Universidad de Cantabria

Facultad de Ciencias

Desarrollo de un reproductor de música Android para vehículos

Development of an Android Music Player for Vehicles

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Marcos Fernández Alonso
Director: Alfonso de la Vega Ruiz
Codirector: Brian Sal Barria
Convocatoria: Septiembre - 2023

Agradecimientos

Quisiera expresar mi más sincero agradecimiento a todas las personas que han sido parte fundamental en la realización de este Trabajo de Fin de Grado, que representa un hito importante en mi trayectoria académica y personal.

En primer lugar, tengo que mencionar el apoyo incondicional de mi familia, cuyo amor y aliento me han motivado en todo momento. Sus palabras de ánimo y comprensión han sido mi mayor fortaleza y han contribuido en gran medida a alcanzar este logro.

Un agradecimiento especial va dirigido a mis tutores, cuya guía, paciencia y orientación han sido esenciales en cada etapa de este proyecto. Sus valiosos consejos y la disposición para aclarar mis dudas han sido un faro en el proceso de investigación y redacción.

No puedo dejar de mencionar a la Universidad de Cantabria por su dedicación y compromiso en impartir una educación de calidad y pública, además de su constante apoyo en mi desarrollo como estudiante. Sus conocimientos y enseñanzas han sido fundamentales para alcanzar este logro.

Este trabajo no solo representa mi esfuerzo, sino también el resultado del trabajo conjunto de muchas personas que creyeron en mí y me brindaron su apoyo. A todos mis amigos y compañeros que compartieron conocimientos, debates y risas en este viaje, ¡gracias por hacerlo inolvidable!

En última instancia, este TFG es un reflejo de todas las conexiones y aprendizajes que he acumulado durante mi tiempo en la universidad. A cada persona que ha formado parte de mi experiencia académica y personal, les estoy profundamente agradecido.

Sin ustedes, este logro no habría sido posible. Gracias de corazón.

Resumen

La presente iniciativa abarca la adaptación de una aplicación de reproducción de música existente para Android. El objetivo principal de este proyecto es aplicar ingeniería del software para adaptar dicha aplicación al uso en el mundo de la automoción. Este proceso de adaptación se ejecuta con la intención primordial de preservar la totalidad de la funcionalidad existente de la aplicación, siempre y cuando esta no dificulte a un posible conductor el uso de la aplicación.

A su vez, en el proyecto se propone la introducción de funcionalidades innovadoras con el propósito de enriquecer la experiencia de uso por parte del conductor sin imponer una carga cognitiva excesiva. En términos de desarrollo de software, el marco general de la arquitectura original de la aplicación se ha mantenido escrupulosamente a lo largo del desarrollo de estas nuevas características.

Además, la integración de estos atributos se lleva a cabo con la intención de coexistir con la filosofía de diseño original, de tal manera que se permita llevar estas funcionalidades a la aplicación original sin tener que hacer grandes modificaciones. Es importante señalar que intentar comprender la lógica que subyace en una aplicación preexistente supone un desafío, especialmente cuando hay una escasez de documentación, el tamaño del proyecto es considerable y la comunicación y arquitectura no son triviales.

En conclusión, este proyecto busca demostrar cómo podemos usar la ingeniería del software para la adaptación de una aplicación de gran envergadura, aún cuando en un principio se carezca de información detallada sobre su funcionamiento interno.

Palabras clave

Adaptación, Refactorización, Desarrollo Software, Integración Continua, Patrones de diseño, Desarrollo ágil, Algoritmo aleatorio, Automatización de Integración, Calidad Software, Ingeniería de Requisitos, Desarrollo Aplicación Android, Reproductor, Coche, Música.

Abstract

The present initiative encompasses the adaptation of an existing music playback application for Android. The main objective of this project is to apply software engineering to tailor the application for use in the automotive world. This adaptation process is carried out with the primary intention of preserving the entirety of the application's existing functionality, as long as it does not hinder a potential driver's use of the application.

Furthermore, the project proposes the introduction of innovative features with the purpose of enhancing the user experience for the driver without imposing an excessive cognitive load. In terms of software development, the overall framework of the original application's architecture has been meticulously maintained throughout the development of these new features.

Additionally, the integration of these attributes is carried out with the intention of coexisting with the original design philosophy, allowing these functionalities to be brought into the original application without requiring major modifications. It is important to note that attempting to understand the underlying logic of a pre-existing application presents a challenge, especially when there is a shortage of documentation, the project size is considerable, and communication and architecture are non-trivial.

In conclusion, this project aims to demonstrate how we can use software engineering to adapt a large-scale application, even when detailed information about its internal functioning is initially lacking.

Keywords

Adaptation, Refactoring, Software Development, Continuous Integration, Design Patterns, Agile Development, Random Algorithm, Integration Automation, Software Quality, Requirements Engineering, Android Application Development, Car, Music.

Índice

Índice de figuras

Índice de tablas

1. Introducción

1.1. Contexto

Al crear y desarrollar aplicaciones, a menudo se busca que abarquen una amplia gama de funciones, pero a veces se descuida su público y contexto concreto. Esta evolución ha causado que se generen diferentes tendencias, entre ellas podemos encontrar la priorización de la estética sobre la eficacia o funcionalidad sobre usabilidad. En este punto de evolución tecnológica, las aplicaciones móviles son fundamentales en la vida diaria. Sin embargo, es fácil caer en la trampa de la complejidad al priorizar la estética sobre la eficacia. La proliferación de aplicaciones con múltiples funciones puede resultar abrumadora, especialmente en situaciones como conducir.

Este proyecto surgió de la búsqueda de un equilibrio entre la funcionalidad avanzada, y específica, y la simplicidad en el diseño de aplicaciones móviles, especialmente en un entorno tan sensible y crítico como la conducción. La idea original fue tomar una aplicación de reproducción de música, una herramienta común y familiar para muchos, y adaptarla para su uso en el entorno de un automóvil.

El proyecto se enfocó buscando la convivencia de simplicidad y eficacia en el diseño de aplicaciones móviles, ofreciendo una experiencia tecnológica avanzada en automóviles que prioriza la seguridad y practicidad.

Durante el presente informe no solo se documenta el proceso de desarrollo y las funcionalidades agregadas a la aplicación de reproducción de música, sino que también se exploran las decisiones de diseño, investigación e implementación cuidadosa que se han llevado a un producto que busca la simplicidad en un mundo cada vez más complejo.

1.2. Objetivo

El objetivo de este proyecto ha sido adaptar una aplicación móvil de reproducción de música existente, simplificando su diseño y funcionalidad para hacerla adecuada para su uso en el entorno de los automóviles. El enfoque principal radicó en optimizar la experiencia del usuario al ofrecer una interacción mínima, intuitiva y segura durante la conducción, sin comprometer las características esenciales que hacen de una aplicación de música una herramienta valiosa.

El objetivo mencionado comprende dos aspectos cruciales, como son la simplicidad y funcionalidad, y busca un equilibrio entre ellos. La simplicidad se logró mediante la mejora de la navegabilidad de la aplicación. A su vez, con las nuevas funcionalidades, se buscó ofrecer la realización de tareas no triviales con una o dos pulsaciones, manteniendo así el equilibrio mencionado anteriormente.

Para alcanzarlo, se siguió un proceso de desarrollo software. Dicho desarrollo no comenzó desde cero, ya que se realizó una selección y posterior refactorización de una aplicación de reproducción de música genérica ya existente. Durante esta refactorización, se siguió un proceso de ingeniería de software, que incluye análisis de requisitos, diseño, implementación y pruebas iterativas.

En resumen, el objetivo de este proyecto es aplicar un proceso de ingeniería cuidadoso para adaptar una aplicación de reproducción de música. A través de esta adaptación, se busca no solo ofrecer una solución funcional y segura, sino también inspirar una nueva perspectiva sobre cómo las aplicaciones móviles pueden adaptarse y optimizarse para situaciones específicas, con el objetivo de mejorar la experiencia del usuario.

2. Herramientas, tecnologías y materiales usados

2.1. Herramientas

Hoy en día, en los proyectos *Software* se hace necesario un amplio catálogo de herramientas que posibilitan su desarrollo, en esta subsección, se detallan cuáles han sido las principales en el proyecto.

- **Entorno de Desarrollo**: Android Studio [1]. Este es el IDE oficial para desarrollar aplicaciones para Android. Se seleccionó tanto por ser el entorno oficial de Android, como por su velocidad a la hora de compilar y ejecutar las aplicaciones, la versatilidad de dispositivos y que personalmente estaba familiarizado con el entorno.
- **Plugins para Android Studio**: Statistic [2] y Rainbow Brackets [3]. Statistics es un *plug-in* disponible en el *marketplace* de Android que permite analizar la cantidad de clases y líneas de código del proyecto. Es un *plug-in* muy simple que permitió obtener una visión inicial de la aplicación sobre la que se trabajó. Rainbow Brackets es otro *plug-in* del *marketplace* que separa elementos como llaves, paréntesis y otros elementos XML, HTML, entre otros, por colores. Este se ha utilizado para simplificar la lectura del código.
- **Alojamiento del código**: GitHub [4]. Esta plataforma permite a los usuarios alojar sus repositorios de código fuente en línea ofreciendo: control de versiones, gestión de tareas y su propio sistema de integración continua, entre otras. Se seleccionó porque, además de su versatilidad y facilidad de uso, la aplicación inicial tiene su propio GitHub [5] y al seguir utilizando esta plataforma se facilita la posible integración de una de las nuevas funcionalidades a la versión original.
- **Control de Versiones**: Git [6]. Esta herramienta de control de versiones de código abierto, permite mantener un proyecto organizado y documentar el proceso de desarrollo. Dado su fácil integración con GitHub se ha seleccionado como herramienta de control de versiones.
- **Integración Continua**: GitHub Actions [7]. Este es un servicio de integración y entrega continua, lo que permite automatizar pruebas y otros procesos. Este está integrado directamente en la plataforma de GitHub. Fue seleccionado dado que es nativo de GitHub, lo que ofrece una mayor facilidad de uso, pero también seguridad y escalabilidad, dado que va a ser más fácil de mantener y no habrá problemas de versiones ni necesidad de mantener un servidor externo.
- **Calidad de Código**: SonarCloud [8]. Esta plataforma analiza el código de un proyecto de manera estática y determina su calidad. Se seleccionó frente a otros como sonarQube, puesto que ofrece una solución en la nube en forma de servicio, lo que significa que no es necesario ni configurar ni mantener una instancia local. Además de su gran compatibilidad con GitHub y GitHub Actions.
- **Diseño de Diagramas**: yEd [9]. Es una herramienta de creación y representación de grafos, diagramas, mapas mentales, entre otros. Esta ha sido la herramienta escogida por encima de otras dada su sencillez de uso y gran variedad de opciones de exportación (png, jpg, svg, pdf, entre otros).

2.2. Tecnologías

Las tecnologías utilizadas a lo largo del proyecto han sido:

- **Lenguaje de Programación**: Java [10]. Este es un lenguaje de programación de alto nivel orientado a objetos. La aplicación finalmente elegida para su adaptación fue desarrollada en Java, por lo que se ha mantenido este lenguaje durante la adaptación.
- **Base de datos**: SQLite [11]. Este es un sistema de gestión de bases de datos relacional. Al igual que el lenguaje, el motor de base de datos de la aplicación era SQLite y así se ha mantenido.
- **Plataforma de Desarrollo**: Android. Este es uno de los sistemas operativos más usados en el teléfonos móviles. Al ser una aplicación móvil ya desarrollada en Android se ha mantenido igual, proporcionando una sólida base para la implementación de las funcionalidades clave de la aplicación, al tiempo que ha permitido la adaptación de la interfaz de usuario para su uso en dispositivos móviles y, en última instancia, en el entorno de los automóviles.
- **Plataforma de Pruebas**: Espresso [12], dispositivos emulados Android Studio [1] y móviles reales. Espresso es un conjunto de herramientas de prueba para el desarrollo de aplicaciones Android que permite a los desarrolladores automatizar y realizar pruebas de interfaz, por ello se ha utilizado para las pruebas de interfaz de usuario. Mientras que para las pruebas de integración se han usado dispositivos emulados ofrecidos por Android y para las de aceptación se han utilizado dispositivos reales, incluyendo un *smartphone* y una unidad android para vehículo .

2.3. Materiales

Al partir de un proyecto ajeno, ya se disponía de ciertos materiales y otros se han ido generando.

- **Código Fuente**: El código inicial de la aplicación era abierto y es el que se ha utilizado para la implementación de las funcionalidades y nueva lógica de la aplicación.
- **Control de Versiones**: El cual se puede encontrar en el repositorio de GitHub.
- **Dispositivos Móviles**: Utilizados para realizar pruebas y evaluar el rendimiento en situaciones reales.

3. Análisis de requisitos

La ingeniería de requisitos comprende todo el proceso desde la identificación hasta la verificación de cada requisito. Sabiendo que un requisito es "la descripción de los servicios y restricciones de un sistema de software, es decir, lo que el software debe hacer y bajo qué circunstancias debe hacerlo." [13], podemos dar comienzo a este proceso.

3.1. Requisitos funcionales

Los requisitos funcionales son una parte fundamental en el proceso de ingeniería de software definiendo cómo el sistema debe comportarse y cómo debe interactuar con los usuarios o con otros sistemas. Sirviendo estos como base y guía para desarrollar y diseñar el sistema en cuestión.

Para definir los requisitos, resumidos en la Tabla 1, primero se identificaron los requisitos, a continuación se redactaron en historias de usuario y finalmente se atribuyó un valor a cada uno para permitir su priorización.

Tabla 1: Tabla Requisitos Funcionales

Id del Requisito	Descripción del Requisito	Valor
R.F.01	Como usuario, quiero que el reproductor de música sea capaz de leer la música de los archivos locales de mi dispositivo, para poder acceder y reproducir mi biblioteca de música personal.	10
R.F.02	Como usuario, quiero que cuando añada música a la carpeta por defecto, el reproductor de música añada automáticamente las canciones nuevas a la biblioteca, para que no tenga que realizar una acción adicional para tener acceso a las nuevas canciones.	5
R.F.03	Como usuario, quiero que el reproductor de música muestre las carátulas correspondientes a las canciones y álbumes, para mejorar mi experiencia visual y poder identificar rápidamente la música que deseo reproducir.	8
R.F.04	Como usuario, quiero que al iniciar la aplicación de reproductor de música se muestren cuatro modos: álbum, playlist, artista y modo canciones, para poder navegar y acceder fácilmente a diferentes categorías de música.	10
R.F.05	Como usuario, quiero que la música se almacene automáticamente por autor y álbum/es, basándose en la estructura de carpetas existente, para poder tener una biblioteca organizada y encontrar rápidamente la música de mis artistas y álbumes favoritos.	9
R.F.06	Como usuario, quiero tener la opción de añadir o eliminar canciones de la cola de música que se está reproduciendo en ese momento, ya sea de forma individual, un álbum o un artista completo, para tener control sobre la lista de reproducción y personalizar mi experiencia auditiva.	10

R.F.07	Como usuario, quiero poder crear playlists a partir de las colas de reproducción personalizadas, para poder agrupar y organizar mis canciones favoritas de diferentes géneros o estados de ánimo.	9
R.F.08	Como usuario, quiero que las playlists sean modificables, para poder añadir o eliminar canciones de una playlist existente y adaptarla a mis preferencias musicales cambiantes.	9
R.F.09	Como usuario, quiero que el reproductor de música me permita reproducir, saltar canciones, retroceder, pausar y reanudar la reproducción, para tener un control total sobre la música que se está reproduciendo y poder adaptarla a mis necesidades y gustos.i	10
R.F.10	Como usuario, quiero tener la opción de activar el modo de bucle, para que cuando la lista de reproducción actual termine, se reproduzca nuevamente desde el principio, proporcionando una experiencia de escucha continua y sin interrupciones.	8
R.F.11	Como usuario, quiero poder activar el modo aleatorio en la lista de canciones que estoy escuchando, para que las canciones se reproduzcan en un orden aleatorio y descubrir nuevas canciones y artistas.	8
R.F.12	Como usuario, quiero poder personalizar el tema de la aplicación del reproductor de música, para adaptarlo a mis preferencias visuales y crear una experiencia de usuario personalizada.	5
R.F.13	Como usuario, quiero poder elegir el modo de funcionamiento de la aleatoriedad, para tener control sobre cómo se seleccionan y reproducen las canciones en el modo aleatorio en función del artista y/o álbum.	7
R.F.14	Como usuario, quiero poder detener y saltar canciones desde fuera de la aplicación, utilizando los controles de reproducción disponibles en la pantalla de bloqueo o en la barra de notificaciones, para tener acceso rápido y conveniente a las funciones de reproducción.	9
R.F.15	Como usuario, quiero poder modificar las opciones del ecualizador en el reproductor de música, para ajustar el sonido de acuerdo a mis preferencias y mejorar la calidad de la reproducción de audio.	4
R.F.16	Como usuario, quiero poder buscar canciones, álbumes y artistas por nombre en el reproductor de música, para encontrar rápidamente canciones específicas en mi biblioteca de música.	7
R.F.17	Como usuario, quiero que en el modo playlist la primera playlist que aparezca sea la del modo “Party Mode”. Este modo reproducirá música aleatoria de mi biblioteca y me dará la opción de elegir entre las 10 siguientes canciones que ha elegido para mí.	8
R.F.18	Como usuario, quiero que en el modo álbum el primero que aparezca sea el modo “Save The Album”. Este modo Reproducirá un álbum de un artista aleatorio en el orden original.	8
R.F.19	Como usuario, quiero que si elimino una canción del almacenamiento local quiero que se elimine de mis playlists y biblioteca.	4

R.F.20	Como usuario, quiero que la app tenga una opción para ver las canciones que más escucho y me diga cuantas veces las he escuchado, estén o no ya en mi biblioteca.	6
R.F.21	Como usuario, quiero poder integrar mi aplicación con Spotify y ser capaz de traer mis playlists.	3
R.F.22	Como usuario, quiero poder compartir mis playlists con mis contactos del móvil por WhatsApp.	2
R.F.23	Como usuario, quiero que la aplicación cuente con una sección de preguntas frecuentes y otra con información de la app, como versión, creadores...	6
R.F.24	Como usuario, quiero poder viajar desde el álbum y la canción al artista.	3
R.F.25	Como usuario, quiero poder desactivar la descarga de carátulas con datos para evitar malgastos innecesarios.	1
R.F.26	Como usuario, quiero poder desactivar la opción de mostrar carátulas para optimizar mi app.	1

En la Tabla 1 se mencionan dos modos de reproducción definidos específicamente para su uso en automóviles, y por lo tanto objetivos de gran importancia en este proyecto. Estos son "Save the album" y "Party Mode". El primero de estos debe seleccionar un álbum de la biblioteca aleatoriamente y reproducirlo de principio a fin. La selección debe ser aleatoria en función del artista, con el objetivo de no beneficiar a los artistas con un gran número de álbumes. El segundo modo, que surge de una funcionalidad existente en el reproductor de contenidos Kodi [14], debe crear una *playlist* de canciones aleatorias en función del artista y álbum. Cuando el usuario seleccione una canción de esta, la canción pasará a ser la primera de la *playlist* y las que estaban entre medias de la que sonaba y la seleccionada se eliminarán. Los huecos que dejen las canciones eliminadas se rellenarán por el final de la cola con otras canciones aleatorias. Este modo tiene como objetivo ofrecer al usuario una lista de canciones aleatorias en pantalla sobre la que puede hacer una selección, de forma rápida y sencilla, sin tener que recurrir al habitual pase de canciones una a una en aleatorio hasta que se reproduzca una que sea adecuada. Al acabar de reproducir la *playlist*, esta se volverá a aleatorizar.

3.2. Requisitos no funcionales

Los requisitos no funcionales se centran en cómo debe funcionar el sistema entendiendo, dentro de este "cómo", ámbitos como el rendimiento, escalabilidad o usabilidad, entre otros.

En la Tabla 2 se muestran los requisitos no funcionales definidos para el reproductor con una breve descripción y la categoría a la que pertenecen. Es importante destacar dos de estos, los cuales resultan fundamentales en este proyecto: la usabilidad y la adaptabilidad. La usabilidad está estrechamente relacionada con la simplicidad, uno de los objetivos principales. Por otro lado, la adaptabilidad es esencial, ya que nuestra aplicación será utilizada tanto en dispositivos móviles como en automóviles.

Tabla 2: Tabla Requisitos No Funcionales

Id del Requisito	Descripción del Requisito	Tipo
R.N.F.01	Como usuario, quiero que el reproductor sea fácil de usar, con una interfaz y controles intuitivos. Siendo capaz de reproducir la música en un máximo de dos pulsaciones.	Usabilidad
R.N.F.02	Como usuario, quiero que la aplicación se visualice y función correctamente en cualquier dispositivo Android con una versión 5.0 o superior.	Portabilidad
R.N.F.03	Como usuario, quiero que el reproductor sea eficiente en términos de recursos y tiempo de respuesta, para garantizar una reproducción fluida y sin interrupciones.	Rendimiento
R.N.F.04	Como usuario, quiero que el reproductor sea compatible con una amplia gama de formatos de música populares y dispositivos de almacenamiento externo, para garantizar la versatilidad de la aplicación	Compatibilidad
R.N.F.05	Como usuario, quiero que el reproductor sea estable y confiable, evitando bloqueos o cierres inesperados durante la reproducción de música.	Estabilidad
R.N.F.06	Como usuario, quiero que el reproductor solo tenga permisos de lectura sobre la música para evitar poder eliminarla del dispositivo. A su vez solo necesitará permisos para enviar notificaciones en caso de que el usuario lo desee.	Seguridad
R.N.F.07	Como usuario, quiero que el reproductor se adapte a diferentes tamaños de pantalla y resoluciones, asegurando una experiencia de usuario óptima tanto en dispositivos móviles como en pantallas de mayor tamaño, como las de los coches.	Adaptabilidad

3.3. Consideraciones

En esta sección, se revisarán las consideraciones clave para el diseño y desarrollo de la aplicación. Es esencial comprender estos aspectos para garantizar que la aplicación cumpla con los requisitos funcionales y no funcionales, así como para proporcionar una experiencia de usuario efectiva y segura.

Actores y usuarios

En el caso de esta aplicación solo hay un actor, el propio usuario de la aplicación. Este será quien gestione todo su contenido y será el único responsable del mismo. Como es bastante probable que el usuario final sea la persona que conduce el coche, es esencial que todas las funcionalidades de la aplicación sean accesibles para este contexto.

Seguridad

Dado que la aplicación no es accesible desde el exterior y se centra en la gestión de canciones, las preocupaciones de seguridad se reducen principalmente a cuestiones de permisos a nivel de sistema operativo, como se puede ver en la Tabla 2 en el apartado de seguridad, por lo que resulta esencial garantizar que los permisos estén bien gestionados para proteger la privacidad y la integridad de los datos del usuario.

Derechos de Autor

En la aplicación se permite a los usuarios cargar y gestionar sus propias canciones. Es importante destacar que la aplicación permite reproducir el contenido del usuario y no asume la responsabilidad directa sobre el contenido que se carga en ella.

4. Proceso de selección de la Aplicación

En esta sección se presenta el procedimiento seguido durante la selección de la aplicación base que sirvió como punto de partida para su posterior adaptación. Este proceso surge a partir de la detección de una nueva necesidad, un reproductor de música para el coche suficientemente sencillo para usar mientras conduces con modos de reproducción específicos que faciliten la reproducción de música.

4.1. Proceso de investigación

Al comienzo del proyecto, se propuso partir de una aplicación de reproducción de música ya existente frente a comenzar de cero. Esta decisión se tomó puesto que un reproductor para coche comparte muchas similitudes con uno normal, lo que permite evitar la reinvención innecesaria y lograr una óptima gestión de los recursos disponibles.

Partiendo de esta idea, el punto de partida es la elección del reproductor a analizar. Para ello, se recurrió a F-Droid [15], un repositorio de aplicaciones Android que aloja tanto archivos APK como enlaces a repositorios, donde se encuentra el código fuente de aplicaciones. F-Droid está totalmente enfocado a aplicaciones de código abierto y permite a los usuarios subir sus propias aplicaciones y dar extensas descripciones sobre las mismas, lo cual facilita la búsqueda de una que cumpla una serie de requisitos.

Todas las aplicaciones consideradas fueron seleccionadas de este repositorio, atendiendo a los siguientes criterios:

- **Lenguaje de programación:** Las opciones más comunes en Android son Java y Kotlin. En este caso se buscó que la aplicación estuviese desarrollada completamente con Java por familiaridad.
- **Calidad y estructura del código:** Se priorizó la elección de una aplicación con una estructura lógica y bien organizada en su árbol de carpetas, nomenclatura clara en las clases y uso de una arquitectura reconocida.
- **Usabilidad y compatibilidad:** Cada aplicación evaluada se instaló y probó para verificar su funcionamiento, optimización y usabilidad. Se buscó una aplicación con una curva de aprendizaje no excesiva y una cantidad razonable de funcionalidades. Se estableció como requisito que la aplicación fuese compatible con versiones de Android desde la 5.0 (incluida).
- **Estado del proyecto:** Un requisito adicional fue que el proyecto hubiera tenido actividad en los últimos dos años.

Después de establecer con precisión los criterios de selección, la primera semana del proyecto se enfocó en identificar una aplicación que cumpliera de manera coherente con al menos cuatro de estos requisitos. Esto dio como resultado la selección de tres aplicaciones: Odyssey [5], Unpopular Music Player [16] y Symphony [17].

4.2. Proceso de selección

Para decidir con cual quedarse de las tres aplicaciones candidatas identificadas, se compararon qué requisitos funcionales, definidos en la Tabla 1, estaban implementados y cuáles no. El resultado queda reflejado en la Tabla 3.

Tabla 3: Comparación de cumplimiento de requisitos entre Odyssey, Unpopular Music Player y Symphony.

Id del requisito	Odyssey	Symphony	Unpopular Music Player
R.F.01	Si	Si	Si
R.F.02	No	Si	No
R.F.03	Si	Si	Si
R.F.04	No	No	No
R.F.05	Si	No	No
R.F.06	Si	No	No
R.F.07	Si	No	No
R.F.08	No	No	No
R.F.09	Si	Si	Si
R.F.10	Si	No	Si
R.F.11	Si	Si	No
R.F.12	Si	No	Si
R.F.13	Si	Si	No
R.F.14	Si	Si	Si
R.F.15	Si	No	No
R.F.16	Si	No	No
R.F.17	No	No	No
R.F.18	No	No	No
R.F.19	No	No	No
R.F.20	No	No	No
R.F.21	No	No	No
R.F.22	No	No	No
R.F.23	Si	No	Si
R.F.24	Si	No	No
R.F.25	Si	No	No
R.F.26	Si	No	No

El reproductor Odyssey cumple con diecisiete de los veintiséis requisitos, lo que ofrece un sólido punto de partida. También cumple todos los criterios iniciales mencionados en 4.1. Estos, entre otros, fueron los principales motivos por los que se eligió como aplicación para adaptar.

4.3. Proceso de formación

Tras la elección, se llevó a cabo un análisis más profundo de Odyssey utilizando herramientas como SonarCloud [8] y Statistics [2], con el propósito de evaluar su estado actual en términos de calidad y desempeño. Posteriormente, se buscó profundizar en la estructura, arquitectura, funcionalidad y navegación de la aplicación.

Statistics

Mediante este complemento, se puede visualizar de manera sencilla la envergadura del proyecto. Después de descargarlo desde el mercado de complementos de Android Studio, simplemente se

ejecuta para obtener los datos de la Tabla 4. Esta muestra un total aproximado de 43,000 líneas de código, siendo de particular interés las cerca de 29,000 líneas distribuidas en 138 clases Java.

La Tabla 5 amplía el enfoque sobre la sección Java. En esta se evidencia que aproximadamente la mitad del total de líneas constituye comentarios, lo que reduce la cantidad inicial de alrededor de 29,000 líneas a 15,652 líneas de código real, representando el 54 % del total. Esto se ha considerado algo positivo puesto que en su mayoría están enfocados a explicar el código, lo cual, en un proyecto del cual no se dispone de ningún tipo de documentación, es de agradecer.

Adicionalmente, se destaca la relevancia de dos clases en particular: el servicio de reproducción y la interfaz de la canción en curso. La primera de estas clases alberga una significativa porción de las funcionalidades de la aplicación, abarcando desde la reproducción de música hasta el control de mensajes, estados, saltos y alarmas. La segunda, por su parte, adquiere importancia debido a su alta capacidad de navegación entre distintas vistas y a la posibilidad de cambios de estado asociados con la canción en reproducción.

Extensión	**Clases**	**Tamaño total**	**Líneas de código**
aidl	4	6kB	185
bat	1	2kB	90
gitignore	1	0kB	1
gradle	2	1kB	49
html	1	3kB	92
java	138	1.088kB	28,781
xml	109	472kB	8,730
txt	20	5kB	112
svg	6	420kB	5,132

Tabla 4: Resumen del análisis y estadísticas del proyecto.

SonarCloud

Esta herramienta posibilita efectuar un análisis de calidad y seguridad sobre el código de un repositorio al vincularlo directamente con la cuenta de GitHub dueña del repositorio. A continuación, en la Figura 1, se presenta el resultado obtenido de dicho análisis. Se evidencia que el estado del código, a pesar de superar las 10,000 líneas, no es alarmante. A partir de esta visión global, es posible inferir que la sección relativa a la seguridad presenta deficiencias y, posiblemente, exista una vulnerabilidad importante. Asimismo, resulta preocupante el hecho de que la cobertura sea del 0 %, debido a la ausencia de pruebas en el repositorio.

Archivo Fuente	Total de Líneas	Líneas de Código Fuente	Líneas Comentadas
PlaybackService.java	2108	1085	687
NowPlayingView.java	1563	864	480
OdysseyMainActivity.java	1083	716	170
MusicLibraryHelper.java	901	510	189
OdysseyDatabaseManager.java	893	553	168
GaplessPlayer.java	766	387	258
ArtworkManager.java	625	369	157
FilesFragment.java	578	330	156
MyMusicFragment.java	496	291	122
ArtistAlbumsFragment.java	489	285	121
ListViewItem.java	482	276	134
BulkDownloadService.java	473	334	40
AlbumTracksFragment.java	462	274	116
PlaylistTracksFragment.java	445	244	125
PlaybackServiceStatusHelper.java	443	235	136
ArtworkDatabaseManager.java	440	251	96
OdysseyPlaybackServiceInterface.java	391	303	35
GenericSectionAdapter.java	346	182	104
MediaScannerService.java	344	222	43
TrackModel.java	333	161	124
ArtistsFragment.java	301	172	77
Total de Lineas	28774	15645	8304

Tabla 5: Resumen de Líneas de Código

Figura 1: Análisis general de SonarCloud

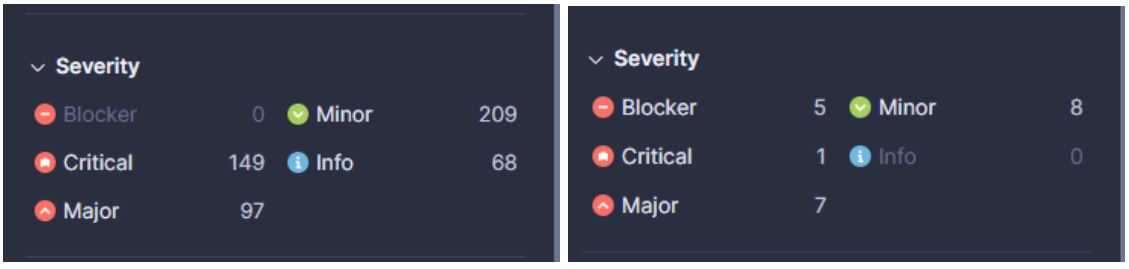
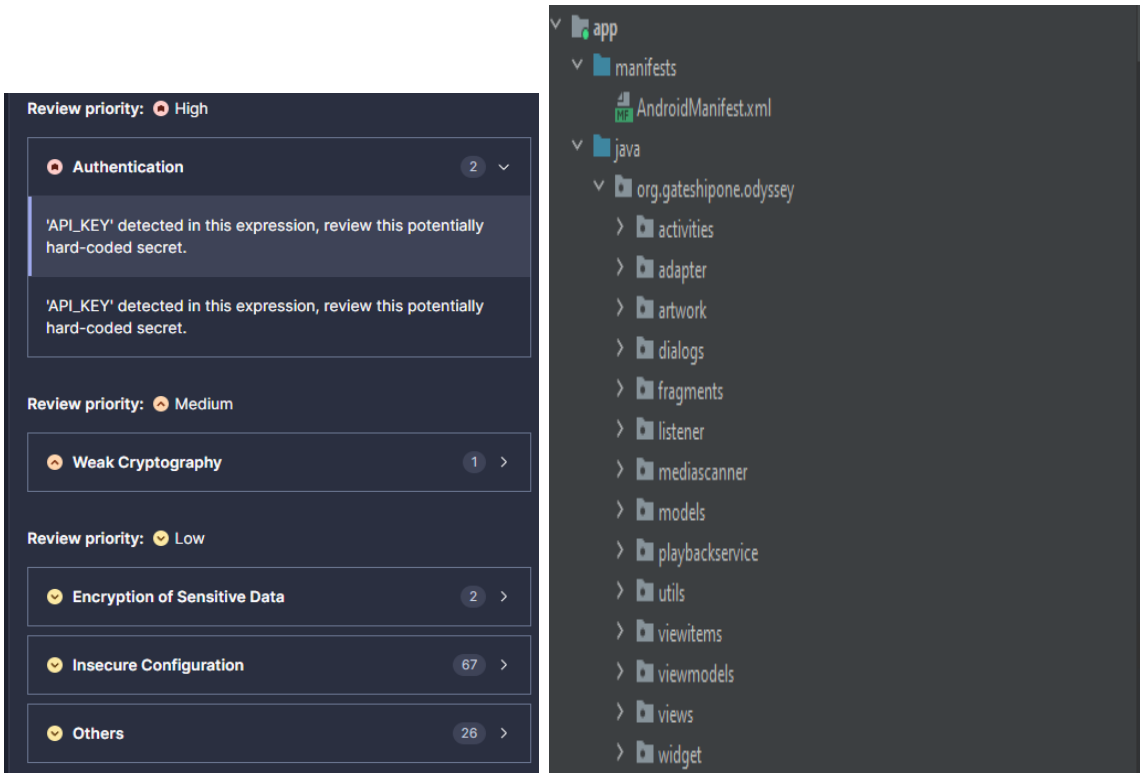


Figura 2: Code Smells y *bugs* encontrados

En la Figura 2 se puede ver en más detalle el análisis de los *bugs* y *code-smells*. En este se cons-

tata que, aproximadamente, la mitad de estos se clasifica en la categoría de menores, mientras que la otra mitad se distribuye equitativamente entre las categorías de mayor importancia y criticidad. Resulta especialmente inquietante la presencia de cinco *bugs* clasificados como bloqueadores y uno crítico, los cuales podrían hacer que la aplicación no siempre funcione como se desea.



(a) Resultado seguridad de SonarCloud **(b)** Estructura inicial de carpetas

Figura 3: Security Spots encontrados por Sonar y estructura inical de carpetas

La Figura 3a muestra los problemas de seguridad. Estos, en su mayoría, poseen un nivel de gravedad considerado bajo, mientras que los más significativos se relacionan con la presencia de dos "API_KEYS.[en] formato de texto plano dentro del código, lo cual conlleva un riesgo significativo para la seguridad. El elemento catalogado con prioridad "medium" se refiere a la generación de un número pseudoaleatorio, aunque esta consideración carece de relevancia en términos de fortalecimiento de la seguridad. Como conclusión del apartado de seguridad habría que considerar modificar que las "API_KEYS" se encuentren en texto plano, los demás aspectos no son cruciales.

A partir de este análisis superficial, se confirma que el código inicial demuestra una calidad considerable, considerando la magnitud del mismo, si bien su principal limitación radica en la ausencia de pruebas funcionales y falta de documentación.

Estudio del funcionamiento

El proceso de formación continuó con la estructura interna de la aplicación. Afortunadamente, como muestra la Figura 3b, las clases están bien estructuradas y organizadas lo que permite identificar fácilmente donde está cada elemento.

Una vez analizada la estructura externa y determinada que arquitectura estaba usándose, se buscó entender más en profundidad cuales eran las clases con mayor peso y como se comunican las diferentes capas de la aplicación. Esto se verá más en profundidad en el apartado de diseño y arquitectura 6.

Finalmente, ante a falta de documentación en el repositorio y la carencia de *tests*, se trató de contactar con el creador de la aplicación para preguntar si disponía de alguno de estos. Tras una semanas se recibió la respuesta de la Figura 4: "*...So many parts are considered legacy and apart of the comments in the code itself there is no further documentation available.* " la cuál indicaba claramente que no había más documentación.

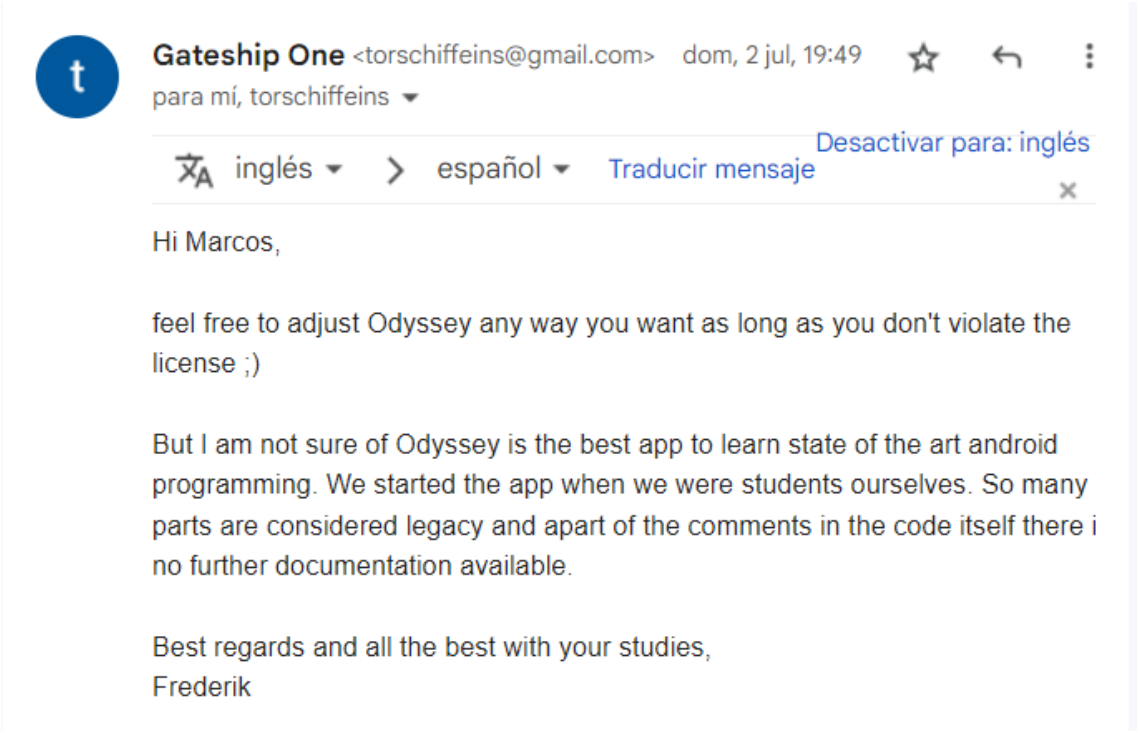


Figura 4: Imagen del correo recibido por el creador de Odyssey.

5. Metodología

En esta sección se describe la metodología iterativa/incremental empleada para el desarrollo, el trabajo realizado en cada iteración o *sprint*, y algunas consideraciones de integración contínua y del seguimiento realizado durante el proyecto.

5.1. Metodología de desarrollo

Una vez seleccionada y analizada la aplicación, empezó la fase de desarrollo. Desde un principio se estableció que el proyecto iba a seguir una metodología de desarrollo incremental. Se escogió esta frente a una metodología tradicional o de cascada, puesto que nos permite ir mejorando la aplicación de manera progresiva y adaptar el ritmo de trabajo. Esto es muy importante, sobretodo cuando se trabaja con una aplicación ajena por primera vez, puesto que no se sabe al 100 % dónde está cada cosa ni dónde se va a tener que modificar o añadir código para cualquier cambio o directamente si el código es completamente funcional. Las iteraciones realizadas pueden, por lo general, dividirse en cuatro etapas

- **Análisis:** en esta etapa se definen los objetivos a realizar en la iteración correspondiente. En este proyecto en concreto se acordaron *milestones* extra que podrían realizarse en caso de que sobrase tiempo. Esto se debió a la escasez de tiempo de la que se disponía, pero también otorgó mayor flexibilidad y margen de error a la hora de infravalorar la capacidad de desarrollo.
- **Implementación:** tras definir los objetivos da comienzo la fase de implementación de la lógica de estos. Durante esta siempre se pensó como incluir estos objetivos manteniendo la estructura y arquitectura original.
- **Pruebas:** terminada la segunda etapa se verifica, con el uso de pruebas de interfaz, unitarias, integración o aceptación, que la aplicación funcione correctamente y que no se haya generado algún *bug* nuevo.
- **Integración y *release*:** finalmente se integra el nuevo código y sus pruebas a la rama principal haciendo uso de las *pull requests*, revisando siempre la calidad del código de las nuevas implementaciones con ayuda de SonarCloud. Y con esto generar una nueva *release*.

A continuación, se describe el trabajo realizado.

5.2. Integración continua

La integración continua es una práctica en el desarrollo de software en la que los cambios en el código son integrados y probados automáticamente de manera frecuente y consistente, a medida que se realizan modificaciones en el proyecto. Cuyo objetivo es detectar errores de manera temprana.

A pesar de que en el proyecto solo había una persona involucrada en el desarrollo se han realizado subidas de código al repositorio cada máximo 3 días. Además por cada *sprint* se ha realizado una *pull request*, exceptuando el primero que por falta de experiencia se realizaron 3, dado que se desconocía que si se volvían a subir los nuevos cambios se actualizaba la *pull request*

y no hacía falta cerrarla y volver a abrirla. En la Figura 5 podemos ver el estado del proyecto tras cuatro iteraciones.

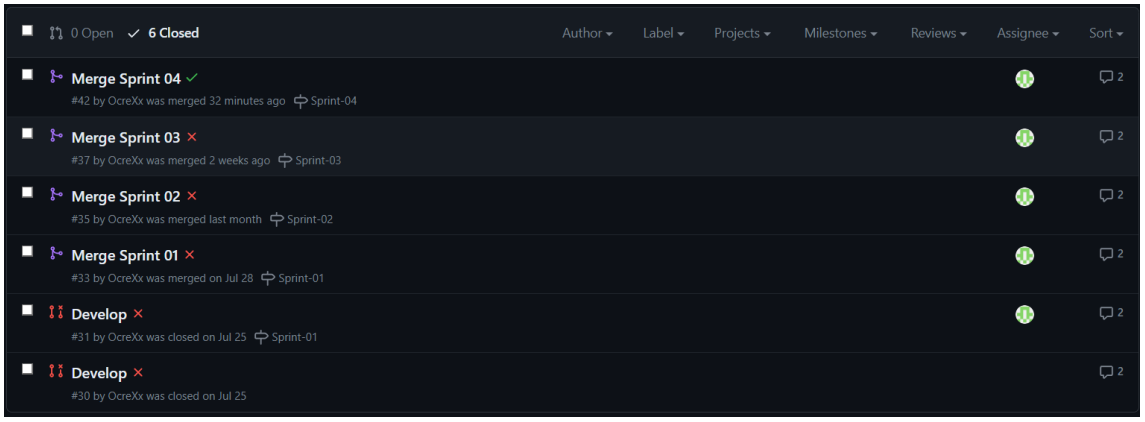


Figura 5: Estado de las pull request tras cuatro iteraciones.

Para llevar acabo una buena integración continua se han establecido usado ciertas herramientas proporcionadas por GitHub [4], dado que todo el desarrollo y seguimiento del proyecto se ha hecho ahí.

GitHub Actions

GitHub Actions [7] es un servicio de integración y entrega continua que nos proporciona GitHub [4]. Este nos permite definir un flujo de trabajo personalizado para automatizar ciertas tareas, como compilar, pasar tests, generar *releases*, entre otros.

Se partió de un flujo de trabajo vacío y este se fue desarrollando a medida que fueron surgiendo necesidades de automatización. La primera necesidad que surgió fue que, al subir los cambios a *develop* o *master*, se lanzase el análisis de Sonar Cloud [8] y ejecutase los *tests* de manera automática. Una vez implementado surgió una nueva necesidad dado que sonar, de base, no es capaz de analizar la cobertura de los tests, para ello habría que añadir *Jacoco* [19] (o cualquier otro proyecto de código abierto que nos permita integrarlo con Sonar), el cual permite analizar la cobertura de los *tests*. Esto se ha dejado como un tique puesto que su coste temporal resultaba alto en comparación con lo que aporta.

La siguiente necesidad que apareció fue automatizar la creación de *releases*. Para ello se realizó un flujo de trabajo que primero compila la aplicación y genera una APK (en este caso sin firmar). Una vez termine de crearla, la sube con ayuda del *action upload artifact* [20] al repositorio. Al acabar esta primera parte del flujo da comienzo la segunda. En esta se crea la *release* con ayuda del *action release* [21]. En este caso como solo hay una persona en el proyecto se ha decidido mantener la carpeta *build*, donde se encuentran las APKs en el .gitignore, lo que hace que tampoco se incluyan en la *release*. Aún así las APKs se quedan guardadas en la ejecución del flujo de trabajo y se pueden descargar desde allí.

Para una visión más en detalle de los *workflows* se pueden ver en el GitHub [22].

5.3. Seguimiento del proyecto

Para ayudar a la organización de los Sprints y poder mantener una integración continua clara se utilizó la herramienta de GitHub Project [23], la cual permite crear *issues*, los cuales se han utilizado para registrar el desarrollo de los requisitos funcionales. Al igual que otras herramientas, como ScrumDesk [24], también permite categorizar las tareas en distintas columnas, en este caso se han usado: *"Todo"* para las que quedan por hacer, *"In Progress"* para las que se están desarrollando ahora y *"Done"* para las que están hechas. Esto se puede apreciar mejor en la Figura 6. A esto se le ha añadido el uso de *milestones* para separar el trabajo de cada *sprint* y distinguir aquellos *issues* que ya estaban implementados previamente a iniciar la adaptación de la aplicación.

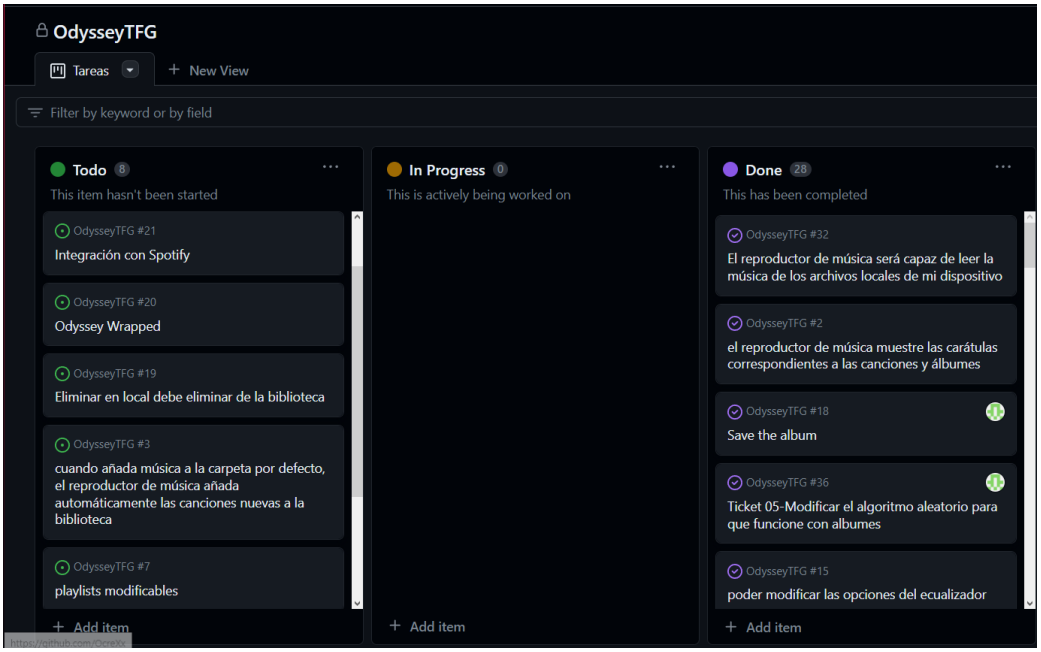


Figura 6: GitHub Project de la aplicación

5.4. Iteraciones realizadas

Con la metodología de desarrollo clara, se estimaron cuatro *sprints* centrados completamente en el desarrollo de las nuevas funcionalidades y adaptación de la aplicación. Estos se detallan a continuación.

5.4.1. Iteración 1

En el primer *sprint*, en la fase de análisis, se definieron dos objetivos. El primero fue crear una nueva ventana o vista inicial, que permitiese al usuario elegir a qué menú de la aplicación quiere acceder, y el segundo fue integrar SonarCloud con GitHub Actions. El primero facilita al conductor acceder al menú que más le interese, porque como muestra la Figura 7, aunque puedes ir a cualquier menú desde la pantalla original, los botones son pequeños y dificultan la navegación. Y el segundo objetivo permite automatizar el análisis de código cuando se hace un cambio en *master* o *develop*.

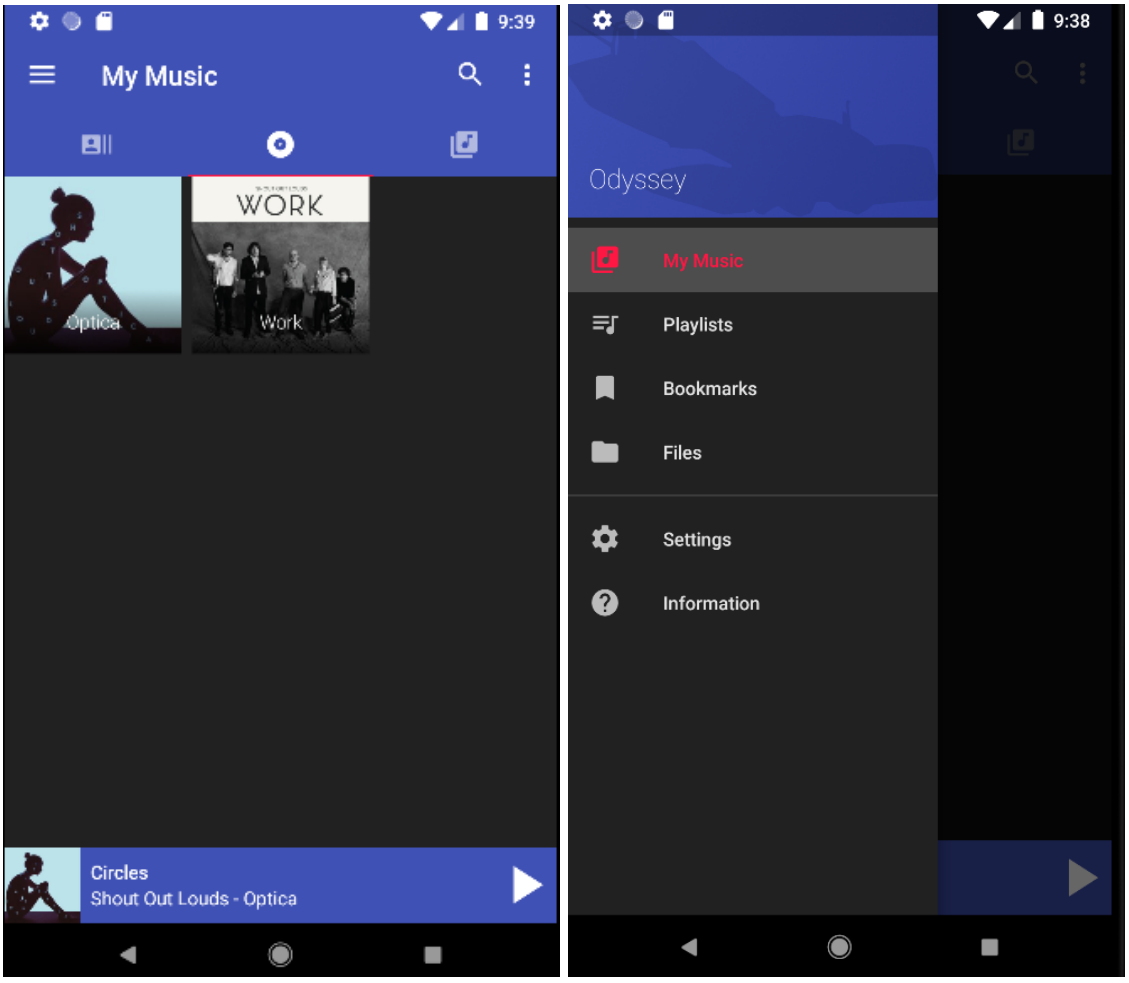


Figura 7: Menú inical y menú lateral original

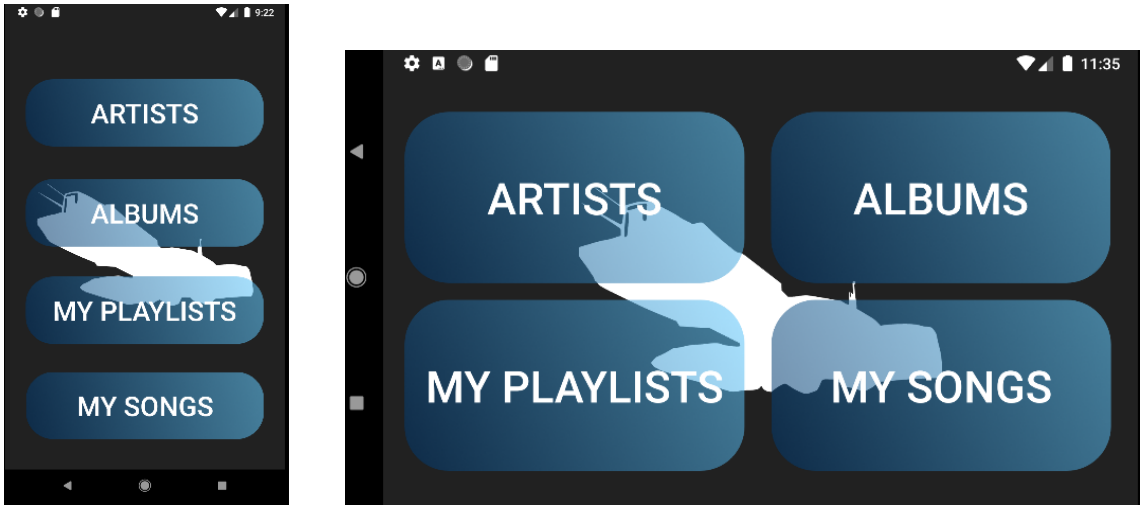
Pasado el análisis, dio comienzo el desarrollo. Con el nuevo menú 8a, 8b se permite una navegabilidad mucho mas sencilla y eficaz, la cual facilita al usuario usar el reproductor en caso de que estuviese conduciendo. Además, esta pantalla con el logo en grande le da cierta entidad a la aplicación.

La otra parte que se realizó fue integrar SonarCloud con el proyecto a través de GitHub Actions. Esto no dio muchos problemas, exceptuando que debido a la extensión del código, se tuvieron problemas de tamaño con la Java Virtual Machine. Esto se solucionó añadiendo al archivo gradle.properties: **org.gradle.jvmargs=-Xmx2048m -XX:MaxPermSize=512m**, lo que aumenta a 2GB.

En esta iteración no se completaron los *tests*, así que como resultado tampoco se realizó una *release*.

5.4.2. Iteración 2

En esta segunda iteración, se establece como objetivo completar los tests y abordar los tickets que surgieron durante el primer sprint. Estos incluyen la posibilidad de volver al menú selector 8a utilizando el botón de retroceso del dispositivo y la adición de un botón *home* que permita acceder



(a) Nuevo menú selector

(b) Nuevo menú selector (Apaisado)

Figura 8: Imágenes del menú selector.

a ese menú desde cualquier parte de la aplicación. Además, se observó que las listas de reproducción estaban ligeramente ocultas, como se puede observar en la Figura 7, por lo que también se propuso ubicarlas junto a los artistas, álbumes y la biblioteca.

El desarrollo comenzó implementando los tests de interfaz correspondientes al primer *sprint*. Para ello, se utilizó Espresso. A continuación, se desarrolló la funcionalidad del botón *home*, se añadió la sección *playlists* al menú y se crearon los correspondientes *tests*. El resultado se puede observar en la Figura 9.

Al acabar los *tests*, se llevó a cabo una nueva *release*, que continuaba con el versionado de la aplicación original, y se le añadió su APK correspondiente. Debido a que realizar esto manualmente resultaba ser una tarea pesada, se creó adicionalmente un nuevo *workflow* que crea la APK y realiza una *release* cuando se sube un *tag* con el formato "vX.X.X", donde "X" son números. Lo normal es que las APKs se encuentren en la carpeta *build*, pero por simplicidad, esta carpeta se mantuvo en el *.gitignore* para evitar conflictos. Por lo tanto, las APKs se pueden encontrar en la sección del GitHub Actions del repositorio.

5.4.3. Iteración 3

La tercera iteración, se enfocó en el algoritmo de aleatoriedad. Como objetivo adicional, se propuso la implementación de la funcionalidad del modo *Save the album*, ver sección 3.1.

Inicialmente, la aplicación contaba con un algoritmo que aleatorizaba las canciones en función del artista. A partir de aquí, se presentaron varios desafíos. El primero consistía en que, para la funcionalidad de *Save the album*, se necesitaba que el algoritmo aleatorizara los álbumes en función del artista, en vez de las canciones. El segundo desafío consistía en que se quería agregar la capacidad de aleatorizar las canciones por álbum, artista o ambos. Esto generaba otra pregunta importante: ¿cómo modificar la interfaz para que el usuario pueda elegir si desea aleatorizar por artista, álbum o ambos?

Lo primero era comprender cómo funcionaba el aleatorizador y su selección. La selección era

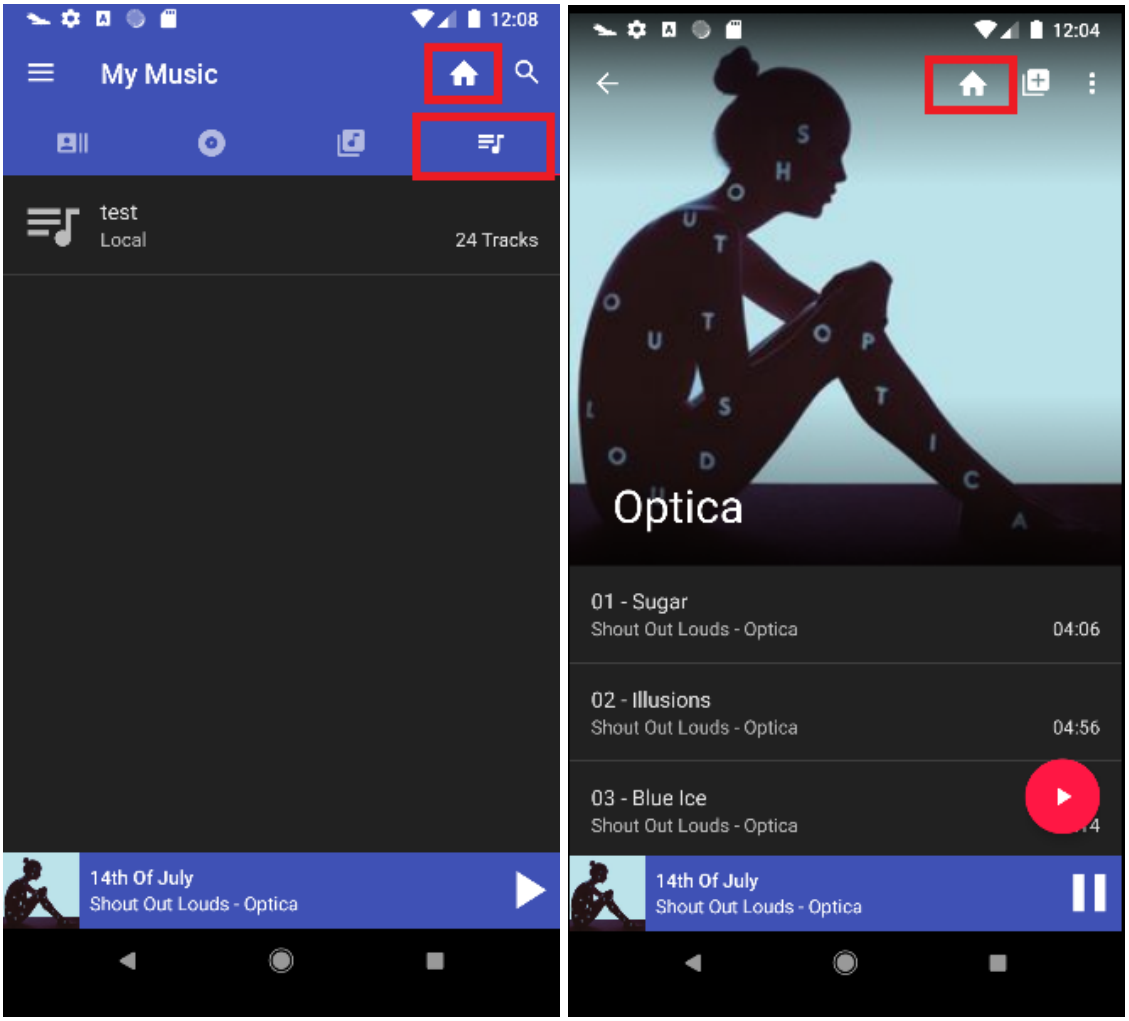


Figura 9: Menú actualizado e integración del homeButton

bastante sencilla: se disponía de un *slider*, y si se movía hacia la izquierda, se utilizaba una aleatorización basada en un generador aleatorio de Java, mientras que si se movía hacia la derecha, se utiliza un generador específico de la aplicación. Todo lo que esté entre los extremos determinará la probabilidad de utilizar el modo tradicional o el *smart random*, que es el nombre dado al generador por el creador. Por ejemplo, si se seleccionaba 75, habría más posibilidades de que se realizase la aleatorización usando *smart random* que usando el método tradicional, pero ocasionalmente se utilizará el método tradicional. Si se configuraba en 25, sucederá lo contrario. Los extremos, 100 o 0, aseguran el uso de uno u otro respectivamente.

El planteamiento original de la interfaz y cómo funcionaba la aleatorización resultó atractivo, por ello se decidió mantenerlo, aunque con algunas modificaciones. Ahora, si se mueve el *slider* hacia la izquierda, se aleatorizará por álbum; hacia la derecha, por artista; entre el 40 y 60 se hará por ambos, y en los extremos, como en el *slider* original, nos aseguraremos de aleatorizar por uno u otro. Esto se puede observar de manera más visual en la Figura 10.

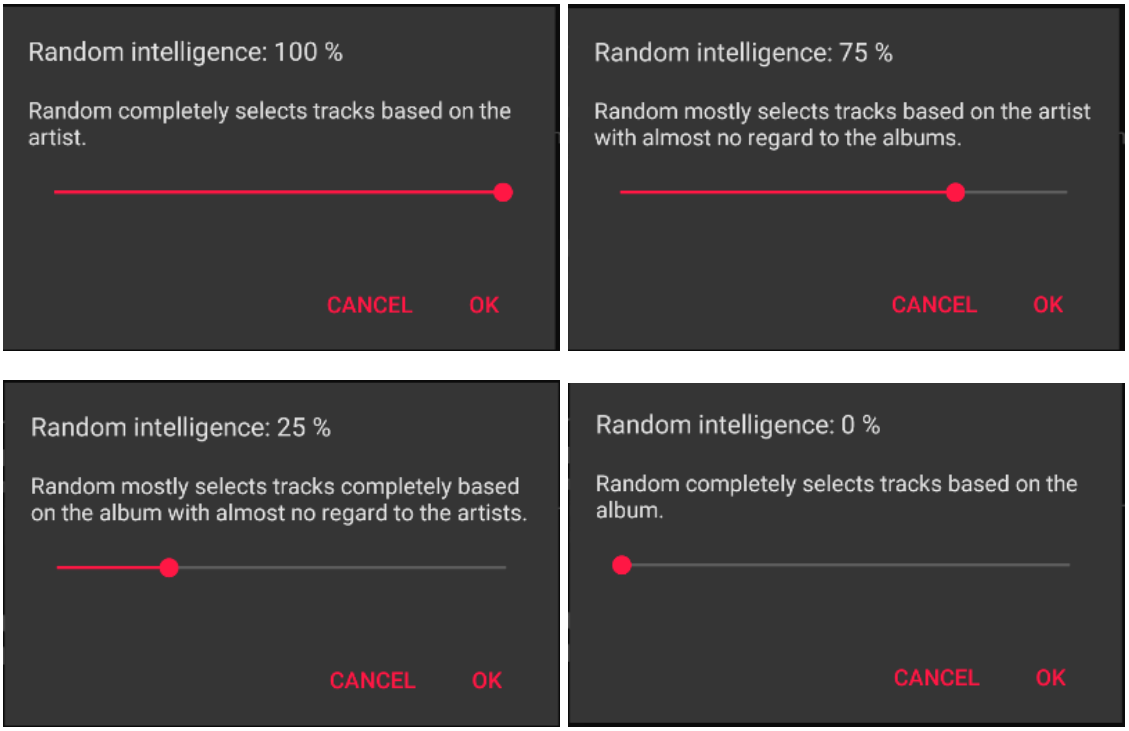


Figura 10: Nuevo menú de selección del modo de aleatoriedad

La modificación del *slider* implicó la traducción de este texto a los diferentes idiomas de la aplicación.

Lo siguiente fue adaptar el aleatorizador para que funcione por artistas, álbumes o artistas y álbumes. Además se descartó el uso del generador de Java baja cualquier circunstancia.En la Sección 7.3 se proporcionan más detallas acerca de este desarrollo concreto.

Como tras la fase de pruebas sobro tiempo, se implementó el modo *Save the album*, que permite reproducir de manera automática un álbum aleatorio de la biblioteca, ver Figura 11. Además, éste se diferencia de los otros álbumes porque al pulsar sobre él se empieza a reproducir de manera automática. Tras esto se realizaron pruebas unitarias del algoritmo y tras su validación se cerró la iteración creando una nueva *release*.

5.4.4. Iteración 4

En la siguiente iteración, se dispuso de menos tiempo de lo normal. Por esta razón, se propuso implementar el *Party Mode*, ver sección de requisitos funcionales 3.1, y algunos cambios más sencillos, como modificar el icono y nombre de la aplicación, añadir un icono a la funcionalidad añadida a *Save the album* y hacer que todos los álbumes se reproduzcan automáticamente al pulsar sobre ellos.

Para implementar el nuevo icono, se utilizó una IA de imágenes autogeneradas llamada *Crayon* [18]. Se le indicó que se buscaba crear un icono para una aplicación en el que una nota musical condujese un coche, y entre todas las imágenes generadas se eligió la imagen de la Figura 12.

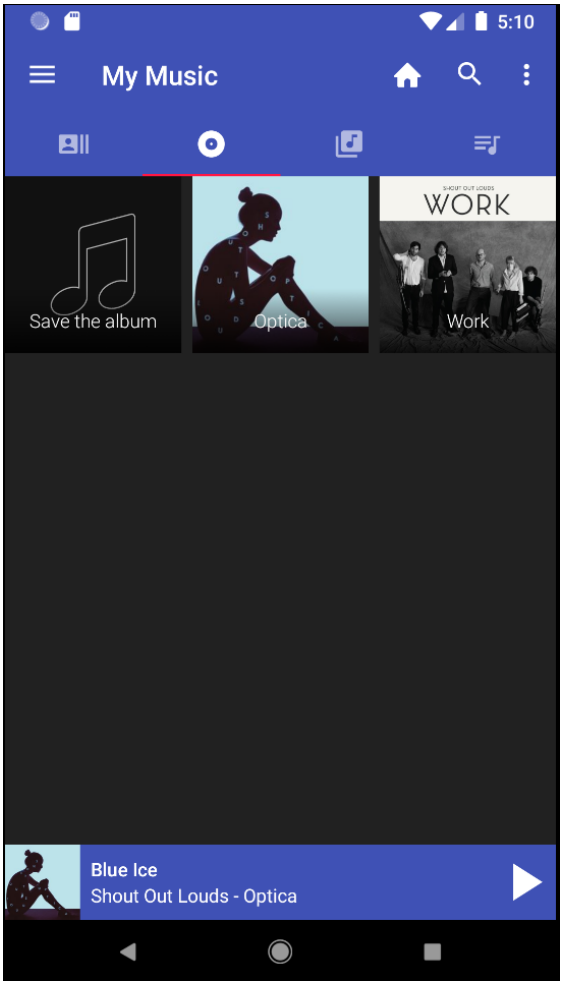


Figura 11: Implementacion de la funcionalidad Save the album

Figura 12: Icono auto generado

Para que todos los álbumes se reprodujesen de forma automática, se extendió la funcionalidad creada para *Save the album*. Y para el icono se eligió el original de la aplicación, ver Figura 13

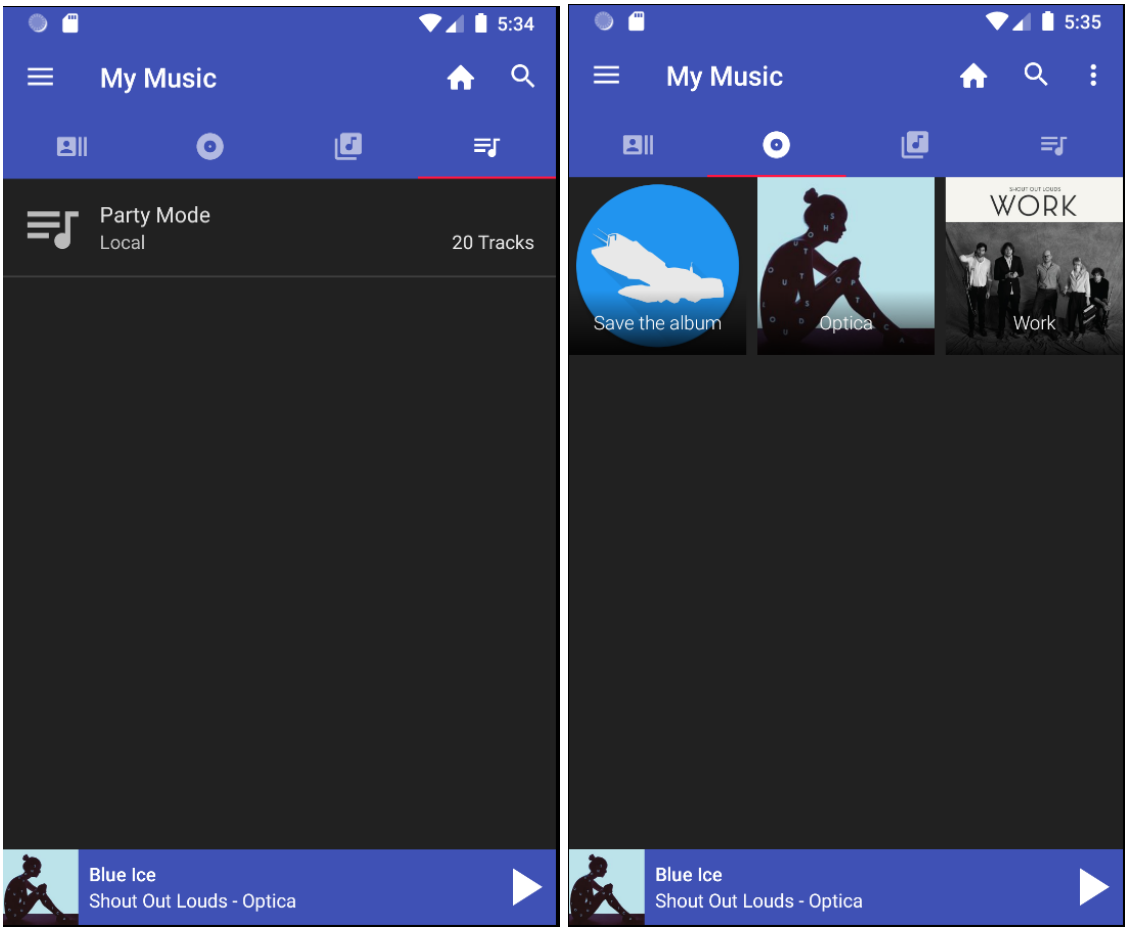


Figura 13: Icono Save the album y Party Mode

derecha.

El modo *Party* no pudo ser implementado completamente, ya que no se incluyó la función que aleatoriza automáticamente todas las canciones al finalizar la reproducción de la *playlist*. Aunque esta funcionalidad se ha añadido como un nuevo ticket, no se considera de alta prioridad, ya que con las 20 canciones disponibles, el usuario ya disfruta de más de una hora de música, y además, al reiniciar la aplicación, la lista se volverá a aleatorizar automáticamente.

Tras esto, se probó su correcto funcionamiento y se generó una nueva *release*. SonarCloud detectó que se generó el mismo *security hotspot* varias veces; tras ser revisados, se observó que se debían a secciones de depuración de código y se asumieron. En una versión final de la aplicación, habría que proponer una solución, puesto que hay múltiples ocurrencias de este *security hotspot* en la aplicación original y quizás en alguna de ellas la información mostrada podría ser crítica.

6. Diseño

En esta sección se describe la arquitectura que sigue la aplicación de la que se partió. En este contexto, es de suma importancia llevar a cabo un análisis detallado de su arquitectura y los diversos patrones de diseño implementados. Este proceso tiene como objetivo preservar y mantener la integridad de la estructura original de la aplicación, evitando así la posibilidad de degradar su arquitectura. Es pertinente destacar que, en el ámbito de las aplicaciones de código abierto, existe el riesgo de que múltiples contribuyentes realicen modificaciones sin una comprensión adecuada, lo que podría resultar en la mezcla inadecuada de patrones y arquitecturas.

6.1. Arquitectura de 3 capas

El patrón arquitectónico que se usa es el modelo-vista-modelo de vista (model-view-view model). En la aplicación claramente se sigue esta estructura:

- **Modelo:** los modelos se usan para representar la capa de datos y toda la lógica de negocio. Contienen los datos y las operaciones relacionadas con esos datos, como consultas a la base de datos, manipulación de datos y validaciones. Los modelos no deberían tener conocimiento directo de la interfaz de usuario. En la Figura 14 podemos ver algunas clases que entran en esta categoría, por ejemplo,la clase "AlbumModel" representa un álbum de un grupo de música .

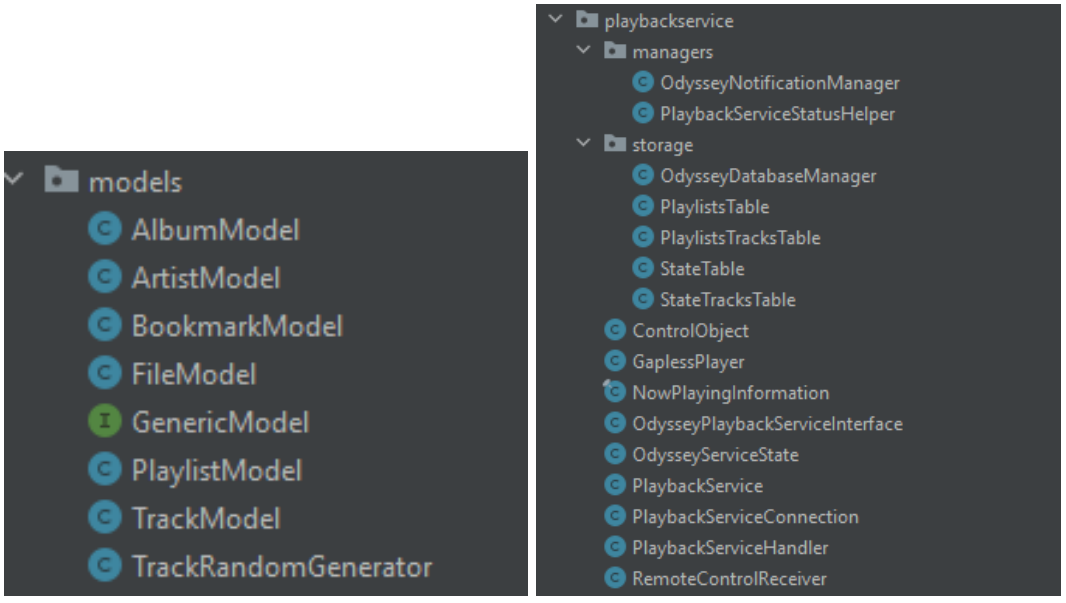


Figura 14: Algunas de las clases dentro del modelo vista.

- **Vista:** usada para representar la información, es decir, es la parte de la aplicación que el usuario ve y con la que interactúa. Estas deben ser pasivas en el sentido de que no contienen lógica de negocio significativa. Su principal tarea es mostrar datos y eventos de usuario, delegando la lógica y manipulación de datos al ViewModel correspondiente. En la Figura 15 podemos ver algunas clases que entran en esta categoría, por ejemplo, los fragmentos que

encontramos en la carpeta *fragments* representan una parte de lo que ve el usuario cuando entra en un menú, por ejemplo "AlbumsFragment" representa lo que se ve en la Figura 11.

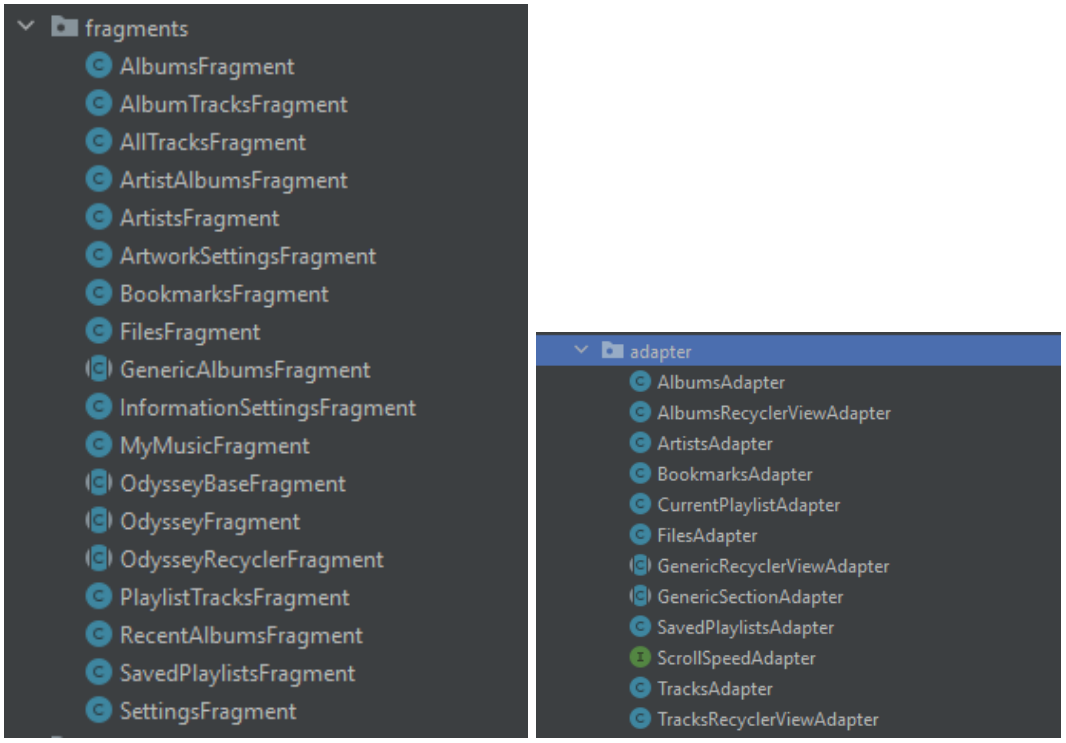


Figura 15: Algunas de las clases que entrarían dentro de la categoría de vista.

- **Modelo de vista:** este actúa como intermediario entre las dos anteriores. Contiene la lógica que conecta los datos del Modelo con la Vista. Su objetivo principal es proporcionar los datos en el formato adecuado para la Vista y manejar la lógica de presentación. La Figura 16 muestra clases que entran en esta categoría.

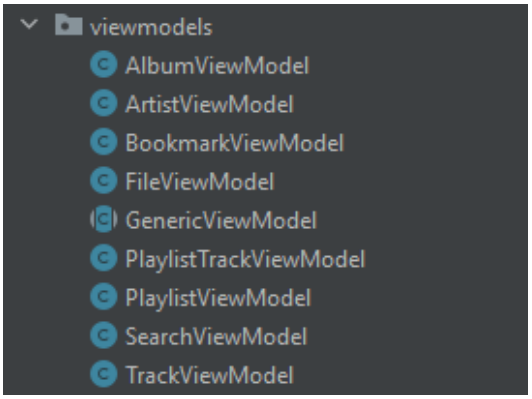


Figura 16: Algunas de las clases dentro del modelo vista.

Ahora bien, la estructura, que como se puede ver encaja con este patrón, no es lo único que determina que este es el utilizado. MVVM utiliza LiveData, este es un mecanismo que nos permite observar los datos y actuar cuando estos cambian. La diferencia con otros observadores es que

```java
ublic abstract class GenericViewModel<T extends GenericModel> extends AndroidViewModel {

    private final MutableLiveData<List<T>> mData;

    abstract void loadData();

    GenericViewModel(@NonNull final Application application) {
        super(application);

        mData = new MutableLiveData<>();
    }

    public LiveData<List<T>> getData() { return mData; }

    public void reloadData() { loadData(); }

    public void clearData() { mData.setValue(null); }

    protected void setData(final List<T> data) { mData.setValue(data); }
```

(a) Uso del LiveData.

```java
public static class PlaylistViewModelFactory implements ViewModelProvider.Factory {

    private final Application mApplication;

    private final boolean mAddHeader;

    private final boolean mOnlyOdysseyPlaylists;

    public PlaylistViewModelFactory(final Application application, final boolean addHeader, final boolean onlyOdysseyPlaylists) {
        mApplication = application;
        mAddHeader = addHeader;
        mOnlyOdysseyPlaylists = onlyOdysseyPlaylists;
    }

    @NonNull
    @Override
    public <T extends ViewModel> T create(@NonNull Class<T> modelClass) {
        return (T) new PlaylistViewModel(mApplication, mAddHeader, mOnlyOdysseyPlaylists);
    }
}
```

(b) Uso de factorias.

Figura 17: LiveData y Factorias.

LiveData tiene en cuenta el ciclo de vida de los componentes de la aplicación garantizando que solo se actualicen los componentes con un ciclo de vida vivo. En la Figura 17a se observa como hace uso de este patrón. Lo usa en una clase genérica de la que luego extiende los demás *view models*. Tiene cierto parecido con el lenguaje de programación "Rust" y permite distinguir entre variables mutables (lectura y escritura) y no mutables (solo lectura). Esto permite que al usar el modelo vista con *LiveData* cuando hay un cambio en una vista (actividad, fragmento...) se genera un evento en el *view model* que lanzará una petición al modelo y cuando la reciba y aplique los cambios la vista se actualizará sola con los nuevos, este proceso esta representado en la Figura 18.

También se puede ver que, dentro de los modelos de vista, sigue una buena práctica de programación recomendada por Android [25] y crea factorías de los *view models*, Figura 17b, dado que muchos de ellos tienen dependencias de otros datos que forman parte de su contenido. De esta manera se pueden recuperar instancias del *view model*. Luego utiliza estos en los fragmentos para acceder y utilizar la información. Además, los *view models* pueden ser compartidos por el mismo fragmento lo que facilita pasar datos de uno a otro.

Durante el proyecto se han llevado a cabo múltiples cambios, los cuales se pueden ver en más detalle en la Sección 7. La mayoría de estos han afectado a la parte del modelo y de la vista, dejando

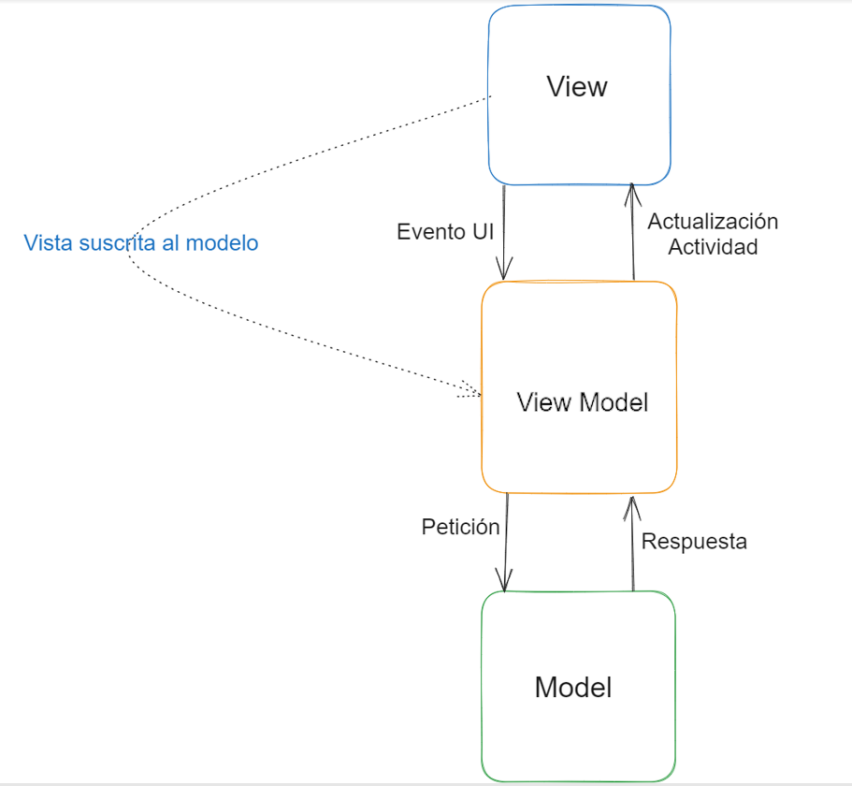


Figura 18: Funcionamiento del MVVM.

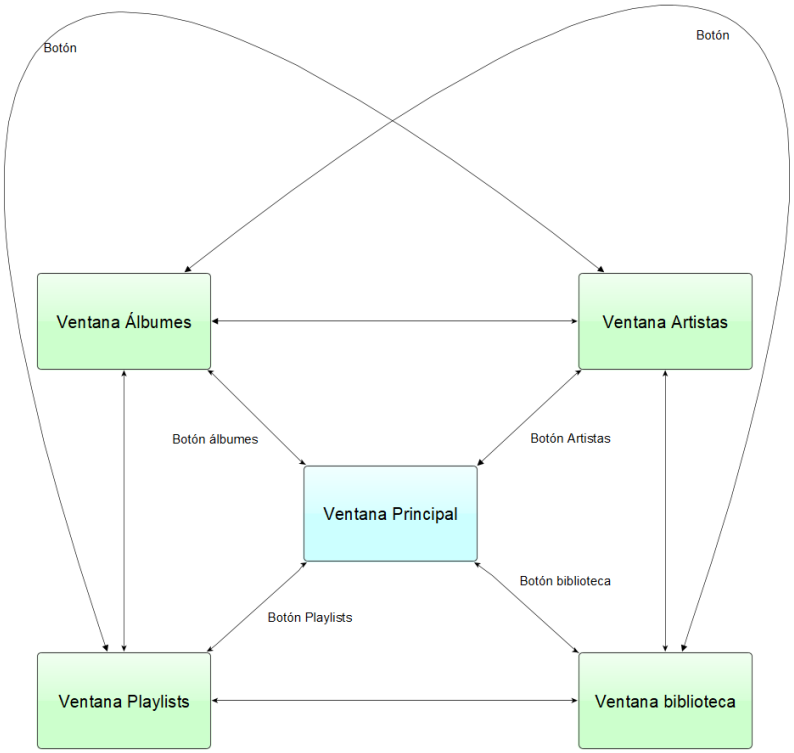
el modelo de vista sin cambios. Las modificaciones en el modelo se han centrado en añadir la lógica de las nuevas funcionalidades, como la parte de aleatorización de álbumes y canciones, mientras que los cambios en las vistas se han centrado en hacer accesibles las nuevas funcionalidades, como *Save the album*, a los usuarios de la aplicación.

6.2. Capa presentación

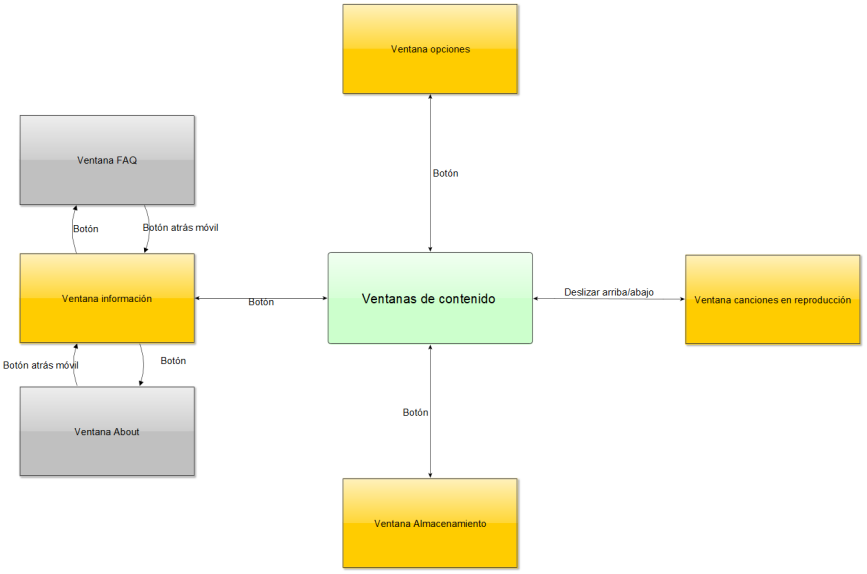
Esta se ha implementado exclusivamente con layouts XML nativos de Android y se ha adaptado para el uso en pantallas grandes apaisadas dado que es lo más común en los coches, como se puede ver en la Figura 8b.

Como se menciona en los requisitos de usabilidad de la Tabla 2, es muy importante poder acceder a la música en dos pulsaciones o menos, por ello se diseño el mapa de navegabilidad representado en la Figura 19a, el cual se acabó acabó plasmando en la vista de la Figura 8a. Inicialmente esta pantalla principal iban a ser dos botones que llevasen al modo *Party* o al modo *Save the album*, desechando gran parte de la funcionalidad inicial. Finalmente, se replanteó la pantalla hacia esta otra, de manera que se permitía a las antiguas y nuevas funcionalidades coexistir. Además, permite navegar entre las distintas ventanas sin tener la necesidad del volver al home, cosa que en el diseño inicial con dos botones no era posible.

Tras esto, siguió el diseño de la navegabilidad entre ventanas internas. En la Figura 19b la ventana de contenido representa a la de artistas, biblioteca, álbumes y *playlists* y desde cualquiera



(a) Navegabilidad inicial



(b) Navegabilidad interna

Figura 19: Navegabilidad Completa

de estas debemos ser capaces de acceder a todo el contenido. Esto ya estaba prácticamente implementado en la versión original, solo faltaba añadir la pestaña de *playlists* a la principal, como se muestra en la Figura 9, donde ya están las cuatro pestañas accesibles con un click. Finalmente, también se modifico la *Toolbar* para que se añadiese un icono de *Home*.

6.3. Capa aplicación

Para la gestión de la música, la aplicación implementa un servicio que coordina la reproducción a través de un sistema de mensajes, con dos clases clave: una como *manager* y otra como estado del reproductor, que también almacena información sobre la canción actual.

Además, escanea y extrae metainformación de archivos móviles, utilizando las APIs de Fanart.tv [26] y MusicBrainz [27] para organizar esta información en bases de datos. Esto asegura una biblioteca de música actualizada y organizada.mTambién ofrece un servicio con Last.FM [28] conocido como *scrobbling*, el cual permite hacer seguimiento de las canciones que se reproducen en tu dispositivo.

Un aspecto destacado es el algoritmo de selección aleatoria de canciones, basado en "Marsaglia XORshift RNGs", explicado en la Sección 7.3, con ocasional uso del generador de números aleatorios de Java. En resumen, este generador de números aleatorios usa una lista de listas que representan los artistas con sus canciones. A partir de esta genera números aleatorios en el rango del tamaño de la lista y sus sublistas para escoger primero el artista y luego la canción. Tras esto elimina la canción seleccionada para no tener repeticiones.

El trabajo del proyecto en esta capa se centra en la aleatorización descrita en el párrafo anterior. Sobre este se han hecho modificaciones y añadido funcionalidad para poder aleatorizar canciones por diferentes factores y también poder aleatorizar la selección álbum, pero se ha mantenido la lógica y estructuras de datos originales. A su vez se ha eliminado la posibilidad de usar el generador de números aleatorios de la biblioteca de Java para obtener un control total sobre la forma de aleatorizar. Pueden encontrarse detalles más concretos respecto al código en la Sección 7.3.

Finalmente mencionar que, en el contexto de este proyecto, no se encontró una gran justificación para no usar un algoritmo aleatorio proporcionado por la biblioteca estándar de Java. Se entiende que el creador buscaba un mayor control sobre la aleatoriedad, pero se supuso que, principalmente, buscaba experimentar y estudiar la generación de números aleatorios.

6.4. Capa datos

La aplicación utiliza SQLite, el cual tiene un gran número de ventajas para esta aplicación como, por ejemplo, su velocidad y ligereza, las cuales se deben a que no es necesario usar un servidor y permite almacenar los datos en local, lo que también hace que no se necesite conexión a Internet para su uso.

La aplicación cuenta con cuatro Tablas para guardar información genérica de las canciones:

- Tabla *Playlists*: se usa para almacenar el nombre de las *playlists* y el número de canciones que tienen. Además utiliza un id único para identificar cada *playlist*.
- Tabla *PlaylistsTracks*: se usa para almacenar cada canción con su álbum, artista y *playlist* a la que pertenece. Para almacenar el artista, la canción y el álbum también utiliza ids. Además

también guarda la posición que ocupa la canción el la *playlist*.

- Tabla *State*: se usa para guardar cual ha sido la última canción en sonar, en que estado esta el randomizador, si el modo de repetición esta activo o no y el número de pistas que hay en la cola actual.
- Tabla *StateTracks*: guarda el numero de pista, su título, álbum, artista, duración y su url.

Y también cuenta con dos Tablas más para guardar las imágenes del álbum y del artista:

- Tabla *AlbumArt*: se usa para almacenar el nombre del álbum y el artista, junto con el id del álbum, el *path* de la imagen y dos campos que indican si se encontró la imagen y si la imagen tiene el *path* completo o no.
- Tabla *ArtistArt*: se usa para almacenar el nombre del artista, junto con su id, el *path* de la imagen y dos campos que indican si se encontró la imagen y si la imagen tiene el *path* completo o no.

A partir de estas, es capaz de almacenar todos los datos de las canciones con sus álbumes, artistas, y respectivas caratulas, y guardar el último estado de la *playlist* que este sonando antes de cerrarse para poder recuperarlo. Esto último es muy útil en caso de que la aplicación, por cualquier motivo, se caiga o se cierre. También resulta especialmente útil si, por ejemplo, se esta escuchando una lista de *podcasts* y se quiere seguir por donde se dejó. Las tablas no guardan relación entre sí por lo cual no hay claves foráneas o *Foreign Keys*, a pesar de estar usando un lenguaje relacional.

Cabe destacar que la aplicación no utiliza un *framework* de mapeo objeto-relacional (ORM) como Room [30], seguramente porque el desarrollo de esta aplicación comenzó en 2016, un año antes de que Google anunciase y lanzase Room, y para cuando estos se popularizaron ya no le interesaría implementarlo. En la aplicación las consultas se hacen en la clase OdysseyDatabaseManager, las cuales son llamadas por las clases que forman el modelo de vista para extraer los datos, siendo estas las clases de acceso a datos.

Para leer los ficheros se utiliza el servicio (MediaScannerService.java), mencionado en la capa de aplicación de la Sección 6.3. Este servicio *Media Store* [29] del móvil y utiliza una clase (MediaStoreProjections.java) que guarda en diferentes listas la información que se encuentra en la *Media Store*. De los archivos se extraen todos los meta datos con ayuda de la clase MetaDataLoader.java y luego con esta se crean los modelos de datos que posteriormente se guardan en la base de datos.

Se utiliza una clase auxiliar llamada MusicLibraryHelper.java que accede a los datos de la *Media Store* para poder conseguir información como las canciones de un álbum o artista o el id de un artista, entre otros. Finalmente, las operaciones a la base de datos se hacen a través de la clase OdysseyDatabaseManager.java, esta clase permite realizar operaciones CRUD sobre la base de datos siendo la que actúa de intermediaria entre la lógica de aplicación, acciones del usuario y base de datos. Los cambios realizados en esta capa, se centran en estas clases, a las cuales se les han hecho pequeñas modificaciones para guardar el álbum *Save the album* y la *playlist Party Mode* con unos datos concretos, como el id, dentro de la aplicación.

7. Implementación

En esta sección se revisará el desarrollo de las diferentes funcionalidades y las complicaciones encontradas durante el desarrollo. Se verán las modificaciones más importantes, dejando de lado las pequeñas adaptaciones del código original que también se hicieron.

7.1. Menú inicial

Para este desarrollo se plantearon varias opciones. La primera de ellas era rehacer la actividad principal para contener estos nuevos botones y tratar de mantenerlo todo en una actividad. La segunda era cargar un nuevo fragmento encima desde la actividad principal. Y la tercera opción, crear una nueva actividad que luego llamase a la principal y le indicase a que menú desplazarse. Las dos primeras opciones se descartaron porque esa actividad ya tenía demasiadas tareas y había crecido de manera desproporcionada, lo que hacía que añadirle más funcionalidad fuese contraproducente. Por ello, finalmente se optó por la tercera opción. Al crear una nueva actividad que contenía la lógica de selección, se aumentó la modularidad y se evitó un crecimiento de la complejidad de la clase principal. La única contra parte de esta opción es que las actividades son relativamente pesadas comparadas con un fragmento, pero en este caso es una actividad muy sencilla que se llama por lo general una vez en el ciclo de vida de la aplicación así que apenas afecta al rendimiento.

El código se basa en que, en función de que botón pulsemos, le indicamos a la actividad principal que pantalla,artistas, álbumes, biblioteca o *playlists*, debe cargar tras iniciarse. Para ello, usamos un enumerado ya existente en la aplicación que se usa para distinguir justamente estas pantallas. Para esta primera versión, había que diferenciar, en la actividad principal, cuando se llamaba a las *playlists* de las otras, puesto que estaba en un fragmento distinto, y también había que añadirla al enumerado, puesto que esta no se encontraba en él.

7.2. Ventana de playlists y botón Home

Para el desarrollo del botón se buscó una imagen que encajase con el diseño de la aplicación en Freepik [31] y posteriormente se añadió a la *toolbar*. Como se quiere poder acceder a este botón desde cualquier menú de navegación no hizo falta programar comportamientos especiales para cada vista.

Para añadir la ventana de las *playlists*, lo primero fue modificar la clase MyMusicFragment.java (que es el fragmento que guarda artistas, álbumes y la biblioteca) dado que aquí se encontraba el adaptador que permitía la navegabilidad entre los fragmentos. Inicialmente estaba limitado a tres (artistas, álbumes y biblioteca), así que se aumentó a cuatro y se añadió que si se pulsaba el cuarto se abriese el fragmento de las *playlists*. Además se le añadió el icono que se usa previamente para las *playlists* en el menú lateral. Adicionalmente al estas poder ser modificadas de manera dinámica mientras se esta fuera del fragmento, para evitar que nuevas *playlists* no apareciesen se añadió que al pulsar sobre este se refrescase la información.

Finalmente se modificó la actividad principal puesto que ya no era necesario diferenciar *playlists* de las demás puesto que ya si estaban en el mismo fragmento, lo cuál hizo más legible el código.

Este desarrollo, aparentemente sencillo, llevó más tiempo del esperado por la falta de conocimiento sobre la parte del diseño (XMLs) de la aplicación. El problema estuvo, concretamente, en la

toolbar, que es el elemento que contiene el icono de home en la Figura 9, y en el *viewPager*, que en la misma Figura 9, es el elemento que contiene los dibujos de la sección artistas, álbumes, biblioteca y *playlists*. Inicialmente no se entendió bien el comportamiento de la *toolbar* y, de manera errónea, se pensó que el *viewPager*, que era el que permitía la navegación entre los distintos fragmentos, debía encontrarse completamente definido en la parte del diseño. Para entender el correcto funcionamiento de ambos se utilizó la documentación de Android ([32] y [33]) y se fueron buscando similitudes con el código. Tras esto aumentó la familiarización con esta parte considerablemente.

7.3. Algoritmo de aleatorización

En la tercera iteración, comentada en la Sección 5.4.3, se estableció que se iba a necesitar aleatorizar por álbum, por artista y por las dos a la vez para las canciones y *Party Mode*, y por artista y álbum para el modo *Save the album*.

Para adaptar la forma de aleatorizar, se siguió el modelo que utilizaba originalmente la aplicación. Este funciona de la siguiente manera: a partir de una lista de canciones, la cual se pasa como parámetro de entrada, se genera una lista de listas la cual separa a los artistas y sus respectivas canciones, de tal forma que para cada artista se mantiene una lista interna con sus canciones. La peculiaridad es que no se guardan las canciones en sí, si no que se guarda la posición de la canción en la lista que se pasa como parámetro de entrada, por ello es necesario guardar una copia de esta en una variable.

Con el objetivo de implementar el modo *Save the album*, se reutilizó esta estructura completamente, pero con álbumes en vez de con canciones. Para ello se creó el método *fillAlbumFromList(...)*, mostrado en el Listing 1, el cual a partir de una lista de álbumes, primero guarda una copia y después genera la lista que separa los artistas con sus álbumes. Adicionalmente, se optimizó ligeramente usando el método *computeifAbscent()* para reducir principalmente la complejidad cognitiva, dado que había muchos if consecutivos y anidados que dificultaban la legibilidad. Con esta función si la clave no está presente en el mapa o el valor nulo está relacionado con la clave, entonces intenta calcular el valor utilizando la función de mapeo proporcionada. También ingresa el valor calculado en el mapa a menos que el valor calculado sea nulo.

Listing 1: Código del método fillAlbumFromList

```
public synchronized void fillAlbumFromList(List<AlbumModel> albums) {
        // Clear all entries
        mDataAlbum.clear();

        mOriginalListAlbum = albums;

        LinkedHashMap<String, List<Integer>> artistAlbumsMap = new
            LinkedHashMap<>();

        if (albums == null || albums.isEmpty()) {
                // Abort for empty data structures
                return;
        }

        // Iterate over the list and add all albums to their artist lists
        int albumNo = 0;
```

```
            for (AlbumModel album : albums) {
                String artistName = album.getArtistName();
                //If the artists doesn't exist in the map we create it,
                    else we add the new value
                List<Integer> list =
                    artistAlbumsMap.computeIfAbsent(artistName, k-> new
                    ArrayList<>());

                // Add pair of position in original list and track itself
                    to artists bucket list
                list.add(albumNo);

                // Increase the album number (index) of the original list
                albumNo++;
            }
            //Add values
            mDataAlbum.addAll(artistAlbumsMap.values());
            Collections.shuffle(mDataAlbum);
        }
```

Tras esto, se reutilizó el generador de números aleatorios que ya existía, pero con la nueva lista de artistas-álbumes. Este sigue una lógica sencilla: coge la lista con sus artistas y sus canciones y genera un número limitado por el tamaño de esta para escoger artista. Luego repite el mismo proceso para escoger la canción y la elimina de la lista. Si la lista se acaba, la rellena a partir de la principal que se mantiene intacta en la variable local.

Finalizada la parte de aleatorizar la selección de un álbum por artista, quedaba solventar que se pudiese aleatorizar por artista y álbum. Para ello, se añadió otra lista de enteros más, para que al elegir un artista se puedan almacenar sus canciones por sus álbumes. Esta no hubiese sido necesaria si en vez de usar listas de enteros que no permiten saber a que artista/álbum pertenece ese entero se hubiese modificado la funcionalidad para que se usasen mapas, estos hubiesen hecho el código más simple y sobretodo legible, pero se decidió mantener la estructura propuesta inicialmente. EL funcionamiento es muy parecido al primer método enseñado, primero se escoge un artista, luego a partir de todas sus canciones se genera la lista de álbumes con sus listas internas canciones, se escoge una nueva canción y se elimina de la lista artistas-canciones.

Listing 2: Código del método randomizeByArtistAndAlbum

```
        private int randomizeByArtistAndAlbum(){
            //mDataArtists contains artists-tracks lists
            int songNumber;
            //First randomize by artist then by album

            if (mDataArtists.isEmpty()) {
                // Refill list from original list
                fillFromList(mOriginalList);
            }

            // First level random, get artist
            int randomArtistNumber =
                mRandomGenerator.getLimitedRandomNumber(mDataArtists.size());

            // Get artists bucket list to artist number
```

```
            List<Integer> artistsTracks;

            // Get the list of tracks belonging to the selected artist
            artistsTracks = mDataArtists.get(randomArtistNumber);
            List<TrackModel> tracks = new ArrayList<>();
            for(Integer i:artistsTracks) {
                    tracks.add( mOriginalList.get(i));
            }
            //Divide all the albums from the artist
            //fills mDataSelectedAlbumTracks with albums -tracks from the
                choosen artists
            fillArtistsAlbumfromList(tracks);

            //Select a random album
            int randomAlbumNumber =
                mRandomGenerator.getLimitedRandomNumber(mDataSelectedAlbumTracks.size());
            // Get a random album bucket list
            List<Integer> albumTracks =
                mDataSelectedAlbumTracks.get(randomAlbumNumber);

            // Check if an album was found
            if (albumTracks == null) {
                    return 0;
            }
            //get random track
            int randomTrackNo =
                mRandomGenerator.getLimitedRandomNumber(albumTracks.size());

            songNumber = albumTracks.get(randomTrackNo);

            // Remove track to prevent double plays
            artistsTracks.remove(albumTracks.get(randomTrackNo));

            // Check if tracks from this artist are left, otherwise remove the
                artist
            if (artistsTracks.isEmpty()) {
                    // No tracks left from artist, remove from map
                    mDataArtists.remove(randomArtistNumber);

            }
            return songNumber;
    }
```

Finalmente, para aleatorizar solo por álbum se añadió otra lista que guardase albumes-canciones a partir de una lista de canciones y se repetía el proceso del caso anterior.

Test del algoritmo aleatorio

La aplicación utiliza una implementación del algoritmo de aleatorizado de "Marsaglia', conocido como "XORShift RNGs" [34]. Este se basa en una serie de operaciones XOR bitwise (operaciones a nivel de bits) que mezclan los bits con el objetivo de tener un valor más aleatorio. Para validar si dicha implementación era correcta y justa con todas las posibilidades, el código original contaba con un *test*. Este recibe dos parámetros, el primero determina el rango de números con el que se

va a trabajar, siendo este de 0 al rango seleccionado , y el segundo el número de veces que se va a escoger un número aleatorio dentro de ese rango. Contando el número de veces que sale cada número dentro del rango indicado calcula la varianza, desviación estándar y relativa.

Para determinar su "justicia", se han realizado 500, 1000 y 5000 selecciones de números entre 0 y 49, con el objetivo de ver como se distribuían el número de selecciones y los datos devueltos por la varianza y desviaciones. Esta selección de números es la que se utiliza para hacer selecciones aleatorias de artista, álbum y canción en los diferentes contextos de la aplicación. Además, estos grupos se han repetido 10 veces y se ha calculado la media de todos los resultados.

La Figura 20 muestra el resultado de los cálculos anteriores, permitiendo determinar que los números seleccionados parecen estar distribuidos de manera uniforme. En el grupo de 500 y 1000 se encuentra un número mayor de picos que en el grupo de 5000, además sus valores, en porcentaje, distan más de lo que sería el número de selecciones ideal (10 y 20 respectivamente). A medida que aumenta el número de selecciones, la frecuencia de selección de cada número se acerca más a un valor uniforme debido a la ley de los grandes números.

En general, los resultados sugieren que el algoritmo utilizado para seleccionar números aleatorios parece estar generando resultados razonablemente uniformes.

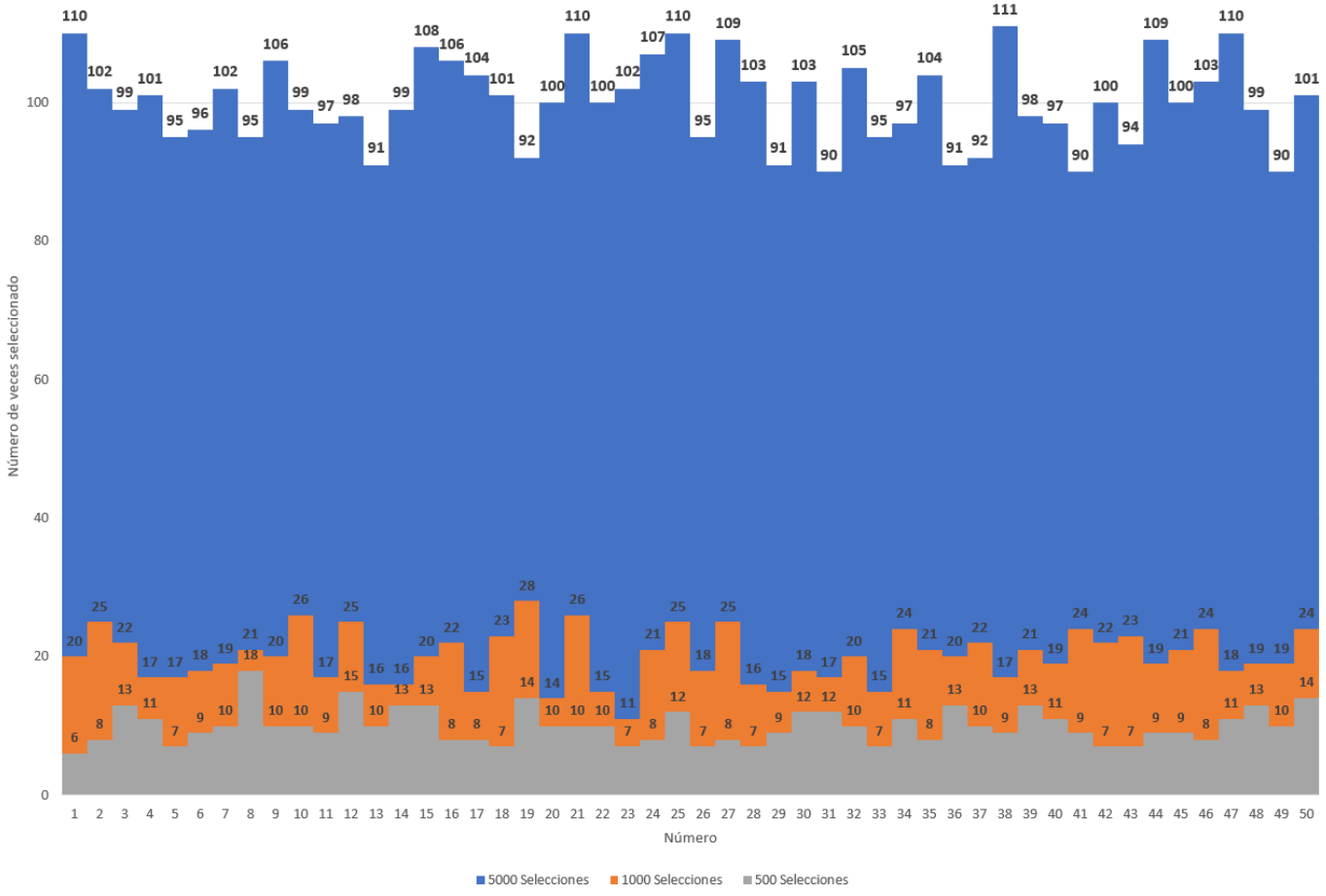


Figura 20: Histograma de los resultados al correr el algoritmo 500, 1000 y 5000 veces.

7.4. Party Mode y Save the Album

Una vez adaptado el generador de números aleatorios, la implementación de estas funcionalidades se reducía a programar el comportamiento específico al pulsar sobre ellas y hacer que se creasen al iniciar la aplicación.

Para incluir ambas funciones en la aplicación, lo primero que se hizo fue añadir una fila especial para el *Party Mode* en la base de datos. Para *Save the album* esto no era tan banal puesto que estos no se guardan en la aplicación. La aplicación cuenta con un servicio de lectura de la *Media Store*, que es de donde se saca la información sobre los álbumes cuando se necesita. Dentro de este servicio hay un método que devuelve una lista con todos los álbumes, este se modificó para que añadiese a lista el álbum *Save the album*.

En el caso de *Save the album*, para cargar el icono no valió con cogerlo de la carpeta de *drawables* habitual en la estructura de una aplicación Android, sino que fue necesario necesario pasarlo primero a la sdcard del móvil y desde ahí cargarlo porque si no, no lo detecta. Esto conlleva múltiples riesgos de seguridad que han sido valorados y asumidos dado que solo se va a acceder para escribir una imagen controlada y ya se solicita previamente el acceso a esta.

Para añadir un comportamiento distinto en ambos casos, se comprueba, comparando el nombre, si se ha pulsado sobre la funcionalidad *Save the album*/*Party mode* o sobre un álbum/*playlist* normal. En el caso del *Save the album*, se escoge uno aleatorio. Para esto, hay que tener en cuenta que no sea el propio, por ello hay que eliminarlo de la lista antes de generar el número aleatorio. Cuando ya se tiene el nuevo álbum, se continúa con la funcionalidad normal del método, a la cual se añadió que comenzase a reproducirse automáticamente.

En el caso de *Party mode*, no se pudo acabar la implementación completa, como ya se comentó en la Sección 5.4.4. Esto se debió al complejo sistema de mensajes que se utiliza en el servicio de reproducción. La ambigüedad de estos, sumada a la falta de documentación y a su complejidad dificultan su escalabilidad y adaptabilidad. La implementación parcial conseguida hace que, al seleccionar una canción esta pasase al primer puesto y la lista se rellenase por el final con nuevas canciones. Para lograrlo, primero se localizó en que fragmento se seleccionaban las canciones de las *playlists*. Tras encontrarlo se creó la lógica de negocio, la cual primero comprueba si se esta seleccionando una canción de la *Playlists Party Mode*, de ser así extrae todas las canciones de la biblioteca y todas las canciones que forman el *Party Mode* actual, generando así dos listas de canciones. Tras esto se eliminan de la primera las que se encuentran en ambas listas con el objetivo de evitar repeticiones. A la vez que se van eliminando las canciones anteriores a la seleccionada, a partir de esta nueva lista, se van seleccionando canciones de manera aleatoria por artista y álbum para ir rellenando por el final de la cola. Tras rellenar de nuevo la lista se comprueba que el modo *Repeat* sigue activo, en caso contrario se vuelve a activar.

Para activar el modo *Repeat* se utiliza el servicio de reproducción, el cual guarda información de la lista y canciones que se están reproduciendo actualmente y del modo de repetición, que puede ser ninguno, canción o todos. Modificando este último, y actualizando la vista, se consigue que la *playlist* entre en bucle y el usuario vea como el botón de "repetición" esta activado.

8. Validación y Pruebas

En este capítulo se revisará el proceso seguido para comprobar el correcto funcionamiento de la aplicación con las nuevas funcionalidades desarrolladas. Dada la extensión de la aplicación y la falta de pruebas previamente desarrolladas sólo se han probado los nuevos desarrollos.

Las pruebas están organizadas en diferentes módulos en función del tipo del que sean, y dentro de estos módulos se separan en clases que distinguen la funcionalidad que se quiera probar. Estas se pueden encontrar en el repositorio de GitHub [22].

8.1. Pruebas de interfaz

En esta sección, se detallan las pruebas de interfaz realizadas en el desarrollo de la aplicación. Estas pruebas se han llevado a cabo utilizando el marco de pruebas Espresso [12], que es ampliamente reconocido por su capacidad para automatizar pruebas de interfaz de usuario en aplicaciones Android. El objetivo de estas pruebas es garantizar la correcta funcionalidad y usabilidad de los componentes de la interfaz de usuario, incluido un menú selector y los fragmentos asociados.

8.1.1. Escenarios de prueba

Se han definido varios escenarios de prueba con el objetivo de abordar distintos aspectos de la interfaz de usuario. Cada escenario se diseñó para evaluar tanto la navegación entre fragmentos como el funcionamiento de los elementos de navegación dentro de cada fragmento. Los siguientes escenarios de prueba fueron implementados:

- Selección de Categorías: Se simula la selección de cada una de las cuatro categorías disponibles en el menú selector. Cada categoría conduce a un fragmento específico: "Álbumes", "Artista", "Biblioteca" y *"Playlists"*.
- Navegación de Fragmentos: Se prueba la navegación dentro de cada fragmento mediante la simulación de acciones de usuario, como hacer clic en elementos y botones de *home* y *back*. Se verifica que al interactuar con el botón *home*, el usuario sea redirigido nuevamente al menú selector, y que al usar el botón *back* del dispositivo se logre el mismo resultado.

8.1.2. Resultados y conclusiones

Todas las pruebas de interfaz se ejecutaron con éxito, demostrando la funcionalidad coherente y predecible de la interfaz de usuario. Se verificó que los elementos de navegación cumplen su propósito y que el usuario puede moverse fluidamente entre los distintos fragmentos y el menú selector. Estas pruebas han sido esenciales para garantizar una experiencia de usuario de alta calidad en la aplicación.

8.2. Pruebas unitarias

En esta sección, se presentan las pruebas unitarias diseñadas y ejecutadas para validar los métodos relacionados con la adaptación del algoritmo aleatorio en la aplicación. Estas pruebas se realizaron utilizando el marco de pruebas unitarias de Android. Están diseñadas para garantizar que los métodos cumplan con sus funcionalidades esperadas en diversos escenarios.

8.2.1. Escenarios de prueba

Los métodos sujetos a las pruebas unitarias son los siguientes:

Método fillFromList(List<TrackModel>tracks)

Este método se encarga de crear listas de artistas y sus canciones, así como listas de álbumes y sus canciones, con su posición en la lista de reproducción original. A continuación, se describen los aspectos a comprobar de este método:

- Limpia las listas existentes.
- Crea las listas de artistas y álbumes con sus respectivas canciones y sus posiciones con los datos de entrada.
- Ante una entrada vacía, limpia las listas.

Método fillAlbumFromList(List<AlbumModel>albums)

Este método crea una lista de artistas y sus álbumes con su posición en la lista de reproducción original, utilizada para la funcionalidad de *Save the album*. A continuación, se describen los aspectos a comprobar de este método:

- Limpia las listas existentes.
- Crea la lista de artistas con sus álbumes y sus posiciones con los datos de entrada.
- Ante una entrada vacía, limpia las listas.

Método getRandomTrackNumber()

Este método genera un número de pista aleatorio dentro de una de las listas de pistas original utilizada en la llamada a fillFromList. La lista sobre la que actuará dependerá del factor de inteligencia que hayamos puesto, pero siempre que se trabaje sobre la misma lista nunca deberá salir la misma canción. A continuación, se describen los aspectos a comprobar de este método:

- Con un factor de inteligencia de 0 aleatorizará por álbum.
- Con un factor de inteligencia de 100 aleatorizará por artista.
- Con un factor de inteligencia entre 1 y 39 aleatorizará por álbum o artista con una mayor probabilidad de usar el álbum. Este caso podrá causar repeticiones de la canción elegida puesto que alterna sobre las dos listas.
- Con un factor de inteligencia entre 61 y 99 aleatorizará por álbum o artista con una mayor probabilidad de usar el artista. Este caso podrá causar repeticiones de la canción elegida puesto que alterna sobre las dos listas.
- Con un factor de inteligencia entre 40 y 60 aleatorizará por álbum y artista, pero no podrá causar repeticiones.
- Se comprueba el correcto funcionamiento en caso de que las listas estén vacías.

Método getRandomAlbumNumber()

Este método genera un número de pista aleatorio dentro de la lista original utilizada en la llamada a fillAlbumFromList. La aleatoriedad de este número no depende del factor de inteligencia. A continuación, se describen los aspectos a comprobar de este método:

- Se comprueba que nunca se repite álbum escogido.
- Se comprueba el correcto funcionamiento en caso de que la lista estén vacía.

8.2.2. Resultados y conclusiones

Todas las pruebas unitarias se ejecutaron con éxito, lo que indica que los métodos relacionados con la adaptación del algoritmo aleatorio funcionan según lo previsto. Estas pruebas han sido fundamentales para garantizar la confiabilidad y el correcto funcionamiento del algoritmo en la aplicación.

8.3. Pruebas integración

Las pruebas de integración son un tipo de pruebas que se centran en evaluar cómo interactúan y funcionan en conjunto los diferentes componentes, módulos o unidades de un sistema. Su objetivo principal es detectar y resolver posibles comportamientos incorrectos entre las partes que componen una aplicación o sistema más grande.

En este caso no se contaba con ninguna prueba que asegurase el correcto funcionamiento entre las diferentes actividades, fragmentos y sus módulos. Como crear nuevas pruebas de integración que asegurasen el correcto funcionamiento de la aplicación hubiese supuesto un trabajo excesivo se probó, a través del emulador, que la aplicación funcionase correctamente y no se dieran errores inesperados ni a nivel de uso ni a nivel interno, estos últimos se reflejaban a través de la consola *Logcat*.

En este contexto, en base a los recursos disponibles, al coste de oportunidad, y considerando que se desarrollaron pruebas unitarias que probaban el funcionamiento individual, se consideró que no tendría sentido realizar pruebas de integración de los nuevos métodos puesto que se carecen de las demás.

Esto es un caso excepcional y habría que realizar las pruebas, no solo para asegurar la correcta interrelación de los componentes, sino también para determinar si, en caso de efectuar modificaciones en el código, se ha afectado su correcto rendimiento.

8.4. Pruebas de aceptación

En esta sección, se detallan las pruebas de aceptación que se llevaron a cabo para validar los requisitos no funcionales de estabilidad y usabilidad, definidos en la Tabla 2, de la aplicación. Estas pruebas son esenciales para garantizar que la aplicación cumple con los requisitos funcionales y expectativas del usuario final. Las pruebas de aceptación se realizaron en dos etapas: utilizando un simulador durante los sprints de desarrollo y posteriormente en un dispositivo móvil real.

8.4.1. Pruebas con Simulador

Durante cada *sprint* de desarrollo, se realizaron pruebas de aceptación utilizando un simulador de dispositivos móviles. Estas pruebas permitieron verificar que las nuevas funcionalidades implementadas se ajustaran correctamente al flujo de la aplicación y que no se introdujeran problemas graves de usabilidad. Los escenarios de prueba incluyeron la navegación por la interfaz, la funcionalidad de las nuevas funcionalidades y botones y la interacción con los distintos fragmentos.

8.4.2. Pruebas en Dispositivos Reales

Después de completar el desarrollo y las pruebas con el simulador, se procedió a realizar pruebas de aceptación en un dispositivo móvil real. Esta etapa es fundamental para validar la aplicación en un entorno más cercano al uso real. Las pruebas en un dispositivo móvil permitieron detectar posibles problemas relacionados con el rendimiento, funcionalidad y la adaptación de la interfaz a diferentes tamaños de pantalla.

Las Figuras 22 y 21 muestran como, en una última instancia, la aplicación también fue probada en un automóvil.

8.4.3. Resultados y conclusiones

Las pruebas de aceptación fueron fundamentales para asegurar la calidad y la usabilidad de la aplicación antes de su lanzamiento. La realización de pruebas en diferentes etapas, tanto con simuladores como en dispositivos móviles reales, permitió identificar y corregir problemas en una variedad de entornos. Estas pruebas contribuyeron significativamente a la mejora de la aplicación y a la entrega de una experiencia de usuario sólida y satisfactoria.

9. Conclusiones y trabajos futuros

Para finalizar, en esta última sección se recogen las conclusiones obtenidas durante el proceso de la realización del proyecto, así como un conjunto de posibles mejores que se pueden incorporan en un futuro para obtener una aplicación más completa.

9.1. Conclusiones

Este proyecto ha sido un emocionante viaje de aprendizaje y logros personales. La tarea de transformar un reproductor de música preexistente en una herramienta adaptada a las necesidades del entorno automovilístico ha demostrado ser un desafío gratificante. A través de esta experiencia, he reafirmado mi capacidad para abordar problemas complejos y aplicar soluciones creativas. Algunos de los puntos más esenciales han sido:

- Trabajar con una arquitectura Modelo-Vista-Modelo de Vista (MVVM), la cual inicialmente desconocía, ha supuesto un reto, pero también ha ampliado mi caja de herramientas de diseño y desarrollo.
- La lectura de código representa un método sumamente efectivo para adquirir conocimientos. Tomar un repositorio del cual eres ajeno y esforzarse por comprender su funcionamiento constituye un ejercicio altamente beneficioso. Este ejercicio ha proporcionado una valiosa oportunidad tanto para analizar implementaciones exitosas como para identificar posibles fallos o errores que se han llevado a cabo.
- La reutilización de código ha sido esencial para optimizar el proceso de desarrollo y garantizar la eficiencia en el uso de los recursos. Al integrar componentes de software existentes en el nuevo contexto, he aprendido la importancia de mantener un equilibrio entre la innovación y la aplicación de soluciones ya probadas.
- La colaboración con mis tutores ha sido un pilar fundamental en este proyecto. Compartir ideas, obtener retroalimentación y discutir enfoques me ha proporcionado nuevas perspectivas y ha fortalecido mi capacidad para trabajar en equipo.
- El uso de GitHub, un entorno profesional de desarrollo colaborativo, respaldado por una metodología profesional, como es el caso de la metodología ágil, acompañado de una integración continua y parcialmente automatizada, ha destacado la importancia de estas herramientas y enfoques en proyectos de desarrollo de Software. Aprender a utilizarlas a un nivel más alto exprimiendo sus capacidades de organización y automatización, ha establecido un punto de inflexión significativo en la calidad de mi software.

En última instancia, el proyecto de adaptar un reproductor de música para su uso en el coche ha sido un hito personal. No solo he ampliado mis habilidades técnicas y conocimientos en desarrollo de software, sino que también he reafirmado mi pasión por abordar desafíos complejos y encontrar soluciones innovadoras.

9.2. Trabajos futuros

La aplicación actual está bastante completa, pero hay varias propuestas que se han planteado:

- Mejorar la usabilidad de las *playlists*. No hay una opción para añadir una canción a una *playlist*, lo que hace de esto una tarea tediosa puesto que hay que ponerla a reproducir, añadir la canción a la cola y sobrescribir la *playlist*. En principio esta tarea se consideró de baja importancia en este proyecto, ya que la única *playlist* sobre la que se ofrece funcionalidad es una generada aleatoriamente para el *Party Mode*.
- Añadir una nueva ventana que nos muestre un top de artistas, álbumes y canciones más escuchados. En esta ventana adicionalmente también se podría poner el número de veces que hemos escuchado cada canción, álbum o artista. Esto permitiría al usuario hacer un seguimiento de sus gustos y abriría la puerta a nuevas funcionalidades que usen estas estadísticas, e. g. generar una *playlist* en función de sus canciones más escuchadas.
- Terminar de implementar el *Party Mode* para dar al usuario una experiencia completa.
- Mejorar la comunicación del servicio de reproducción. La forma de usarlo con un sistema de notificaciones mezclado con los *listener* resulta confuso, además el reproductor se encuentra encapsulado dentro del servicio y este tiene su propio ciclo de vida ajeno a la aplicación lo que dificulta su acceso.
- Generar una documentación completa de la aplicación.
- Realizar pruebas unitarias y de integración de las actividades y los fragmentos.
- Mejorar el algoritmo de aleatorización usando mapas en vez de listas de listas.
- Añadir compatibilidad con Android Auto.
- Añadir que el usuario pueda pasar de canción o reproducir la anterior usando comandos de voz.
- Subir la aplicación resultante a repositorios de aplicaciones como el propio F-Droid.

10. Referencias

[1] Android Studio. https://developer.android.com/studio.

[2] Statistics. https://plugins.jetbrains.com/plugin/4509-statistic.

[3] Rainbow Brackets. https://plugins.jetbrains.com/plugin/10080-rainbow-brackets.

[4] GitHub. https://github.com.

[5] Gateship-One. (2016) https://github.com/gateship-one/odyssey.

[6] Git. https://git-scm.com.

[7] GitHub Actions. https://docs.github.com/es/actions/quickstart.

[8] SonarCloud. https://www.sonarsource.com/products/sonarcloud/.

[9] yEd. https://www.yworks.com/products/yed.

[10] Java. https://www.java.com/es/.

[11] SQLite. https://www.sqlite.org/index.html.

[12] Espresso. https://developer.android.com/training/testing/espresso?hl=es-419.

[13] I. Sommerville, *Software engineering*, 9th ed. Boston: Pearson, 2011, oCLC: ocn462909026.

[14] Kodi. https://kodi.wiki/view/party_mode.

[15] F-Droid. `https://f-droid.org/es/`.

[16] Unpopular Music Player. https://f-droid.org/es/packages/de.kromke.andreas.unpopmusicplayerfree/.

[17] Symphony. https://f-droid.org/es/packages/org.fitchfamily.android.symphony/.

[18] Crayon. https://www.craiyon.com.

[19] Jacoco. https://github.com/jacoco/jacoco.

[20] upload-artifact. https://github.com/actions/upload-artifact.

[21] Ncipollo. https://github.com/ncipollo/release-action.

[22] GitHub del TFG. https://github.com/ocrexx/odysseytfg/tree/master/.github/workflows.

[23] GitHub. https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects.

[24] ScrumDesk. https://www.scrumdesk.com.

[25] Android. https://developer.android.com/topic/libraries/architecture/viewmodel?hl=es-419#java.

[26] Fanart.tv. https://fanart.tv.

[27] MusicBrainz. https://musicbrainz.org.

[28] LastFM. https://www.last.fm/es/.

[29] Android MediaStore. https://developer.android.com/reference/android/provider/mediastore.

[30] Android Room. https://developer.android.com/topic/libraries/architecture/room?hl=es-419.

[31] Freepik. https://www.freepik.es.

[32] Android ToolBar. https://developer.android.com/training/appbar/setting-up?hl=es-419.

[33] Android ViewPager. https://developer.android.com/training/animation/screen-slide?hl=es-419.

[34] Marsaglia, G. (2003). Xorshift RNGs. Journal of Statistical Software, 8(14), 1–6. https://doi.org/10.18637/jss.v008.i14.

A. Anexo A

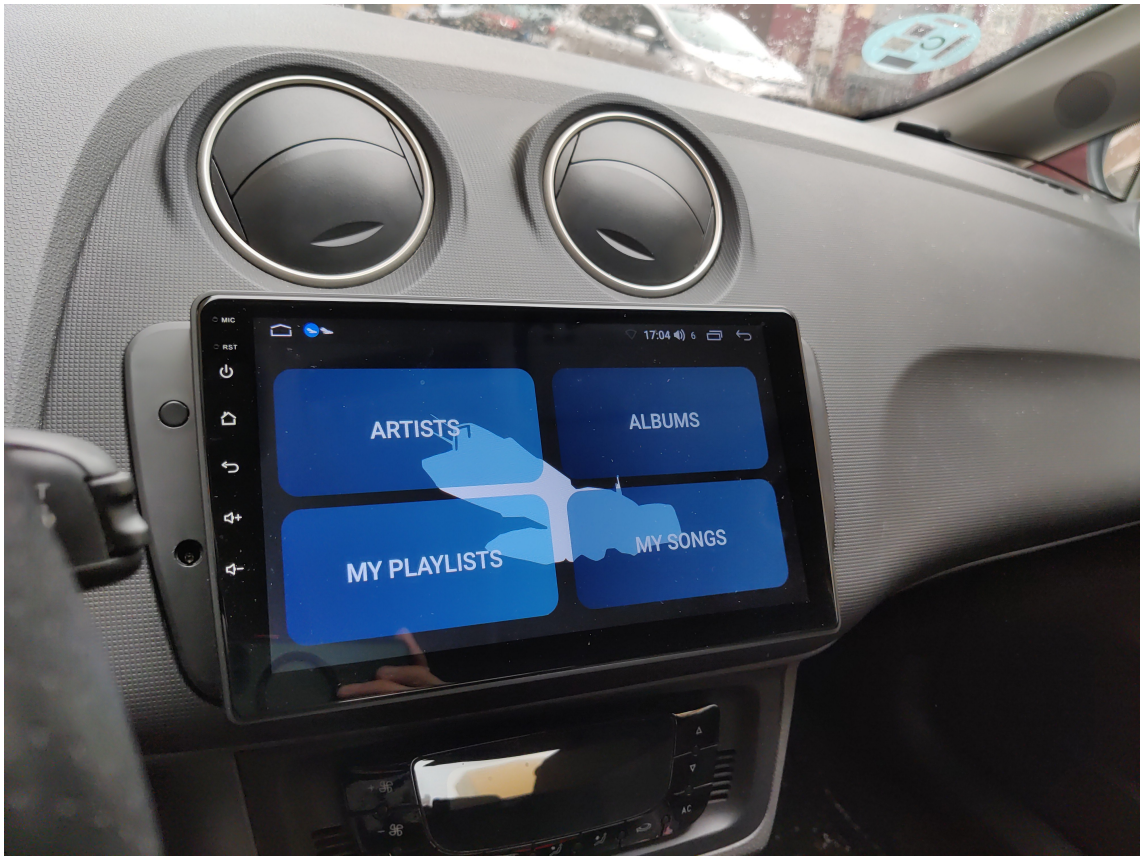


Figura 21: Imagen de la aplicación en uso en un automóvil.

Figura 22: Imagen de la aplicación en uso en un automóvil.