siguientes características, dependiendo del sistema operativo:

- **Windows**
  - Microsoft Windows 7/8/10 (32-bit o 64-bit)
  - 3 GB de RAM mínimo, se recomiendan 8 GB de RAM (más 1 GB para el Emulador de Android)
  - 2 GB de espacio de disco disponible mínimo, se recomiendan 4 GB (500 MB para el IDE más 1.5 GB para Android SDK e imagen de sistema de emulador)
  - Resolución de pantalla mínima de 1280 x 800
- **Mac OS**
  - Mac OS X 10.10 (Yosemite) o superior, hasta 10.13 (High Sierra)
  - 3 GB de RAM mínimo, se recomiendan 8 GB de RAM (más 1 GB para el Emulador de Android)
  - 2 GB de espacio de disco disponible mínimo, se recomiendan 4 GB (500 MB para el IDE más 1.5 GB para Android SDK e imagen de sistema de emulador)
  - Resolución de pantalla mínima de 1280 x 800
- **Linux OS**

- Escritorio GNOME o KDE. Probado en Ubuntu 14.04 LTS, Trusty Tahr (distribución de 64 bits capaz de ejecutar aplicaciones de 32 bits)
- Distribución de 64 bits capaz de ejecutar aplicaciones de 32 bits
- Biblioteca C GNU (glibc) 2.19 o posterior
- 3 GB de RAM mínimo, se recomiendan 8 GB de RAM (más 1 GB para el Emulador de Android)
- 2 GB de espacio de disco disponible mínimo, se recomiendan 4 GB (500 MB para el IDE más 1.5 GB para Android SDK e imagen de sistema de emulador)
- Resolución de pantalla mínima de 1280 x 800

Suponiendo que se cuenta con un ordenador de dichas características, el siguiente paso es descargar Android Studio desde su página oficial. A continuación se instala y se ejecuta.

Una vez ejecutado, para importar el proyecto GitHub se debe seleccionar la opción "Check out from Version Control", y seleccionar "Git". Esto abrirá una ventana donde se debe introducir en el campo "Git URL" la siguiente URL:

`https://github.com/Alberto-Ruiz-Alejandro/TFG.git`

A continuación se selecciona "Clone" y ya se tendría el proyecto clonado en Android Studio.

9.2. Instalación de la app del teléfono

Para instalar la app del smartphone, es necesario disponer de uno que tenga una versión de Android igual o superior a 8.0 y tener al menos 20mb libres de almacenamiento.

Acto seguido, se debe de configurar el móvil para admitir la depuración por USB (hay otras maneras de instalarse, pero ésta es la más fácil). Los pasos que hay que seguir para lograr lo anterior se describen a continuación. Si ya tiene las opciones de desarrollador activadas, ve directamente al paso 4, mientras que si ya tenía activada la depuración puede saltarse todo lo siguiente.

1. Navegar a la configuración del teléfono.
2. Seleccionar la opción "Acerca del teléfono"
3. Pulsar siete veces en la opción "Número de compilación". Estó mostrará un mensaje indicando que se han desbloqueado las opciones de desarrollador
4. Selecciona opciones de desarrollador y activa la depuración por USB.

Una vez hecho esto, conecta por USB el teléfono al ordenador, y abre el proyecto en Android Studio. Automáticamente se reconocerá el dispositivo, y en la barra de arriba podrá seleccionar tanto el proyecto como el dispositivo. La figura 9.1 indica lo necesario para ejecutar la app. Se debe de seleccionar mobile, nuestro dispositivo, y ejecutar la app. Esto instalará y ejecutará la app en nuestro móvil.

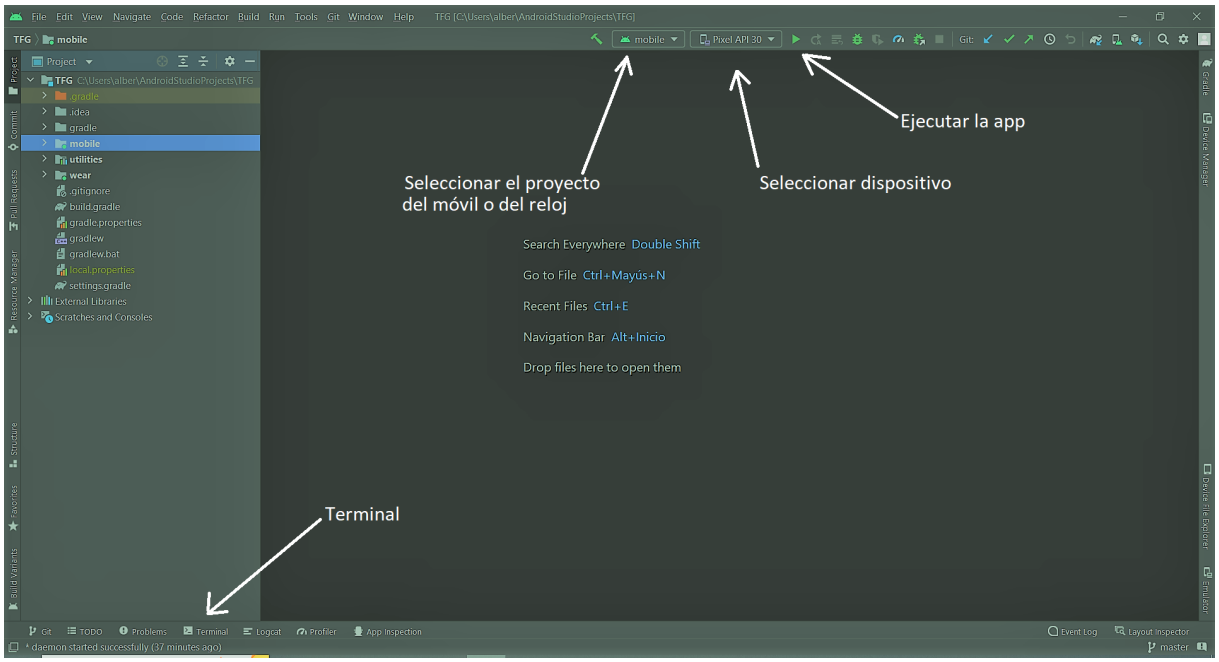

Figura 9.1: Instalación de la app en el móvil

9.3. Instalación de la app del reloj

Para la instalación de la app en el reloj, los pasos descritos anteriormente para activar la depuración son análogos. Lo único que cambia es que en este caso se debe activar la depuración por Wifi.

Una vez hecho esto, abrimos el proyecto en Android Studio y vamos al terminal, indicado en la figura 9.1. Debemos asegurarnos de que tenemos conectados el ordenador y el reloj a la misma red Wifi. A continuación, seguimos los siguientes pasos:

1. Ir a la configuración Wifi del reloj. Pulsamos en la red Wifi a la cual estamos conectados y copiamos la dirección IP que sale. Esto servirá más adelante.

2. Navegar al directorio donde se encuentra lo que se conoce como "platform tools". Típicamente se encuentra en la siguiente ruta:
   \Users\<usuario>\AppData\Local\Android\Sdk\platform-tools
   Por lo tanto, en el terminal introducimos el comando *cd* seguido de la ruta anterior.

3. Introducir el comando siguiente, el cual finalmente establece la conexión entre el ordenador y el reloj. Para ello, introducimos la dirección IP del reloj que copiamos anteriormente.
   adb connect <direccion_IP>

Capítulo 10

Manual de Usuario

Este es el capítulo final de la memoria. Tras haber instalado ambas apps de manera correcta (ver capítulo anterior), lo único que queda es utilizarla. Para ello, se presentan a continuación dos breves tutoriales de como usar tanto la app del reloj como la del móvil.

10.1. Manual de la app del smartwatch

Una vez se inicia la aplicación, se encuentra la pantalla que se ve en la figura 10.1.(a). Si se pulsa el botón play, se inicia la monitorización, como indica el mensaje. Cuando se inicia la monitorización, se visualiza la pantalla de la figura 10.1.(b). En ella se puede apreciar lo siguiente:

(a) Pantalla de inicio (b) Pantalla durante la monitorización (c) Pantalla final

Figura 10.1: Pantallas presentes en la aplicación del smartwatch

- Un mensaje en la parte de arriba de la pantalla. En él se muestra una de las siguientes opciones:
  - "Comprobando reposo": para empezar a monitorizar es necesario estar en reposo al menos 10 segundos. Por ello, mientras se está midiendo dicho tiempo se indica ese mensaje.

  - "Estás en reposo"/"No estás en reposo": cuando se está midiendo la frecuencia cardiaca, es porque se está en reposo. Esto se indica con el primero de los mensajes. En caso de dejar de estar en reposo, cambia la etiqueta a la segunda opción. Cuando no se ha medido todavía una tanda de medidas de manera satisfactoria, cambia de nuevo a la primera etiqueta, ya que se tienen que volver a esperar de nuevo los 10 segundos.
  - Minutos restantes para la monitorización: como se ha comentado anteriormente en la memoria, esta aplicación está diseñada para ejecutarse durante un tiempo prolongado en segundo plano. Por ello, se realiza una tanda de medidas cada media hora. Si el usuario se encuentra en el periodo entre una medida y otra, esta etiqueta muestra los minutos restantes para una nueva monitorización.

- Indicador de la frecuencia cardiaca: cuando se está en reposo, junto con la etiqueta anterior que indica que se está en reposo se muestra el valor de la frecuencia cardiaca medida en tiempo real. Cuando no se está monitorizando la frecuencia muestra simplemente "–".

- Botón pausa: por último, se tiene el botón con el que se puede pausar la monitorización.

Si se pausa la monitorización, se llega a la última pantalla disponible en la app, que se encuentra en la figura 10.1.(c). En ella podemos, o reanudar la monitorización, volviendo a la segunda pantalla descrita, o pararla del todo, y volver así a la primera pantalla descrita.

10.2. Manual de la app del móvil

La app del teléfono móvil es la que tiene un mayor interés para el usuario, ya que es capaz de mostrar diferente información sobre las frecuencias que se han medido. La navegación en la app se controla mediante una barra inferior con cuatro botones, correspondientes a las cuatro funcionalidades principales que tiene la app. Las figuras 10.4 y 10.7 muestran las 4 pantallas correspondientes a estas funcionalidades, que se explican a continuación:

- **Ver última frecuencia:** es lo que se muestra al iniciar la app por defecto. En dicha pantalla se puede ver la frecuencia media de la última tanda medida, junto con su fecha y su ubicación (cuando ésta existe, si no se indica que es desconocida).

- **Historial:** es la funcionalidad principal de la app. Al pulsar en "Historial" en la barra de navegación se visualizan los siguientes gráficos:

  - Última semana: se muestra la media de las frecuencias monitorizadas en la última semana.
  - Último día: se muestran las frecuencias medidas durante el último día. En este caso, se muestran las frecuencias medias por hora.

Figura 10.2: Inicio

Figura 10.3: Historial

Figura 10.4: Pantallas inicio e historial de la app del móvil

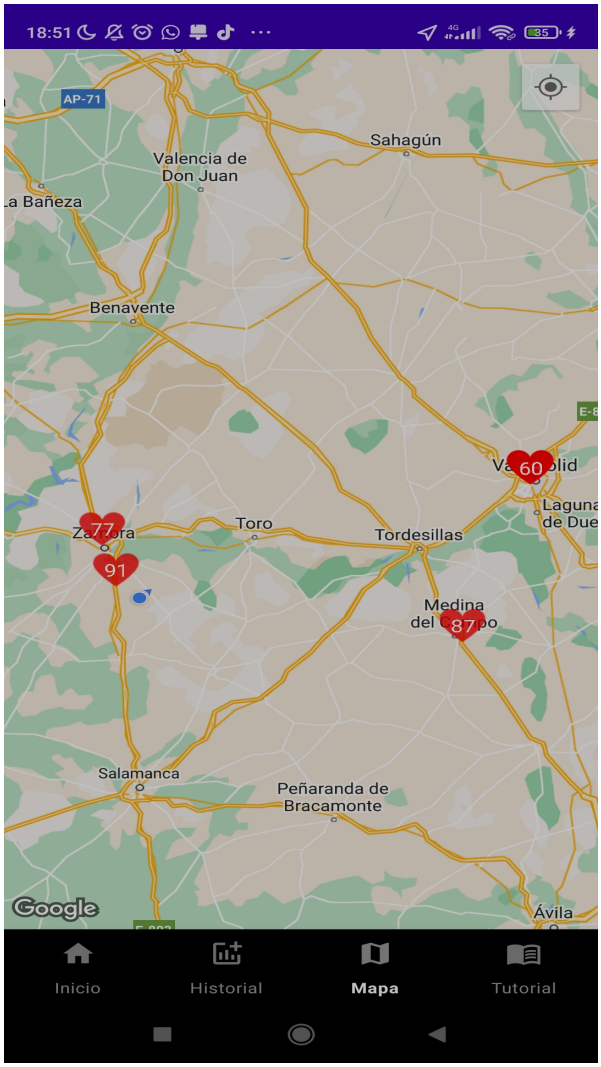

Figura 10.5: Mapa de frecuencias

Figura 10.6: Tutorial

Figura 10.7: Pantallas mapa y tutorial de la app del móvil

  - Último mes: se muestran las frecuencias medidas durante el último mes. En este caso, se muestran las frecuencias medias por día.
  - Último año: se muestran las frecuencias medidas durante el último año. En este caso, se muestran las frecuencias medias por mes.

  Además, en el caso de día, mes y año, si se pulsan los botones "+DÍAS", "+MESES" o "+AÑOS" se muestra una pantalla donde se puede seleccionar el día, mes o año respectivamente y ver las frecuencias del periodo seleccionado.

- **Mapa:** cada frecuencia que se monitoriza es almacenada junto con su ubicación. Si se pulsa "Mapa" en la barra de navegación se accede a un mapa donde se visualiza la ubicación actual del usuario (si está disponible) y un marcador en cada ubicación donde fue medida la frecuencia, junto con su valor.

- **Tutorial:** por último, se tiene un breve tutorial también dentro de la app, por si fuese necesario. Si se pulsa "Tutorial" en la barra de navegación se accede a un breve tutorial de ambas apps. En él se describe de manera resumida lo de este capítulo.

Webgrafía

[1] *Android*. [Online]. Última vez visitado: 24/04/2023. URL: https://es.wikipedia.org/wiki/Android.

[2] *Android Developers*. [Online]. Última vez visitado: 30/04/2023. URL: https://developer.android.com/.

[3] *Canva*. [Online]. Última vez visitado: 05/05/2023. URL: https://www.canva.com/.

[4] *Crear, restringir y usar una clave de API en Google Maps Platform*. [Online]. Última vez visitado: 25/03/2023. URL: https://developers.google.com/maps/documentation/embed/get-api-key?hl=es-419.

[5] *Cómo agregar un mapa usando la SDK de Maps para Android*. [Online]. Última vez visitado: 25/03/2023. URL: https://developers.google.com/maps/documentation/android-sdk/map?hl=es-419.

[6] *Cómo configurar un proyecto de Android Studio para utilizar el SDK de Maps para Android*. [Online]. Última vez visitado: 25/03/2023. URL: https://developers.google.com/maps/documentation/android-sdk/config?hl=es-419.

[7] *Desarrollo de aplicación de asistencia remota para personas con necesidades especiales*. [Online]. Última vez visitado: 20/04/2023. URL: https://idus.us.es/bitstream/handle/11441/38499/TFG%20-%20Tomas%20Acosta%20Almeda.pdf?sequence=1&isAllowed=y.

[8] *Desarrollo iterativo y creciente*. [Online]. Última vez visitado: 20/02/2023. URL: http://aprendiendocositasdelsoftware.blogspot.com/p/desarrollo-iterativo-y-creciente.html.

[9] *Draw.io*. [Online]. Última vez visitado: 27/04/2023. URL: https://app.diagrams.net/.

[10] *Enciclopedia online Wikipedia*. [Online]. Última vez visitado: 18/04/2023. URL: https://es.wikipedia.org/wiki/Wikipedia:Portada.

[11] *Guía de arquitectura de apps*. [Online]. Última vez visitado: 08/04/2023. URL: https://developer.android.com/topic/architecture?hl=es-419.

[12] *Guía de inicio rápido del SDK de Maps para Android*. [Online]. Última vez visitado: 25/03/2023. URL: https://developers.google.com/maps/documentation/android-sdk/start?hl=es-419.

[13] *History of the Android Logo: when and how the logo was created.* [Online]. Última vez visitado: 24/04/2023. URL: https://medium.com/the-curiosity/history-of-the-android-logo-8d9cefd80f44.

[14] *How smartwatches can save lives.* [Online]. Última vez visitado: 20/04/2023. URL: https://www.sacyr.com/en/-/como-los-smartwatches-pueden-salvar-vidas.

[15] *La importancia de las pruebas de software.* [Online]. Última vez visitado: 13/04/2023. URL: https://www.unir.net/ingenieria/revista/pruebas-software/.

[16] *Las enfermedades cardiovasculares suponen la primera causa de muerte en España.* [Online]. Última vez visitado: 20/04/2023. URL: https://www.saludymedicina.org/post/las-enfermedades-cardiovasculares-suponen-la-primera-causa-de-muerte-en-espana.

[17] *Monitorización y recomendación de ejercicio físico a través de una pulsera/reloj inteligente.* [Online]. Última vez visitado: 20/04/2023. URL: https://riuma.uma.es/xmlui/bitstream/handle/10630/23344/Pardal%20Mart%C3%ADn%20Miguel%20Memoria.pdf?sequence=1.

[18] *MpAndroidChart: a powerful and easy to use chart library for Android.* [Online]. Última vez visitado: 10/04/2023. URL: https://github.com/PhilJay/MPAndroidChart.

[19] *Patrón de diseño Observer en Java.* [Online]. Última vez visitado: 10/04/2023. URL: https://gustavopeiretti.com/patron-de-diseno-observer-en-java/.

[20] *Pebble (reloj).* [Online]. Última vez visitado: 20/02/2023. URL: https://es.wikipedia.org/wiki/Pebble_%28reloj%29.

[21] *Stackoverflow.* [Online]. Última vez visitado: 10/05/2023. URL: https://stackoverflow.com/.

[22] *Sueldo medio desarrollador Android en España.* [Online]. Última vez visitado: 25/01/2023. URL: https://www.glassdoor.es/Sueldos/desarrollador-android-sueldo-SRCH_KO0,21.htm.

[23] *Tarifa de luz por hora en España.* [Online]. Última vez visitado: 25/01/2023. URL: https://tarifaluzhora.es/.

[24] *Wear OS.* [Online]. Última vez visitado: 27/04/2023. URL: https://es.wikipedia.org/wiki/Wear_OS.

[25] *Youtube.* [Online]. Última vez visitado: 25/04/2023. URL: https://www.youtube.com/.