UNIVERSIDAD POLITÉCNICA DE MADRID

Grado en Ingeniería de Computadores

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Sistemas Informáticos

TRABAJO FIN DE GRADO

Aplicación móvil emulador pedal de loop

Autor: *González Bogónez, Raúl*
Tutor: *de Mingo López, Luis Fernando*

Marzo de 2019

Prólogo

Esta documentación trata sobre la aplicación SquareLoops, creada por el alumno Raúl González Bogónez como Trabajo de Fin de Grado. A lo largo de la misma se expondrá el funcionamiento, proceso de creación, toma de decisionesy contextode la aplicación.

Índice

Índice de figuras y diagramas

1. Resumen

Este proyecto consiste en el desarrollo de una aplicación para móviles con sistema operativo Android. A grandes rasgos, la aplicación permitirá al usuario la grabación de audio en varias pistas y su reproducción simultánea en bucle. Básicamente esto describe un pedal de loop o “looper” y, precisamente, la intención de esta aplicación es emular dichos pedales.

Los pedales de loop son usados comúnmente por guitarristas para tocar sobre un acompañamiento grabado por ellos mismos en el momento. Huelga decir que la aplicación va dirigida a guitarristas que desean usar un looper desde la comodidad de su dispositivo móvil.

La aplicación cuenta con una pantalla inicial en la que se muestran cuatro pistas disponibles para grabar, representadas mediante cuadrados con sus correspondientes controles de grabación, silenciado, volumen, borrado y compases. Además, el usuario podrá añadir más pistas hasta un total de ocho. A través de un botón se accede al menú de opciones donde pueden modificarse los parámetros del metrónomo.

SquareLoops será, por tanto, un emulador de un pedal de loop para usuarios de móviles Android.

Abstract

The present project consists of the development of a phone application.Basically, the user will be able to record on several tracks through this application and play these tracks simultaneously in a loop. This idea describes a loop pedal or "looper", and actually the purpose of the application is to emulate a loop pedal.

Loop pedals are usually used by guitarists in order to play on a musical accompaniment recorded by themselves during the performance. Needless to say that the application target is guitarists who want to use a loop pedal through their phone.

The main screen of the application shows four tracks available to record, which are represented by squares with their own controls for recording, muting, volume, deleting and compasses. Furthermore, the user will be able to add more tracks to a total of eight. By clicking an icon, a menu is shown where the metronome parameters can be changed.

Therefore, SquareLoops will bea loop pedal emulator for Android phone users.

2. Introducción

Desde un primer momento, la intención era la de desarrollar una aplicación o programa, puesto que el autor del proyecto deseaba profundizar en la creación de software. Más adelante se profundizará en la decisión de hacer una aplicación sólo para dispositivos móviles.

La idea de una aplicación capaz de emular un pedal de loop nace de la necesidad del autor de la misma de tener acceso en cualquier momento a un looper.

Como consecuencia, nace la idea de aportar a los músicos una herramienta sencilla para poder grabar varias pistas sin depender de un equipo informático o del propio pedal de loop. Por lo tanto, la intención de SquareLoops no es en absoluto la de sustituir a los pedales, sino la de ser utilizada cuando no se tiene acceso a ellos.

Actualmente, existen realmente muy pocas aplicaciones en el mercado que ofrezcan esta funcionalidad, y aún menos gratuitas, lo que supone una motivación extra para llevar a cabo el desarrollo de esta aplicación.

2.1. Pedal de loop

Ya se ha explicado de manera superficial qué es un pedal de loop o looper, pero no qué puede ofrecer realmente y por qué resulta interesante emularlo. En este apartado se profundizará en detalle en qué es, su funcionamiento y se comparará con SquareLoop.

Un pedal de loop es un dispositivo electrónico que permite grabar y reproducir en bucle un instrumento o micrófono que haya sido conectado a él. Estos dispositivos se accionan pisando un pedal (de ahí su nombre), para activar o pausar la grabación, o parar eproducir o detener los bucles o loops ya grabados. Es precisamente porque pueden usarse sin las manos que son muy populares entre guitarristas o teclistas.

Una de sus mayores utilidades es la de grabar una base y probar qué tal funcionan sobre ella diferentes arreglos o melodías. También facilita a un solo músico probar varios instrumentos en conjunto. Es importante destacar que los pedales de loop no se utilizan para grabar canciones, pero sí pueden facilitar en gran medida la composición de las mismas.

Otra utilidad es la de practicar. Con ellos, un músico novato puede practicar fácilmente cómo llevar el ritmo de la canción mientras otros instrumentos están

sonando, e improvisar sobre diferentes bases armónicas grabadas por él mismo.

Pero su uso no se reduce al estudio. También son varios los artistas que los utilizan en conciertos, si se trata de un músico en solitario o del único de la banda que toca ese un instrumento del que necesitan varias pistas.

A continuación se realiza una comparación del funcionamiento de un pedal con el de SquareLoops.

La mayor diferencia es la manera de grabar el audio. Mientras que un pedal lo hace a través de una o dos entradas de Jack o de micrófono, la aplicación deberá hacerlo mediante el receptor del móvil (o un micrófono conectado al mismo si disponemos de él). Esto implica que, con un pedal, sólo se puedan grabar instrumentos electrónicos o se necesiten adaptadores y micrófonos para otros instrumentos (incluida la voz). En cambio, con SquareLoops puede grabarse directamente cualquier fuente de sonido, lo cual encaja perfecto en la idea de sencillez e inmediatez de la aplicación. Obviamente la calidad será inferior, pero, como se ha dicho, este no es el objetivo.

Ocurre lo mismo para la salida del audio. Un pedal requiere de un amplificador o altavoz para reproducir las grabaciones, mientras que SquareLoops aprovecha el altavoz del propio dispositivo. No obstante, sí se recomienda utilizar un altavoz externo más potente.

Con estos datos, podemos decir que SquareLoops facilita mucho su uso, pues sólo se requiere del móvil, el cual solemos llevar siempre con nosotros hoy en día. De esta manera no es necesario tener varios aparatos ni cables.

Otra gran diferencia está en la forma de utilizar ambos productos. Como se ha explicado, para empezar y pausar la grabación en un pedal, hay que pisar el mismo, puesto que su uso con las manos es imposible en muchas ocasiones si se está tocando un instrumento. Sin embargo, como SquareLoops no depende de periféricos, es la propia aplicación la que detiene la grabación de manera automática. Para ello, el usuario debe estipular previamente compases a grabar, tempo, etc. Más adelante se explicará este proceso de manera detallada.

SquareLoops, al disponer de una interfaz, resulta más intuitivo y cómodo de usar que un pedal. Este último cuenta comúnmente con una pantalla en la que simplemente se muestra el número de pista seleccionado, y el cual requiere de combinaciones de pulsaciones entre los dos pedales para navegar por las pistas grabadas.

También es importante el aspecto económico. Un pedal de loop de gama media oscila entre los 150 y 350 €, llegando hasta los 700 € los de gama alta. Por el contrario, la aplicación que se expone es completamente gratuita.

En definitiva, un pedal looper es para un uso profesional que implique grabar para una versión definitiva de la canción o para directos, mientras que SquareLoops es una herramienta enfocada a la composición o entrenamiento en casa o el estudio.

2.2. Conceptos musicales

Para el entendimiento de esta aplicación, es necesario exponer algunos conceptos musicales en los que se basa la misma:

- Metrónomo: Un metrónomo es una herramienta utilizada para marcar el ritmo. Se utiliza comúnmente para ensayar, y en estudio, para mantener el ritmo de una pieza musical. Su funcionamiento es sencillo: cada cierto tiempo emite un sonido corto, como un pitido o chasquido, con una cadencia constante. La duración del silencio entre cada sonido determinará si el ritmo es más o menos rápido. Tradicionalmente los metrónomos eran como el que se muestra en la figura 2, aunque, no obstante, existen metrónomos electrónicos y virtuales. En metrónomos modernos puede configurarse un sonido más destacado que se repetirá cada cierto número de repeticiones con el fin de marcar el principio del compás.

- Pulso: El pulso de una canción o "beat" es una unidad temporal que determina el ritmo de la canción. Este pulso es el sonido que emite un metrónomo y es constante.

- BPM: Siglas de *beats per minute* o beats por minuto. Se refiere al número de beats o pulsos que tienen lugar en un minuto durante una pieza musical. Como la cadencia de los pulsos es constante, si se tienen 60 bpm, se emitirá un pulso cada segundo, y si se tienen 120 bpm, cada medio segundo se producirá un beat. Los bpm son la unidad de medida del tempo en la música.

- Compás: Un compás es la agrupación de dos o más pulsos, es decir, cada cierto número de pulsos se completa un compás y comienza el siguiente. Un compás se representa como una fracción, donde el numerador se refiere al

número de pulsos que componen el compás y el denominador a la duración de esos pulsos. Dicha duración no es una cantidad concreta de tiempo sino que hace referencia a una figura rítmica. Así, y teniendo en cuenta que siempre debe ser potencia de dos, la equivalencia del denominador será:

1 = Redonda.

2 = Blanca.

4 = Negra.

8 = Corchea.

16 = Semicorchea.

Cada figura rítmica dura la mitad de tiempo que la inmediatamente anterior. Así, una negra durará la mitad que una blanca pero el doble de una corchea. Esto se traduce en que un compás de 4/4 estará compuesto por cuatro pulsos al igual que un compás de 4/8, pero el primero durará el doble de tiempo (con igual bpm) puesto que cada pulso es una negra, mientras que cada pulso del 4/8 será una corchea.

Basada en estos conceptos se construye la parte musical de la aplicación. Existirá un metrónomo global que regirá el tempo de todas las pistas que grabe el usuario. Será posible especificar los bpm del metrónomo y la cantidad de pulsos del mismo. En cambio, no es necesario definir el denominador del compás puesto que esta tarea corresponde al músico y en ningún metrónomo común se ofrece esta opción dada su irrelevancia para el funcionamiento de la herramienta. Por último, es necesario dar la posibilidad al usuario de establecer una duración de varios compases dependiendo de la pista en la que está grabando. En definitiva, cada pista grabará un número concreto de compases, que están formados por cierta cantidad de pulsos que se repiten de manera periódica a una velocidad constante.

3. Objetivos

3.1. Objetivos del proyecto

El objetivo principal de este proyecto es el de aprender y ser capaz de desarrollar una aplicación competente funcional mediante un entorno de desarrollo.

Además, tiene otros objetivos tales como:

- Aportar una herramienta útil para músicos.
- Aprender a gestionar elementos de audio desde el punto de vista de la programación.
- Aprender a desarrollar en el entorno de desarrollo elegido.
- Trabajar en un proyecto que ha de ajustarse a unas necesidades específicas, es decir, la aplicación resultante debe cumplir con ciertos requisitos inamovibles para ser capaz de emular un looper.
- Experimentar el proceso de desarrollo de una aplicación de móvil completa.

3.2. Objetivos de la aplicación

La aplicación debe cumplir una serie de requisitos y ser capaz de solventar ciertos problemas para poder ser considerada como herramienta útil y funcional.

En primer lugar, para cumplir su objetivo de emular un pedal de loop debe ser capaz de:

- Grabar varias pistas de audio.
- Concluir de manera automática la grabación, en el momento preciso.
- Reproducir de manera simultánea las grabaciones efectuadas.
- Reproducir las pistas en bucle.
- Mantener la sincronización de las pistas.
- Gestionar las pistas, esto es: añadir nuevas pistas, eliminarlas, silenciarlas…

Además, la aplicación debe cumplir con otros requisitos básicos como:

- Rendimiento óptimo, ofreciendo una respuesta rápida.
- Carencia de “bugs” o errores durante la utilización de la misma.
- Adaptabilidad para dispositivos con distintas resoluciones de pantalla.
- Constancia en el tempo del audio, sin sufrir retardos por falta de rendimiento o mala optimización.

4. Planteamiento del proyecto

4.1. Características de la aplicación

Llegados a este punto, ya se conoce qué hace SquareLoops. No obstante, es necesario exponer de manera detallada las funcionalidades que va a ofrecer al usuario:

- En primer lugar debe grabar distintos audios en pistas separadas, que puedan funcionar y manipularse de manera independiente unas de otras. Es necesario que esta grabación se detenga por sí sola, ajustándose a las especificaciones del usuario y, a partir de ese momento, se reproduzca automáticamente en bucle. Esta es la base de un looper.

- Modificar el volumen de una pista grabada. Esta función permite al usuario obtener una buena mezcla. También debe existir la opción de silenciar por completo una pista y poder recuperar más tarde el volumen al que se encontraba. Esto es útil para aislar momentáneamente otras pistas, pero no perder la configuración del volumen previa.

- Eliminar pistas de audio.

- Incorporar un metrónomo que sirva al usuario como referencia para poder mantener el tempo en sus grabaciones.

- Ajustar parámetros musicales y relativos a la grabación, como los bpm o los beats por compás. Es necesario que cada pista pueda grabar un número específico e independiente de compases a elección del usuario.

- Añadir nuevas pistas si así se requiere.

- Denotar de manera simple e intuitiva lo que está ocurriendo en todo momento, es decir, que el usuario pueda entender de manera fácil cuando una pista se está reproduciendo, cuando se está grabando, etc.

4.2. Inspiración

Para el desarrollo de una aplicación de este estilo, puede ser muy útil tomar como referencia programas similares, puesto que se pretende trabajar con sonido pero de una forma visual. Así, resulta esclarecedor probar algunos programas de éxito en este campo para aprender de sus interfaces y observar qué utilidades ofrecen que podrían ser interesantes en este proyecto. Algunos programas interesantes son:

- FL Studio: FL Studio, anteriormente conocido como FruityLoops, es un software de edición y producción musical con un amplísimo abanico de herramientas y de enorme potencia. Es completamente profesional y uno de los programas más reconocidos en su campo. Si bien su funcionalidad no es en absoluto la misma que la de SqaureLoops, también trabaja con pistas de audio y su forma de tratarlas es muy sencilla.

Figura 3: FLl Studio versión sobremesa (interfaz del Playlist Editor)

FL Studio tiene versión sobremesa, para equipos informáticos, y versión portátil para smartphones y tablets. En esta última versión, la interfaz se simplifica, resultando muy interesante la distribución en la misma de las pistas y los botones de edición y ajustes.

Figura4: Fl Studio versión portátil

- Garage Band: Este es un programa de nuevo de edición música, disponible para Mac yiPads o iPhones. En este caso, nos centraremos en la versión portátil y su editor de pistas.

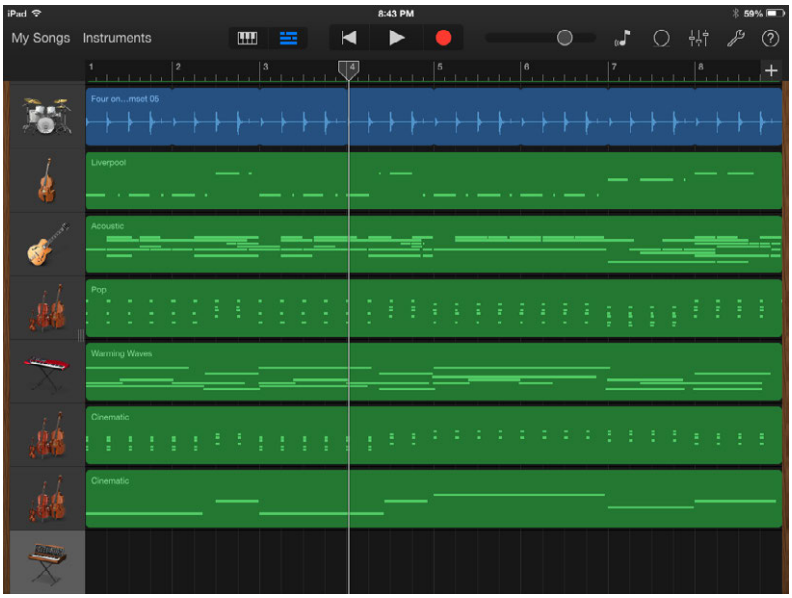


Figura5: Garage Band en iPad (interfaz del editor de pistas)

Su forma de añadir nuevas pistas, editarlas, reproducirlas por separado… resulta muy interesante, así como la señalización de los compases.

4.3. Toma de decisiones

Una vez clara la idea del proyecto y el funcionamiento de la aplicación, así como de los requerimientos de la misma para lograr su cometido, es necesario elegir correctamente a través de qué medio va a utilizarse dicha aplicación y, posteriormente, en qué entorno de desarrollo va a ser creada.

4.3.1. Elección del entorno en el que se va a utilizar la app

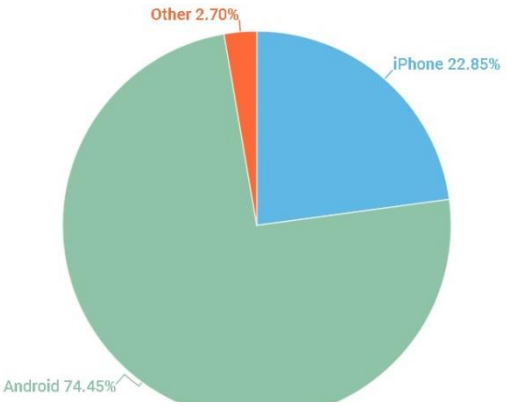
Desde un primer momento se ha hablado del objetivo de desarrollar una aplicación para un dispositivo móvil, concretamente con sistema operativo Android. No obstante, esta propuesta no es banal y se fundamenta en varios puntos.

SquareLoops está enfocada a ser utilizada mayormente en estudio, por lo que, en principio, la portabilidad no es un requisito. Además, el ordenador es una herramienta prácticamente indispensable en un estudio de músico. Sin embargo, SquareLoops pretende ser una herramienta rápida que facilite la composición de piezas musicales o el ensayo, por lo que no tiene nada que ver con la producción musical. Por este motivo, ya que una aplicación móvil resulta mucho más accesible y rápida que una aplicación de ordenador, resultará mucho más cómodo para el usuario utilizar el emulador a través de su dispositivo móvil.

En segundo lugar, y relacionado también con la comodidad de los dispositivos móviles, se encuentra el uso o no de periféricos. Para poder grabar un instrumento musical o la voz en un ordenador, es necesaria una interfaz de sonido y un micrófono, en el caso de recoger el sonido de la voz. En cambio, mediante un móvil es mucho más directo puesto que se utiliza el propio micrófono integrado en el dispositivo. Además, también permite el uso de equipo más sofisticado para conectar un instrumento o micrófono y grabar su sonido. Por tanto, y dado que SquareLoops no está dirigido a músicos con equipo, resulta más práctica la opción del dispositivo móvil.

Ahora bien, ¿por qué sólo para dispositivos Android?

Está claro que actualmente el mercado de los smartphones se reparte entre dispositivos con sistema operativo Android y dispositivos con sistema operativo iOS, como puede verse en la figura 5. Por lo tanto esta aplicación estará orientada a uno de estos dos sistemas o a ambos.



Ni Apple ni Google ofrecen una opción de programación multiplataforma, por lo que es necesario el uso de programas de terceros que sí cuenten con esta característica. Sin embargo, este tipo de software no es de acceso libre por lo que no es una opción para este proyecto. En esta misma línea, es necesario un dispositivo con OSX para desarrollar una aplicación dirigida a dispositivos de Apple, a los que no se tiene acceso para llevar a cabo este proyecto.

Otra razón de peso para elegir desarrollar una aplicación para Android es el dominio del sistema operativo frente a su rival. En el año pasado, la cuota de mercado de Android llegó prácticamente al 75%, y en los último años esta es la tendencia. Por otro lado, el mercado de aplicaciones Android es mucho más abierto que la Apple Store, llegando a haber en torno al doble de aplicaciones en la Play Store de Google.

En definitiva, la opción final de dispositivo en el que se utilizará SquareLoops es un dispositivo móvil con sistema operativo Android.

Elección de entorno de desarrollo

La elección de un entorno de desarrollo adecuado es muy importante en el proceso, ya que debemos elegir un entorno que se adapte correctamente a las necesidades de nuestro proyecto. Cada entorno presenta distintas características que pueden sernos útiles o, por el contrario, dificultar nuestro trabajo, por lo que debemos fijarnos en los siguientes puntos para elegir el IDE (Integrated Development Environment), o entorno de desarrollo integrado, adecuado:

- Lenguajes de programación que soporta.
- Herramientas de depuración que ofrece.
- Extensible mediante plugins.
- Interfaz práctica que favorezca nuestro trabajo.
- Librerías que incluye.
- Permite importar/exportar códigos de/a otros IDEs.
- Utilidades para eldesarrollo de la interfaz del programa.
- Desarrollo de software multiplataforma.
- Sistemas operativos para los que está disponible.

Para nuestro proyecto, cabe destacar que el IDE elegido nos ofrezca un diseñador de interfaz sencillo y, por supuesto, que permita enfocar la aplicación para ser utilizada en móviles. Para cumplir con este requisito, el entorno de trabajo escogido debe permitirnos adaptar de manera sencilla la interfaz de nuestra aplicación para la resolución de las distintas pantallas de los dispositivos móviles, implementar opciones tales como el pulsado y gestos táctiles, los botones del propio dispositivo, etc., así como generar los archivos pertinentes para la instalación y funcionamiento de la aplicación en un móvil.

Con toda la información vista hasta ahora, se barajan dos posibles entornos de desarrollo: Oracle Eclipse y Android Studio.Ambos entornos están perfectamente capacitados para desarrollar una aplicación móvil.

Eclipse es una primera opción, ya que se ha empleado con frecuencia durante la carrera en distintos proyectos. Sin embargo, al analizar ambos entornos, queda patente que Android Studio es superior en todos los sentidos a Eclipse:

- Lo primero es que está diseñado para desarrollar aplicaciones en Android.
- Aunque nuestra interfaz va a ser sencilla, resulta mucho más cómodo diseñarla en Android Studio.

- Es muy sencillo probar la aplicación en distintos dispositivos con diferentes versiones Android y resoluciones de pantalla gracias a que se pueden crear varios emuladores. Además consumen menos recursos.
- Utiliza Gradle, con las ventajas que eso conlleva.
- En un primer contacto, resulta muy intuitivo y sencillo de trabajar con él.
- Al trabajar con Android Studio, queda claro que el rendimiento es mayor y es más rápido.
- La generación de .apk es más fácil.
- Eclipse está cada vez más en desuso para desarrollo de aplicaciones móviles, mientras que Android Studio es lo más popular.

Puesto que Android Studio supera a Oracle Eclipse en todas las facetas, la elección de IDE para desarrollar el proyecto es Android Studio.

5. Desarrollo

5.1. Estudio de mercado

Como se ha dicho, existen muy pocas aplicaciones en el mercado de las aplicaciones para smartphones que ofrezcan lo mismo que SquareLoops. No obstante, es interesante conocer el funcionamiento de alguna de ellas:

- LoopStation: Al iniciar la aplicación por primera vez, se pide calibrar la grabación. Para ello realiza una captura de audio y se muestra la señal recogida mediante una onda. Para completar la sincronización se deben hacer coincidir los "beats" mostrados en la onda con unas líneas rojas en el fondo. Cabe destacar que este sistema no es especialmente eficaz y la aplicación presenta problemas a la hora de capturar el audio en el momento exacto.

Tras la sincronización se muestra la que será la pantalla principal. En ella se puede grabar hasta seis pistas con un simple toque en la pista deseada para iniciar la grabación. Para eliminar una pista, ésta debe ser arrastrada hacia la parte inferior de la pantalla. Estas pistas se representan visualmente mediante una onda de audio. En la parte inferior de la pantalla se encuentra la configuración del metrónomo, pudiendo modificar los *beats* por minuto o bpm, número de compases y número de *beats* por compás, así como iniciar o detener el propio metrónomo.

En la esquina superior izquierda hay un botón para desplegar un menú contextual a través del cual se podrán guardar proyectos, cargarlos, obtener una versión de pago sin publicidad, sincronizar con Facebook y acceder a un menú de configuración. En este menú puede repetirse la calibración de la grabación y, para la versión de pago, llevar a cabo más ajustes en el metrónomo o incluir reverberación en las pistas.

Por último, el método de grabación de la aplicación consiste en grabar constantemente mientras la aplicación está operativa y, más tarde, detener la grabación cuando una pista termina de grabar y recortarla para volver a empezar la grabación. Este método puede resultar poco seguro puesto que, mientras la aplicación está activa todo lo que ocurre alrededor del usuario está siendo grabado, pero es una manera de intentar solventar los problemas de sincronización relativos al uso de la grabadora que presentan los dispositivos Android.

- Looper: Esta es una aplicación muy sencilla en sus funciones pero muy eficiente. En la pantalla principal aparecen seis esferas, cada una de las cuales

representa una pista de audio, excepto la primera. Mediante esta esfera se pueden detener o reproducir de manera simultánea todas las grabaciones que se hayan hecho. Además, manteniendo pulsado sobre ella se abre un submenú en el que se pueden modificar los beats de cada compás y los bpm.

Al pulsar sobre alguna de las otras esferas comienza a grabar. Tras completarse un compás, la grabación se detiene y el audio grabado se reproduce en bucle. Dentro de la esfera se representa el sonido recogido mediante una onda. Manteniendo pulsado sobre la esfera se abre un menú en el que se puede nivelar el volumen de la pista y el número de compases a grabar, si aún no contiene una pista. Haciendo una pulsación sobre una esfera ya grabada se silencia su pista correspondiente.

Por último, una función bastante interesante; al arrastrar una esfera sobre otra, sus grabaciones se fusionan en una y la nueva esfera controlará ambas grabaciones.

Aunque es bastante simple en sus funciones, esta aplicación cumple muy bien con su cometido. Sin embargo, ya no se encuentra en la Play Store.

5.2. Descripción del entorno de desarrollo

Según la página web oficial de Android Studio, *Android Developer*, "*Android Studio es el entorno de desarrollo integrado (IDE) oficial para el desarrollo de aplicaciones para Android y se basa en InteliJ IDEA*". Fue presentado como el entorno oficial de desarrollo Android en 2013 en la conferencia Google I/O, sustituyendo a Eclipse como IDE oficial. Las características principales del entorno son:

- Compatibilidad con distintos sistemas operativos tales como Windows, GNU/Linux y MacOS.
- Entorno unificado con la capacidad de desarrollar para cualquier dispositivo Android.
- Compilación basada en Gradle.
- Compatibilidad con C++ y NDK.
- Emulador de varios dispositivos virtuales Android (ADV) para testear la aplicación.
- Consola de programador.
- Editor de diseño gráfico.
- Herramientas y frameworks de prueba.

5.2.1. Descripción de la interfaz

Durante la elaboración de un proyecto, la Interfaz de Usuario general es la que se muestra en la figura.

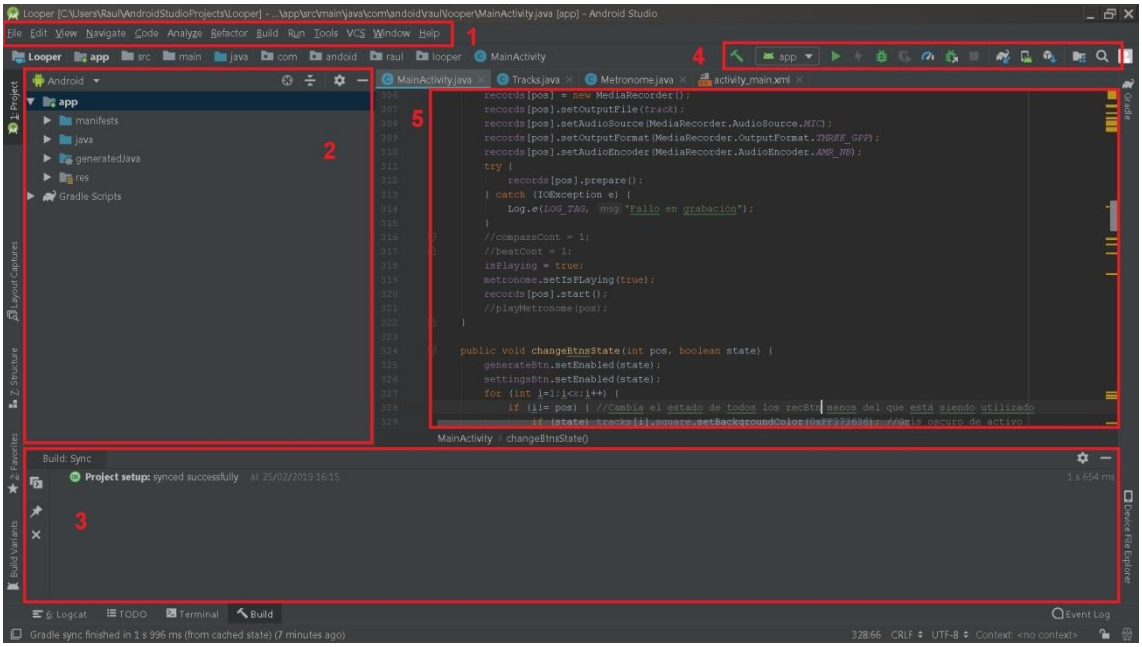


Figura7: Interfaz de Android Studio

Contiene los siguientes elementos:

1- Barra de tareas superior: A través de los distintos submenús se puede acceder a todas las opciones y herramientas que ofrece el programa.

2- Navegador de archivos: Esta sección da acceso a los distintos directorios y archivos que componen el proyecto. Es posible organizar la vista de estos archivos según las preferencias del usuario. En la imagen se muestra la vista más común, mediante la cual se ofrece una vista rápida de los archivos más relevantes para un proyecto:

- Manifest: Contiene el archivo AndroidManifest.xml.

- Java: Contiene los códigos fuentes de la aplicación.

- Res: Contiene todos los recursos tales como imágenes, sonidos, archivos .xml de la interfaz...

3- Barra de herramientas: Desde aquí pueden visualizarse los errores lanzados por la compilación, fallos de ejecución del emulador, el registro del log, un terminal y una serie de información útil para depurar el código.

4- Accesos rápidos: Son accesos rápidos referentes a acciones relacionadas con la compilación y la depuración del código así como con el emulador de dispositivos.

5- Ventana del editor: Aquí se muestra el código tanto Java como XML. También muestra el editor de interfaz gráfica si el programador lo prefiere frente al código XML.

5.2.2. Estructura de un proyecto en Android Studio

En un proyecto de Android Studio existen distintos tipos de archivos que intervendrán en el funcionamiento de la aplicación, algunos de los cuales son estrictamente necesarios y son creados por defecto al iniciar un nuevo proyecto. Se pueden dividir en tres grandes categorías: archivos .java, .xml y otros recursos. Los recursos son archivos externos que utiliza el código en Java y algunos de ellos también son archivos XML: layouts y values.

Archivos .java

Son los archivos Java Class que contienen el código fuente de la aplicación, al igual que en cualquier programa basado en lenguaje Java. Sin embargo, existe una peculiaridad; las activities.

Una activity es un archivo Java Class con un layout asociado. El archivo layout contiene la información en XML sobre la interfaz gráfica como se verá más adelante.

Cada activity gestiona una ventana, a través de la cual el usuario interactuará con la aplicación, por lo tanto toda aplicación debe contar con, al menos, una activity inicial, la cual será llamada nada más ejecutar la aplicación. Existen una serie de activities prefabricadas que Android Studio ofrece al usuario con distintas composiciones de la interfaz las cuales, no obstante, pueden ser personalizadas a gusto del programador y que pueden agregarse haciendo click derecho en la carpeta que contiene los Java Class y seleccionando new\Activity. También puede crearse una nueva clase desde new\Java Class y convirtiendo esta nueva clase en activity mediante código. Además, habrá que crear el layout asociado y declarar la nueva activity en el archivo AndroidManifest.xml. Para llamar a una activity secundaria basta con ejecutar la función startActivity() pasando como parámetro un Intent cuyos parámetros, a su vez, serán el contexto de la activity actual y la activity en cuestión que se desea ejecutar.

Por último, hay que tener en cuenta el ciclo de vida de las activities. Esto es importante para evitar dejar activities abiertas, desincronización de datos o, incluso, "flasheos".

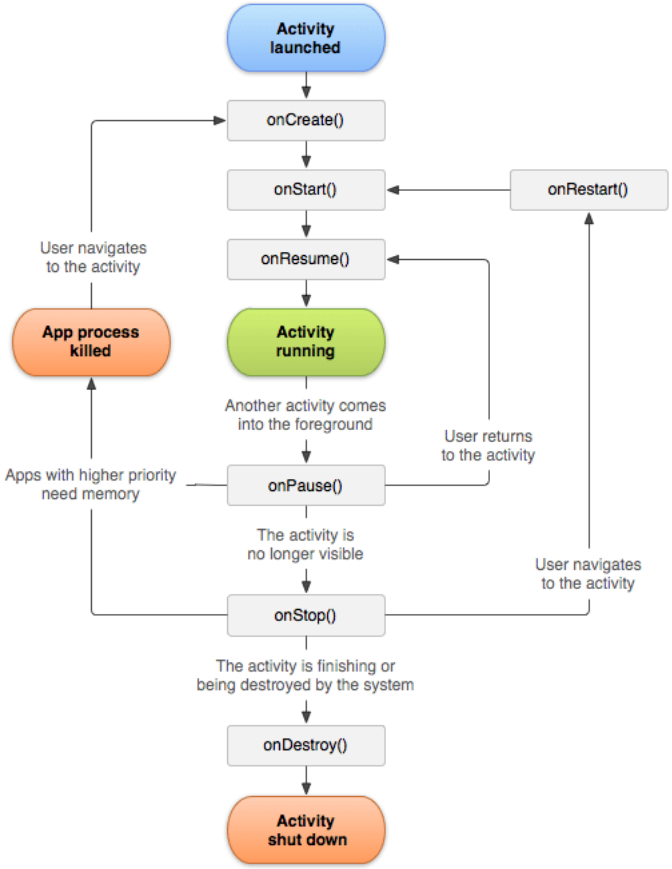


Figura8: Ciclo de vida de una activity

La finalidad de cada uno de los métodos se explica a continuación:

OnCreate:Nada más iniciar una activity se llama a su método onCreate, en el cual se configura la pantalla principal de la activity y, si procede, se declaran views, se enlazan los datos a las listas, etc. A este método se le pasa un parámetro tipo Bundle, consistente en un diccionario para guardar e intercambiar información acerca de los objetos y estados.

OnStart:Justo antes de que la activity sea visible puede ser llamado este método onStart, mediante el cual se realiza alguna tarea que sea necesaria de llevar a cabo justo antes de que es haga visible dicha activity. Un ejemplo muy claro de este método es el de los videojuegos, que necesitan actualizar los datos a su último estado justo antes de abrir una nueva ventana la cual va a ser construida a partir de esos parámetros.

OnPause: Se llama a este método justo antes de poner la activity en cuestión en segundo plano. La utilidad de esta función es la de limpiar datos u objetos, liberar recursos o para confirmar el guardado de datos que hayan cambiado.

Por el contrario, tiene una desventaja y es que hasta que este método no devuelva un valor no podrá iniciarse otra activity. La activity también puede ser destruida desde este método.

OnResume: Se llama a onResume cuando el usuario es quien interactúa con una activity que se encuentra en pausa.

OnStop: Se debe llamar a esta función antes de que la activity sea destruida con el objetivo derealizar las tareas necesarias previas a dicha destrucción. Esto ocurre, sobre todo, cuando la actividad destruida dejar de ser visible para el usuario debido a la aparición de otra ventana en la pantalla. Además, es muy importante llamar a onStop cuando se produce un cambio de orientación para reconstruir correctamente la activity, ya que cuando se da esta situación todas las activities son destruidas y reconstruidas, con lo que pueden perderse los cambios no guardados de la interfaz, por ejemplo, los datos de los formularios.

OnRestart:Sólo se llama a onRestart cuando una activity ha sido detenida y va a ser iniciada de nuevo y siempre debe seguirse de onStart. Se llama a onRestart cuando, en esta situación, se necesitan ejecutar una serie de tareas antes de iniciar de nuevo la activity. Esto ocurre sobre todo cuando la actividad se ha detenido pero no ha sido destruida.

OnSaveInstanceState: La función de este método es muy sencilla: salvar los datos de la activity antes de que se produzca un cambio.

OnDestroy: Es el último método del ciclo de vida de una activity puesto que la mata por completo. Todos los datos de estado de dicha activity se eliminan por completo y el sistema operativo se libera de ella.

Archivos .xml

XML (Extensible Markup Language) no es un lenguaje de programación propiamentedicho, sino un metalenguaje empleado para definir lenguajes de marcas. Lapotencia de XML reside en que permite el intercambio de información entredistintas plataformas, siendo por lo tanto especialmente útil en manejo de bases de datos. No obstante, tiene innumerables aplicaciones, como en el caso de AndroidStudio.

Para manejar los archivos XML del proyecto es importante conocer cómofuncionan. Estos archivos se basan en *elementos* que, a su vez, sedefinen mediante etiquetas. Android Studio ya cuenta con una serie de etiquetasdefinidas, así, para crear un botón en un archivo de layout se escribirá:

```
<Button
.../>
```

La estructura básica de un archivo .xml cuenta con un prólogo y un cuerpo:

-Prólogo: No es obligatorio pero común incluirlo para declarar la versión XML y eltipo de documento. Por ejemplo:

```
<?xml version = "1.0" encoding = "UTF-8"?
```

Además, también es posible incluir unas líneas de comentario explicando lo pertinente acerca del archivo.

- Cuerpo:Debe declararse en él un elemento raíz que, a su vez, contendrá el resto de elementos para realizar las acciones precisas. Un elemento raíz sería:

```
<LinearLayout xmlns:android
="http://schemas.android.com/apk/res/android"
...
</LinearLayout>
```

A su vez, un elemento puede contener más elementos o caracteres. Además, también tienen ciertos atributos, los cuales son propiedades que dan forma a los elementos. Estos atributos siempre van entre comillas, por ejemplo:

```
android:layout_width="match_parent"
```

Basado en estos principios básicos del lenguaje, se pasa a explicar cada tipo de archivo .xml en un proyecto Android:

- AndroidManifest.xml: Se encuentra en la raíz de la aplicación y modificándolo pueden establecerse diversas configuraciones básicas de la aplicación. La forma más sencilla y recomendada de manipularlo es a través de código desde el propio archivo .xml, aunque también existe la posibilidad de hacerlo a través de una interfaz gráfica. Las configuraciones más importantes que se ejecutan a través de AndroidManifest.xml son:

- Declarar todas las activities de la aplicación.

- Escribir la versión de la aplicación según se actualice la misma.

- Conceder ciertos permisos a la aplicación tales como acceso al almacenamiento de memoria externo, acceso a la galería, activar el micrófono, etc. La declaración de todos los permisos que se requieran es esencial para el funcionamiento de la aplicación en un dispositivo, aunque deberá ser el usuario quien acepte dar estos permisos a la aplicación.

- Establecer el icono que se mostrará en el menú de aplicaciones del dispositivo.

- Elegir el tipo de orientación de la pantalla.

- Layouts: Dentro de la carpeta *layout* se encuentran las layouts que componen la interfaz gráfica de una aplicación. Dentro de estos archivos se declaran los elementos que formarán esa vista, teniendo en cuenta el orden y utilizando diferentes recursos para ordenarlos que se explicarán más adelante. El elemento raíz de un layout debe ser siempre, como es lógico, un elemento layout. Existen tres tipos:

- LinearLayout: Los elementos contenidos en este layout se colocan de manera lineal, vertical u horizontalmente según se determine en el atributo *orientation* de LinearLayout.

- WebView: Es un layout que facilita la creación de contenido con formato web.

- RelativeLayout: Este layout ofrece la mayor personalización de la interfaz gracias a que los elementos contenidos en él toman su posición respecto a los demás elementos. Lo más importante que aporta esto es que se puede diseñar una interfaz que no pierda su formato independientemente del tamaño o la resolución de la pantalla, pues las posiciones y tamaños de los elementos no dependen de coordenadas ni pixeles sino de posiciones y tamaños relativos.

Es común combinar un RelativeLayout con varios LinearLayout, ya que permite colocar los LinearLayout de manera relativa entre sí y, dentro de estos, varios elementos seguidos ordenados vertical u horizontalmente. Para otorgar un tamaño relativo a los elementos se usan los atributos *match_parent* y *wrap_content*, que proporcionan el tamaño máximo permitido por el padre del elemento o el tamaño mínimo para que siga siendo funcional respectivamente.

Los elementos tales como botones, formularios, etc. se denominan views. Algunas de las views más comunes son: TextView, EditText, Button, los propios layouts, ImageView..., pero existen muchos más. Cada view tiene una gran cantidad de atributos, muchos de los cuales son exclusivos de cada view ya que hacen referencia a características únicas de éstos. Por ejemplo, un SeekBar posee el atributo progress que hace referencia a la posición del puntero.

Por supuesto, todos los elementos que conforman un layout.xml pueden ser manipulados desde un Java Class, lo que se conoce como manipularlos "programaticalmente". Lo primero es asociar el layout a un activity para que se

muestre en pantalla cuando ese activity se esté ejecutando. Para ello se usa la siguiente línea de código:

```
setContentView(R.layout.id_del_layout);
```

Donde *R* es la carpeta *res*,*layout* la carpeta homónima, y *id_del_Layout* el atributo id establecido desde XML que precisamente sirve para referenciar un view desde java. Esta línea de código debe ser incluida en el método *onCreate* de la clase. Para manipular los views de ese layout se debe declarar un objeto del mismo tipo que el view en cuestión y asociarlo a él con la siguiente línea de código:

```
objeto.finViewById(R.id.id_del_view);
```

Donde *id_del_view* es el id otorgado a ese view desde el archivo .xml. También es posible crear una nueva instancia de un view y colocarlo dentro de un layout de manera programatical. Esto es gracias a que los objetos tienen una serie de métodos que permiten acceder a sus atributos desde un código java. Un view también posee métodos para interactuar con el usuario. Por ejemplo, el método *onClick* de un Button ejecutará una acción cuando el botón sea pulsado por el usuario.

Por último, existe la posibilidad de crear un view personalizado, es decir, crear un elemento para la interfaz gráfica con los atributos requeridos por el programador.

Para ello debe crearse un nuevo Java Class que extenderá de *View*. Esta clase necesita recibir en su constructor un parámetro *Context* y un *AtributeSet* en el caso de crear la view desde XML. El parámetro de tipo *AtributeSet* establecerá los atributos establecidos desde XML. Para dimensionar la view programaticalmentedebe implementarse el método *onMeassure* que establecerá un *match_parent*, *wrap_content*¸o píxeles específicos según requiera el programador.

Esta view personalizada se gestionará como cualquier otra view, tanto desde código Java como desde XML.

- Values: Dentro de la carpeta *Values* hay tres archivos .xml más.

El primero es *colors.xml* donde se establecen los colores predeterminados de la aplicación: *colorPrimay*, *colorPrimaryDark*, y *colorAccent*. También pueden crearse nuevos colores predeterminados para su uso futuro.

El siguiente archivo es *strings.xml,* que sirve para predefinir cadenas de caracteres que se usarán más adelante en la aplicación. Esto es especialmente útil y potente para crear una aplicación en varios idiomas, ya que resulta muy sencillo sustituir los textos.

Por último, está el archivo *styles.xml*. Mediante este archivo se crean temas que pueden utilizarse para establecer la apariencia de un activity, esto incluye colores, transparencia, márgenes, etc. Además, pueden crearse nuevos estilos de las views prediseñadas de Android o de las views personalizadas, cambiando gran cantidad de atributos que de otra forma resulta imposible personalizar.

Otros recursos

El resto de recursos se encuentran en las carpetas *drawable*, *minimap* y *raw*, donde se guardan las imágenes que van a usarse, los iconos de la aplicación para las distintas resoluciones y los archivos de audio respectivamente.

5.3. Planificación de la interfaz

En primer lugar se realiza un esquema de cómo debería ser la aplicación, es decir, todas las pantallas con las que va a interactuar el usuario y cómo va a hacerlo.

Nada más abrir la aplicación se mostrará la pantalla tal y como se puede ver en la figura9. Habrá cuatro pistas o "tracks" representados por un cuadrado cada uno.

Figura9: Pantalla inicial

Todos estos cuadrados tendrán, a su vez, un control para definir el número de compases que van a grabarse en esa pista. Tras pulsar el botón del centro del cuadrado se iniciará la grabación y se producirá un cambio en la interfaz.

El botón en la parte superior izquierda añadirá un cuadrado nuevo en la zona inferior de la pantalla. La barra de la parte superior representa el metrónomo

que, por el momento, está vacía pues aún no se ha inicializado. En último lugar, el botón a la derecha del metrónomo abrirá un submenú desde el que modificar ciertas preferencias de la aplicación.

El submenú de ajustes tendrá el aspecto que se muestra en la figura 10. Deberá ser más pequeño que el tamaño máximo de la pantalla para dar esa sensación de menú contextual.

Figura10: Submenú de ajustes

El primer ajuste se refiere al número de bpm, que se podrá introducir de manera manual o aumentar o disminuir a través de un par de botones al lado del cuadrado. La siguiente opción cambia los beats de los que estará compuesto un compás y podrá modificarse de igual manera que los bpm. Los dos últimos ajustes sólo tienen dos estados: activado o desactivado. Así se podrá activar un compás que sonará antes de empezar la grabación para permitir al músico seguir el tempo. La otra opción activará o desactivará el sonido del metrónomo.

Los botones de la parte inferior del menú sirven para cerrar dicho menú guardando los cambios, a través del botón aceptar, o sin guardar los cambios a través del botón cancelar. Además, al pulsar fuera del submenú, éste deberá cerrarse sin guardar los cambios, siguiendo una lógica intuitiva para el usuario.

Una vez ajustadas las preferencias a las necesidades del usuario, el siguiente paso que seguirá es grabar en una pista. Como se ha dicho, para empezar a grabar se debe pulsar el botón del centro de una pista. Tras esto, los demás cuadrados, así como sus botones, y los botones de "añadir pista" y "ajustes" se desactivarán momentáneamente. Al ser el primer *track* que se graba, se inicializará el metrónomo, comenzando a sonar y representándose visualmente con un llenado de la barra superior. Además, esta barra se dividirá en varias secciones dependiendo del número de beats que compongan un compás. Si el

compás de entrada está activo, se reproducirá un compás entero sin que ocurra ningún cambio más en la pantalla a excepción de la barra de metrónomo que se llenará. Cuando comience el nuevo compás, la barra de metrónomo volverá a llenarse desde cero y se inicializará la grabación.

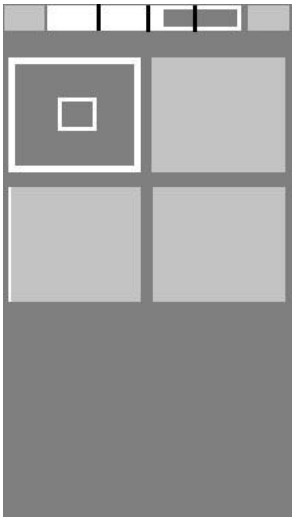


Figura11: Grabación

Si la opción de *sonido del metrónomo* está inhabilitada no se reproducirá el sonido del metrónomo pero la barra se llenará igualmente. En cualquier momento el usuario podrá pulsar sobre el botón central del cuadrado para abortar la grabación, de tal manera que la interfaz volverá a su estado inicial y no se guardará ningún archivo de audio.

Una vez comienza la grabación, el cuadrado correspondiente a la pista que se está grabando comenzará a parpadear para indicar que la grabación está en proceso.

Cuando la grabación concluye se restaura el aspecto original de la interfaz. El cuadrado que contiene la grabación empezará a llenarse con un color único para cada pista. Ya que el número de compases a grabar es independiente de cada track, no todos se llenarán por completo al unísono. Cuando el audio finaliza, el cuadrado se limpia y comienza a llenarse de nuevo mientras que se vuelve a reproducir desde el principio el archivo, generándose así el bucle.

El botón central del cuadrado desaparecerá y en su lugar surgirán dos botones en la parte superior. El primero silenciará completamente la pista o restablecerá el volumen. El segundo botón se encargará de eliminar el archivo de audio correspondiente a la pista y de restaurar el aspecto original del cuadrado, devolviéndolo a la apariencia de un track vacío listo para grabar.

En la parte inferior del cuadrado una barra controlará el volumen del audio de la pista. La opacidad del color que rellena el cuadrado dependerá, de manera proporcional y progresiva, del volumen al que se encuentre, llegando al máximo nivel de opacidad cuando el volumen se encuentre al cien por cien.

Para grabar en cualquiera de los otros tres cuadrados el procedimiento será el mismo, con la diferencia de que el metrónomo ya está inicializado.

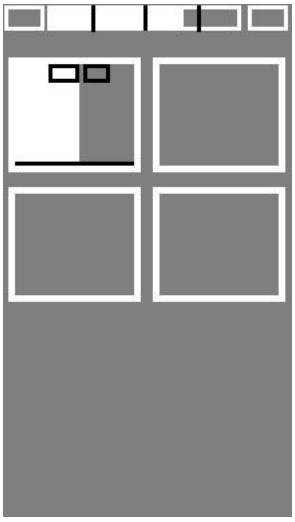


Figura12: Reproduciendo una pista

Mientras haya al menos un track reproduciéndose (aunque esté silenciado) no se podrán modificar los bpm ni los beats por compás, aunque sí podrá habilitarse o deshabilitarse el compás de entrada y el sonido del metrónomo.

Para explicar el proceso de grabar un segundo track va a suponerse que éste se graba en una pista añadida, para poder observar también las diferencias entre un track inicial y unoañadido.

Existe un límite de ocho tracks simultáneos como máximo, por lo que se podrán añadir cuatro más.

Como se ha dicho, para agregar un nuevo cuadrado se debe pulsar en el botón de la esquina superior izquierda, apareciendo automáticamente un nuevo cuadrado en el siguiente hueco disponible. Los cuadrados siempre tienen el mismo tamaño, por lo que si no caben todos en pantalla bastará con deslizar hacia abajo para ver los demás tracks.

El nuevo cuadrado será exactamente igual a los predeterminados con la diferencia de que aparecerá en su parte superior derecha un botón. Este botón sirve para eliminar de la pantalla el cuadrado y siempre será posible pulsarlo.

Para grabar una nueva pista en el track que se ha creado simplemente habrá que pulsar su correspondiente botón de grabación.

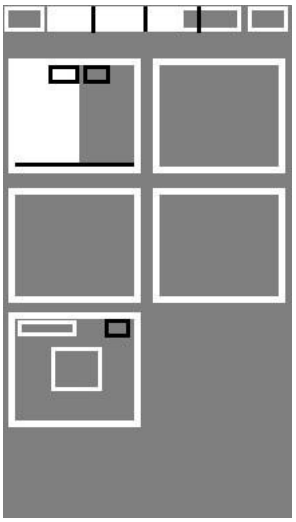


Figura13: Nuevo track

Sin embargo, el compás de entrada no comenzará hasta que acabe el actual compás o, en su defecto, si el compás de entrada está desactivado, no comenzará a grabar hasta el siguiente compás. Esto es de vital importancia para mantener sincronizados los tracks, puesto que todos deben mantener el tempo de la canción.

Tras completar la grabación, el nuevo track se mostrará igual que el primer track que ya contiene un archivo de audio, teniendo en cuenta que este nuevo track puede eliminarse mediante el botón de su esquina superior derecha. Si esto ocurriera, no sólo se eliminaría de la pantalla, sino que debe detenerse y eliminarse el audio correspondiente. En este caso, como sólo existe un track añadido, no hay que reordenar nada. A continuación se explica qué ocurriría si se eliminase un track existiendo tras él más cuadrados añadidos.

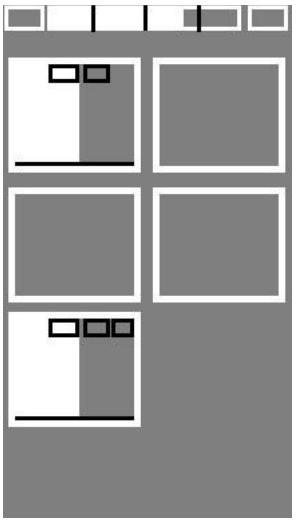


Figura14: Track añadido grabado

Como se aprecia en la figura 15, ahora existen tres cuadrados extra. Si se eliminase el quinto cuadrado, los dos siguientes deberían recolocarse tal y como se muestra, manteniendo siempre las preferencias y características propias de cada unode ellos.

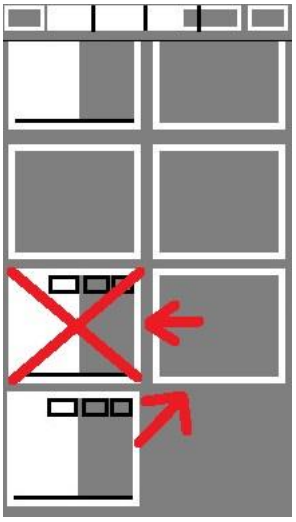


Figura15: Eliminar track

Tras la reorganización, la pantalla quedaría como en la figura 16. Si en algún momento se eliminasen todos los archivos de audio, el metrónomo se detendría, tanto en sonido como visualmente, y se podría modificar de nuevos los bpm y los beats.

Teniendo en cuenta las necesidades y funcionamiento de la aplicación anteriormente descritos, va a utilizarse un diagrama de clases en UML con el objetivo de simplificar la tarea de crear las clases Java necesarias para desarrollar el proyecto en Android Studio.

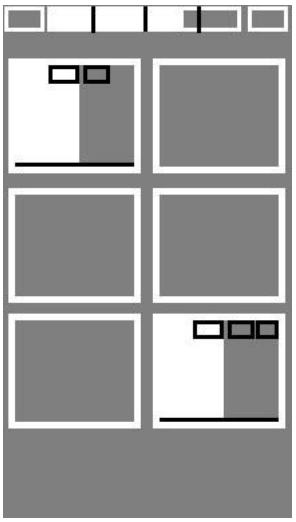


Figura16: Track eliminado

5.4. Diagramas

5.4.1. Diagrama de clases

Un diagrama de clases es un esquema que muestra las clases, atributos, operaciones y relación entre objetos de un sistema concreto. UML (Lenguaje Unitario de Modelado) es un modelo de estandarización que reúne una serie de pautas para facilitar la creación de un diagrama de clases desde el punto de vista de la programación orientada a objetos.

En UML, una clase se representa mediante un rectángulo dividido en tres secciones:

- Sección superior: Corresponde al nombre de la clase.

- Sección central: Incluye todos los atributos de la clase. Esta sección no es siempre necesaria, pues sólo se hace referencia a una instancia específica de una clase.

- Sección inferior: Aquí se enumeran todos los métodos y operaciones que lleva a cabo la clase.

El nivel de acceso de las clases también se ve representado en el diagrama. Dependiendo de su visibilidad, le corresponde un símbolo único para representarlo:

+ Público.
- Privado.
Protegido.
~ Paquete.
/ Derivado.
Si está subrayado es estático.

Estas clases pueden relacionarse entre sí, dotando de gran potencia al diagrama de clases. Estas relaciones también tienen su propia simbología para identificarlas:

- Herencia: Se represente mediante una línea recta acabada en una flecha cerrada que parte del hijo y señala al padre. El hijo hereda todos los atributos y métodos del padre.

- Asociación bidireccional: Se simboliza con una línea recta entre las dos clases implicadas. Esta es la relación básica entre dos clases en la que ambas son conscientes la una de la otra y de la propia relación.

- Asociación unidireccional: Se representa con una línea recta acabada en una punta de flecha abierta, que nace de la clase conocedora y señala a la otra clase. En esta relación solo una clase es consciente de la relación y de la otra clase e interactúa con ella.

Como se ha visto, los diagramas de clase se utilizan para planificar la estructura de las clases de un proyecto y la relación entre ellas. Además aportan una serie de beneficios a la hora de gestionar el proyecto:

- Representar las necesidades específicas de un sistema.

- Facilitar la exposición del proyecto al resto de miembros en un equipo de trabajo.

- Evidenciar código específico que será necesario para el desarrollo de la aplicación.

- Ilustrar modelos de datos simples y complejos.

- Simplificar la visión general del sistema de clases.

- Encontrar soluciones sencillas a través de las relaciones establecidas entre clases.

Basado en los conceptos explicados sobre los diagramas de clases en UML se ha diseñado el diagrama correspondiente al desarrollo en Android Studio de la aplicación SquareLoops.

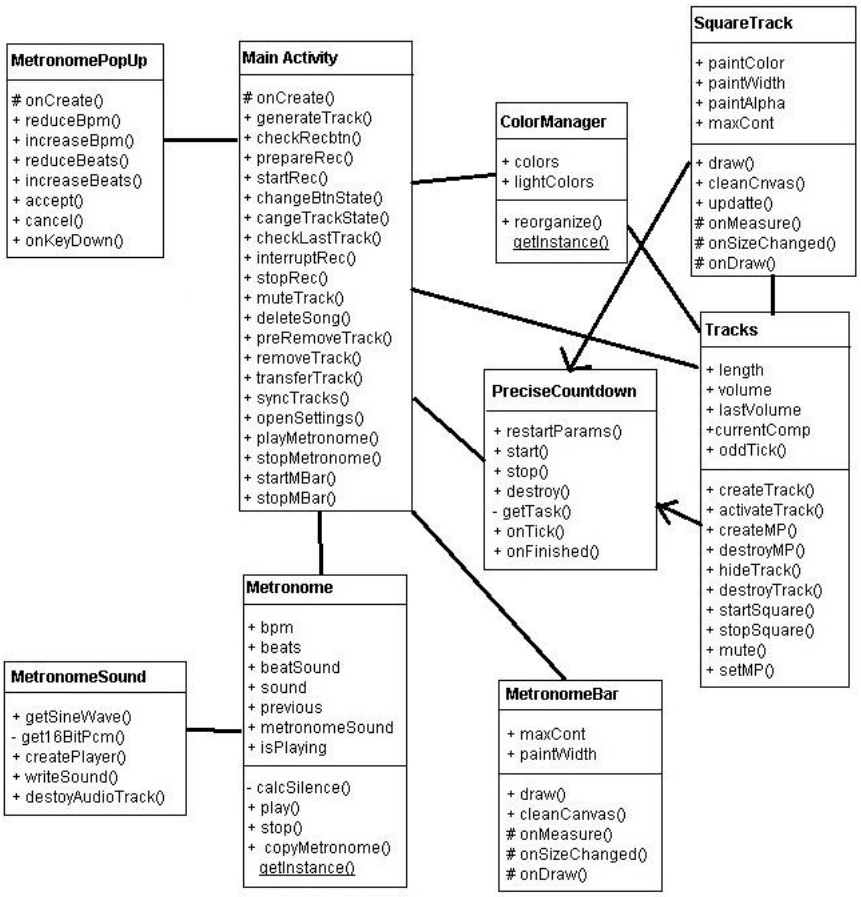


Diagrama17: Diagrama de clases

Como se ve en el Diagrama 1, la clase*MainActivity()* será la que gestione las demás clases. Será la clase que maneje el metrónomo, tanto la parte sonora como visual, llevará el control de los tracks y se encargará de abrir el submenú de ajustes. Además, ejecutará el ciclo de vida de las grabaciones, apoyándose en la clase *PreciseCountdown()* para ser precisa con los tiempos. Esta clase *PreciseCountdown()* también es utilizada por *Tracks()* y *Metronome()* con el mismo fin.

A través de *MetronomeSound()*, *Metronome()* obtiene el sonido del metrónomo que utilizará para crear un metrónomo de precisión. Para rellenar su cuadrado correspondiente, *Tracks()* utilizará la clase *SquareTrack()* y, además, obtendrá su color correspondiente gracias a *ColorsManager()*. Esta clase también es llamada por *MainActivity()* para la correcta gestión de los tracks.

Gracias a este diagrama de clases se ha podido escribir el código de la aplicación con una idea clara del cometido de cada clase. Sin embargo, para poder comprender y diseñar ciertas partes como, por ejemplo, el ciclo de vida

de un audio, se han empleado diagramas de flujo, por lo que, antes de empezar a explicar el código, se van a dar unas ideas generales de qué es y para qué sirve un diagrama de flujo.

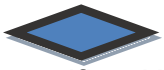
5.4.1. Diagrama de flujo

En primer lugar, un diagrama de flujo es una representación gráfica que describe un algoritmo informático, un proceso o un sistema. Gracias a un diagrama de flujo se puede encontrar solución a un problema de diseño, exponer una idea, representar un proceso, etc. En líneas generales, en un diagrama de flujo se representan los sucesos o acontecimientos mediante figuras geométricas que se relacionan entre sí por medio de flechas. Algunos de los símbolos más comunes en un diagrama de flujo son:


a el principio o final del sistema. Suele contener la palabra "Inicio" o "Fin" según corresponda.


cuando quiere indicarse un paso a seguir dentro de un proceso o un subproceso completo dentro de un proceso más grande.


Esta forma indica que es un documento impreso o informe.


Con esta figura se representa un punto de toma de decisión ramificación. Las líneas que surgen del rombo como diferentes acciones a tomar deben salir de puntos distintos del mismo.


Es un símbolo de entrada o salida, por lo tanto indica una información que entra o sale del sistema.


Representa un punto en el que el usuario debe introducir información al sistema de manera manual.

Este símbolo debe contener una letra, pues es un conector que indica que el flujo continúa en otro símbolo circular que contenga la misma letra.


Se utiliza este símbolo para representar la unión o convergencia de varias ramificaciones.


Uno o más subprocesos o listas se fusionan en uno nuevo.

 Indica que un conjunto de elementos de una lista se reorganiza según un criterio específico

 Este trapecio simboliza que un proceso va a repetirse en bucle hasta que sea detenido de forma manual.

 Con esta forma geométrica se establece el límite para que un bucle se detenga automáticamente.

 Indica un almacenamiento de datos.

 Se utiliza cuando hay un retraso en el proceso.

 Esta forma es más específica de diagramas de flujo orientados a programación, pues representa un bucle for.

Con estas características, un diagrama de flujo resulta una herramienta muy potente para diseñar y exponer un programa. Algunas ventajas que aporta un diagrama de flujo son:

- Visualizar una parte del código del programa de manera clara y sencilla.

- Definir la estructura del programa.

- Organizar el código.

- Diseñar la solución a un problema antes de empezar a escribir el código de la aplicación.

- Tener en cuenta el punto de vista del usuario y comprender cómo éste va a utilizar la aplicación.

Puesto que un diagrama de flujo puede abarcar gran cantidad de información, su diseño puede resultar abrumador. Para ello, es conveniente seguir una serie de pasos que faciliten la tarea de creación:

1- Tener claro el proceso o procesos que quiere representarse mediante el diagrama de flujo.

2- Ordenar los procesos y pasos por orden cronológico.

3- Identificar cada una de las tareas y acciones que se van a representar en el diagrama.

4- Una vez identificadas todas las tareas, relacionar cada una de ellas con su símbolo correspondiente.

Con esta información acerca de los diagramas de flujo se han diseñado algunos diagramas para resolver problemas y representar partes del código que se mostrarán más adelante para ilustrar la explicación del código.

6. Códigoen Android Studio

Apoyado sobre el diagrama de clase y los diagramas de flujo se ha diseñado en Android Studio el proyecto de desarrollo de la aplicación SquareLoops. Por tanto, a continuación se explicará de manera detallada cómo se ha producido el código y demás archivos relevantes del programa partiendo desde cero con el IDE Android Studio.

Android Studio es un programa completamente gratuito que se puede obtener de la página oficial de dicho programa, desde https://developer.android.com. Huelga decir que cuenta con soporte y actualizaciones también gratuitas.

Para crear un nuevo proyecto se selecciona la opción "Start a new Android Studio project". Tras esto hay que dar un nombre a la aplicación, el cual podrá ser modificado más tarde y será el nombre que aparezca en Play Store si se subiera a ella la aplicación. Debe añadirse una dirección web pues Android Studio combina esta URL a la inversa con el nombre del proyecto para crear el directorio del mismo. Esta URL no tiene por qué corresponder con un dominio existente, pero puede ser interesante para el desarrollo profesional de aplicaciones. Un punto muy importante de la configuración del proyecto es elegir la versión mínima que soportará la aplicación, así como tipo de dispositivo al que está dirigida. Este proyecto está enfocado a dispositivos móviles y la versión mínima es 8.0 Oreo con la API 26. La desventaja de utilizar una API baja es que no se podrán implementar ciertas funcionalidades pues no son soportadas por versiones de Android antiguas. Por último hay que elegir la activity inicial. El programa ofrece varias activities prediseñadas, pero en este caso se va a escoger una activity vacía para adaptarla más tarde desde cero a las necesidades de la aplicación.

Una vez finalizado el proceso se tiene la clase *MainActivity()* asociada con el archivo *activity_main.xml* que contendrá la información de los elementos que se verán en pantalla mientras esta activity esté activa.

A continuación, se pasa a explicar clase por clase cómo funciona cada una de ellas.

MetronomeSound

Esta clase se encarga de crear dos pitidos distintos, uno para indicar el principio del compás y otro más grave para el resto de beats. Dado que estos

sonidos deben reproducirse de manera perfectamente precisa, el objetivo será poder gestionarlos de manera que consuma los mínimos recursos posibles.

El gestor de audio más rápido es la clase *AudioTrack*. La función *createPlayer()* se encarga de ello. El constructor de *AudioTrack* recibe varios parámetros referidos al modo del sonido, el canal que usará... Hay que hacer un inciso especial para la variable *sampleRate* que se pasa como parámetro. Esta variable se inicializa en el constructor de la propia clase *MetronomeSound* y es un entero cuyo valor se recibe de la clase que hace una instancia de ésta. Mediante *sampleRate* se determina la frecuencia de la onda, pues, en definitiva, para crear un sonido debe crearse una onda. El tamaño del buffer también es igual a *sampleRate*, pues en él se almacenarán las amplitudes de la onda para crearla. Se pueden ver estos parámetros en la siguiente línea de código:

```
audioTrack = new AudioTrack(AudioManager.STREAM_MUSIC,
sampleRate, AudioFormat.CHANNEL_OUT_MONO,
AudioFormat.ENCODING_PCM_16BIT, sampleRate,
AudioTrack.MODE_STREAM);
```

Además, para preparar la variable *audioTrack*, que instancia la clase *AudioTrack*, se hace `audioTrack.play()`, de manera que se reduce el tiempo de inicialización cuando, más adelante, se quiera reproducir los sonidos.

La función*getSinWave()* crea la onda que será una nota específica cuyo tono dependerá del parámetro *frecuencyOfTone*. En esta función se "dibuja" la onda en un buffer que será devuelto mediante `return sample`. Al realizar el seno de un valor que aumenta según aumenta *bufferPos*, se crea una onda. Como se ve en la fórmula, cuanto mayor sea *frecuencyOfTone* más pequeña será la longitud de onda, dando como resultado una nota más aguda, pues a mayor frecuencia, mayor velocidad de movimiento de la onda y, por tanto, se produce un sonido más agudo.

```
for (int bufferPos = 0; bufferPos < samples; bufferPos++) {
    sample[bufferPos] = Math.sin(2 * Math.PI * bufferPos /
(sampleRate/frequencyOfTone));
}
```

Ahora bien, la clase *AudioTrack* reproducirá una serie de sonidos mediante el método *write()*. No obstante, necesita una señal digital y hasta el momento se

ha generado una onda, con valores numéricos por supuesto, pero a efectos de sonido no es una señal digital. Para convertir dicha señal en digital se usa el procedimiento de modulación llamado manipulación por impulsos codificados (PCM) a través de la función *get16BitPcm()*. En esta función, simplemente se otorga un valor numérico a cada fragmento de la onda que *AudioTrack* podrá entender como señal digital. Este valor depende de la amplitud de onda en dicho fragmento.

A través de la función *writeSound()* se reproducen los sonidos generados. Puesto que anteriormente se puso en marcha *audioTrack* mediante *audioTrack.play()*, al escribir en él un buffer con sonidos se empezarán a reproducir inmediatamente dichos sonidos. Estos sonidos son generados por el método *get16BitPcm()*.

Por último, para detener el sonido y liberar al gestor *AudioTrack*, se utiliza la función *destroyAudioTrack()*.

Metronome

La clase *Metronome* se encarga de gestionar el propio metrónomo de la aplicación, así como de coordinar su sonido. El constructor de esta clase simplemente se encarga de inicializar algunas propiedades cuya utilidad se verá más adelante y preparar la instancia de *MetronomeSound* llamada *mSound* mediante el método de esa clase *createPlayer()*.

La siguiente función es *calcSilence()* y se encarga de calcular cuántos segundos habrá de silencio en el compás. Más adelante se verá cómo encaja esta función en la construcción del metrónomo. La fórmula para calcular dicho silencio es muy sencilla: 60 segundos que dura un minuto entre el número de beats por minuto, bpm. Este tiempo es lo que dura un beat, es decir, la distancia desde que suena un beat hasta el instante anterior a que suene el siguiente. Dado que este intervalo se va a almacenar en un buffer de 8000 posiciones, o samples en este caso, hay que multiplicar por 8000, que es lo que va a ocupar. No obstante, como se quiere conocer el tiempo de silencio, hay que restar los samples que ocupa en el buffer el sonido del beat, que son 1000 samples.

Con *play()* se construye el metrónomo y se reproduce en bucle. La variable booleana *play* controla si el metrónomo está sonando, así que su valor se

cambia a *true*. Se calcula la duración del silencio con la función vista anteriormente y se obtienen de *mSound* dos sonidos gracias al método *getSinWave()*. El primer sonido, tock, será el sonido de todos los beats del compás, excepto el de inicio de compás, tick, que será más agudo para marcar dicho comienzo. Con un valor de *beatSound* igual a mil y de *beat* igual a diez mil, la diferencia de tono es suficientemente notoria.

La manera en la que funciona el metrónomo es la siguiente:

```
for(int instant=0;instant<sound.length&&play;instant++) {
if(t<this.tick) {
if(b == 0)
                sound[instant] = tick[t];
            else
sound[instant] = tock[t];
t++;
} else {
            sound[instant] = silence;
s++;
            if(s >= this.silence) {
                t = 0;
s = 0;
b++;
                if(b > (this.beats-1))
                    b = 0;
}
        }
    }
mSound.writeSound(sound);
} while(this.play);
```

En el buffer *sound* se almacenan los sonidos y silencios para construir un compás. En primer lugar, dado que es el primer beat del compás, se almacenan mil samples de tick, seguidos de los samples necesarios de silencio calculados mediante *calcSilence()*. Cuando*s*, que lleva la cuenta de lo que ocupa el silencio, sea mayor que *silence*, que contiene el resultado de *calcSilence()*, se volverá a empezar el proceso. En cambio, esta vez el beat ya no es el primero (*b* es mayor de cero), por lo que se usará el sonido de tock, procediendo de la misma manera. Este proceso se repetirá hasta llenar el buffer *sound*. Una vez listo el buffer, se escribirá su contenido en el gestor de audio *mSound* para ser reproducido de manera inmediata en la aplicación. Para que el metrónomo no se pare por sí solo, este proceso se repetirá dentro de un *while* dependiente de que el valor de *play* sea *true*.

Para encontrar la forma de hacer funcionar correctamente el metrónomo se ha usado el diagrama de flujo que se muestra a continuación:

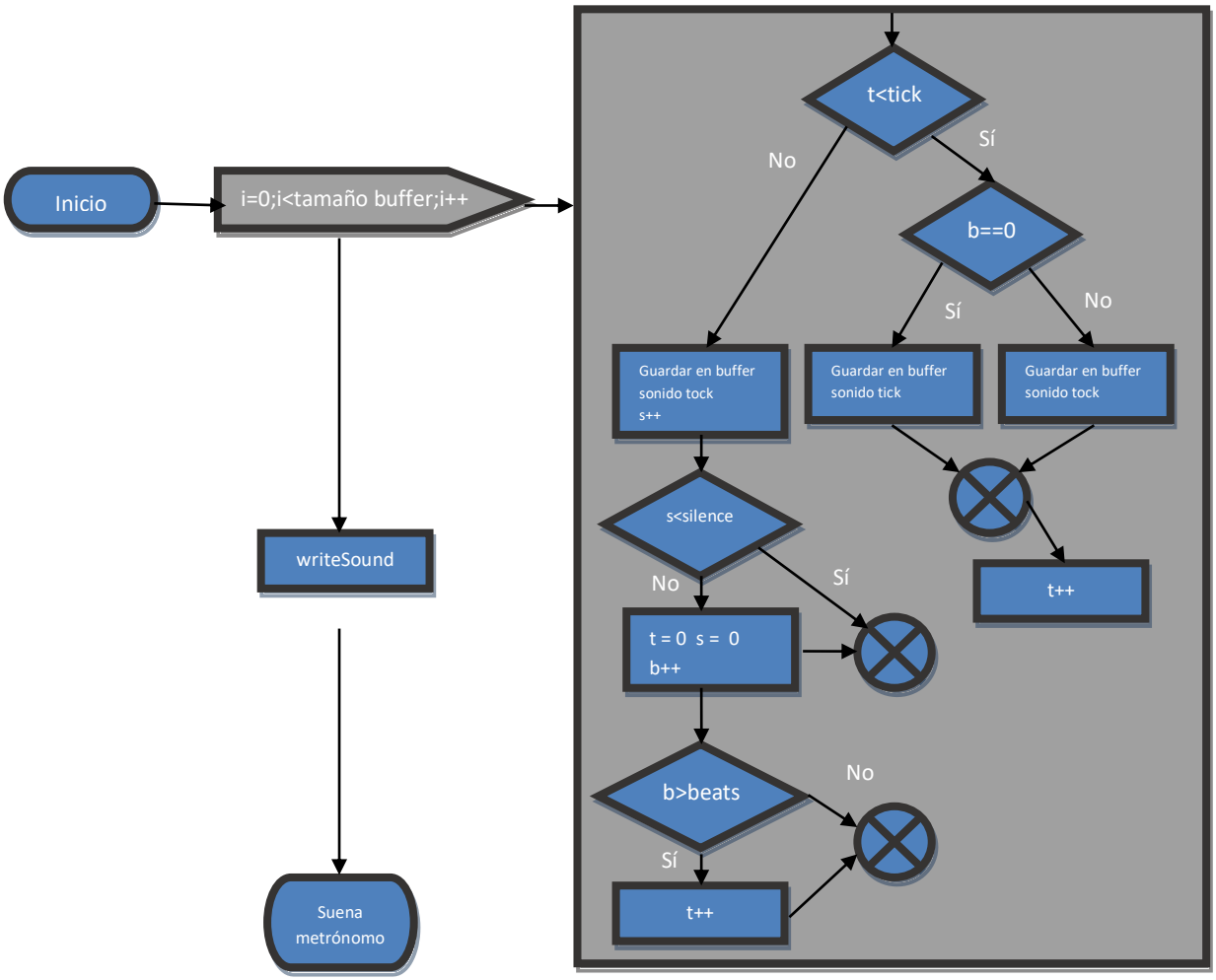


Diagrama 2: Metrónomo

El método *stop()* detiene el sonido del metrónomo. Al poner en *false* el valor de *play* se detiene el bucle for del método *play()*, con lo que no se generan más compases. Además, se detiene el AudioTrack *msound* para que no reproduzca los sonidos que hayan quedado almacenados en el buffer.

Se utiliza una serie de funciones tipo set/get para establecer y obtener los valores de los atributos *bpm*, *beats*, *beatSound*, *sound*, *previous*, *metronomeSound* e *isPlaying*.

La función, *copyMetronome()*, es necesaria para crear una instancia de *Metronome* con las mismas propiedades y que pueda ser modificada libremente desde otras clases.

Por último, hay que hacer global esta clase pues la misma instancia de *Metronome* va a ser utilizada por más de una clase. Para ello, hay que declarar una variable estática del mismo tipo que la clase, *Metronome* en este caso, que será llamada *instance*. A través de la función *getInstance()* otras clases podrán

acceder a la instancia global de esta clase. Esto ocurre gracias a las siguientes líneas de código:

```
public static synchronized Metronome getInstance(){
if(instance==null){
instance=new Metronome();
}
return instance;
}
```

PreciseCountdown

En esta clase se va a crear un contador descendente que se detendrá cuando se agote un tiempo establecido acorde a las preferencias de la clase que use este contador. Cada cierto tiempo puede repetir una acción, así como cuando llegue al final de la cuenta atrás. Es cierto que existe una clase *CountDownTimer*, pero es terriblemente impreciso al ejecutar una tarea cada cierto tiempo y no es útil para la precisión que exige esta aplicación.

El constructor inicializa tres variables:

- *totalTimer*: Tiempo total que durará la cuenta atrás.

- *interval*: Determina cada cuánto tiempo va a repetirse un proceso.

- *delay*: Si es necesario, puede iniciarse la cuenta atrás con un tiempo de retardo.

Todos estos tiempos estarán en milisegundos. También se inicializa *task* para cuando se necesite iniciar la cuenta atrás.

Para reiniciar la cuenta atrás con valores distintos se utiliza *restart()*, que recibe como parámetros tres enteros que serán los nuevos tiempos de *totalTime*,*interval* y *delay*.

Los métodos *start()* y *stop()* simplemente inician y detienen respectivamente a *task*, iniciando y deteniendo la cuenta atrás. Para detener el *PreciseCountdown activo* y liberar la memoria se utiliza *destroy()*.

Con la función *getTask()* se establecen los parámetros de la cuenta atrás para que cumpla con las necesidades requeridas y se crea la cuenta atrás precisa. Es importante saber que la clase *PreciseCountdown* extiende de *Timer*, con lo que se utilizan los métodos *onTick()* y *onFinished()*, que funcionarán igual que

en un *CountDownTimer* normal; *onTick()* realiza una tarea de manera periódica durante la cuenta atrás y *onFinished()* cuando ésta acaba, en ambos casos en un hilo paralelo al principal.

El código es el siguiente:

```
private TimerTask getTask(final long totalTime) {
return new TimerTask() {

@Override
public void run() {
long timeLeft;
            if (startTime <0 || restart) {
startTime = scheduledExecutionTime();
timeLeft = totalTime;
restart = false;
} else {
                timeLeft = totalTime - (scheduledExecutionTime() -
startTime);

                if (timeLeft <= 0) {
this.cancel();
startTime = -1;
onFinished();
                    return;
}
            }

            onTick(timeLeft);
}
    };
}
```

Como puede verse, se devolverá una tarea que se construirá a lo largo de esta función. Siempre que se llama a *getTask()*, la variable *startTimer* es igual a -1, con lo que siempre se pasa por el primer *if* en un inicio. Esto se hace para almacenar en *startTime* el momento en que se inició la tarea, necesario para calcular el tiempo transcurrido más adelante. En ese instante *timerLeft* es todo el tiempo que se pretende que dure la cuenta atrás. En cambio, los siguientes tics *timeLeft* debe ser el tiempo total, *totalTime* recibido como parámetro en la función, menos el tiempo que ya se ha consumido. Este tiempo se calcula como la diferencia entre la hora actual y la almacenada en *startTime*, momento en el que empezó la cuenta atrás. De esta manera se asegura que los ticks ocurran en los instantes precisos.

Para finalizar la cuenta atrás, cuando se haya agotado el tiempo, es decir, *leftTime* sea cero o negativo, se detendrá el timer de esta función, se restablecerá el valor de *startTime*a -1 para asegurar futuras llamadas a

getTask(), se llamará a *onFinished()* para realizar una última tarea y se devolverá esta task.

MetronomePopUp

Es la segunda activity de la aplicación. Al añadir una nueva activity, debe modificarse el archivo *AndroidManifest.xml*. En este archivo se han añadido las siguientes líneas de código para que funcione la nueva activity:

```
<activity android:name=".MetronomePopUp"
android:theme="@style/AppTheme.GauzyTheme"
></activity>
</application>
```

Para esta activity se ha creado un tema que hará que la vista se muestre en pantalla como un "pop-up", con lo que se establece este tema personalizado *GauzyTheme* como el tema de la activity.

Este nuevo tema se crea en el archivo *styles.xml*, en la carpeta, *values*. Aquí se establece que la pantalla fuera de la activity sea transparente, con lo que se verá la actividad principal de fondo por los bordes. Es intuitivo para el usuario que cuando éste pulse fuera del "pop-up" se cierra el menú, por lo tanto también se fija esta características. Por último, se determinan los colores de la activity, que serán los mismos que los de la activity principal, así como ajustarla a los márgenes para que quede centrada en pantalla. Todo esto se consigue incluyendo este código:

```
<style name="AppTheme.GauzyTheme" >
<item name="android:windowIsTranslucent">true</item>
<item name="android:windowCloseOnTouchOutside">true</item>
<item name ="android:alignmentMode">alignMargins</item>
<item name="colorPrimary">@color/colorPrimary</item>
<item name="colorPrimaryDark">@color/colorPrimaryDark</item>
<item name="colorAccent">@color/colorAccent</item>
</style>
```

En cuanto al layout de esta activity, *metronome_popup*, se incluyen todos los elementos que serán visibles en pantalla. En la parte superior del menú aparecerán las opciones a modo de lista. En cada línea, a la izquierda se encuentra un TextView que informa al usuario de qué ajustes se modifica ahí. A la derecha de este TextView se encuentra el o los modificadores necesarios

para ajustar esa opción. Algunos de ellos requieren que el propio usuario introduzca de manera manual mediante teclado un número o que lo aumente o disminuya gracias a unos botones alrededor del EditText que muestra dicho número. En cambio, otros ajustes sólo pueden estar activados o desactivados, con lo cual se utiliza un Switch. En la parte inferior del menú hay dos botones para aceptar o cancelar los ajustes modificados.

Para poder ordenar todos estos elementos se utiliza un LinearlLayout vertical que contiene varios LinearLayout horizontales. El LinearLayout correspondiente a los botones de aceptar y cancelar se mantiene en la parte inferior del menú gracias a la línea de código `android:layout_alignParentBottom = "true"`.

Se utiliza un sistema de pesos con el atributo *weight* para mantener una proporción de tamaño adecuada entre elementos. También se ajustan otros parámetros necesarios de los views como su color, los márgenes el tamaño respecto al espacio disponible en su layout, color del texto e id para poder hacer referencia a ellos desde la clase *MetronomePopUp*.

Dentro ya de la propia clase *MetronomePopUp*, se hace una instancia de *Metronome*, que es una clase global. La línea de código `Metronome metronome = Metronome.getInstance()` se encarga de ello.

En el constructor se establece el tamaño en pantalla de la activity, el último paso para terminar de convertir esta activity en un "pop-up". Para que el menú tenga un tamaño adecuado en los distintos dispositivos, se establece su longitud y ancho en base a un porcentaje del tamaño total de la pantalla.

A continuación, se necesita almacenar en variables la dirección de algunos de los elementos del layout para poder hacer referencia a estos más adelante. Se establece el número de bpms y beats recogiendo el número actual de estos almacenado en *Metronome*y se muestra en sus respectivos EditText. De igual manera, se establece el estado de los Switches correspondientes al compás de entrada a grabar y al sonido del metrónomo.

Si en el momento de abrir este menú existiese algún track y, por lo tanto, el metrónomo estuviese activo, los parámetros bpm y beats no podrían ser modificados. En las últimas líneas de código del constructor se crea esta limitación desactivando los botones y el EditText para que no puedan ser modificados por el usuario y crear una desincronización en futuras grabaciones. Además, se impide que el usuario pueda guardar el estado del EditText como vacío.

Las siguientes funciones se encargan de aumentar o disminuir en una unidad, según corresponda, el valor mostrado por el EditText al pulsar el usuario el botón de aumentar o disminuir ese parámetro. Se muestra a continuación el código de una de estas funciones como ejemplo:

```
public void reduceBpm (View v) { //Reduce los bpm en una unidad
aux = Integer.parseInt(bpm.getText().toString());
    if (aux >1) {
aux--;
bpm.setText("" + aux);
}
}
```

Para las demás funciones cambiará el parámetro a ajustar y si se lleva a cabo una suma o una resta. Es importante destacar que los valores están limitados entre 1 y 999.

El método *accept()* se encarga de cerrar la aplicación aplicando los cambios realizados. Para ello, simplemente toma los valores de los EditText así como el estado de los Switch y modifica los parámetros correspondientes del metrónomo. Con *finish()* se termina el ciclo de vida de la activity, cerrándola. Esta función se llamará al pulsar el botón accept. La llamada a esta función se establece desde el archivo *layout.xml*, con el atributo *onClick*.

Este atributo sirve también para que, al pulsar el botón cancel, se llame la función *cancel()* y se cierre la aplicación sin guardar ningún cambio.

Finalmente, debe tenerse en cuenta que, pulsar el botón de retroceso de los dispositivos con Android, debe ocurrir lo mismo que al pulsar el botón cancel. Para detectar la pulsación del botón de retroceso se implementa el método *onKeyDown()*.

MetronomeBar

Esta clase se encargará de dibujar una representación visual del metrónomo mediante una barra divida en compases que se rellena según avanza el metrónomo y al unísono de este. Transcurrido un compás se rellena por completo y se reinicia el proceso. Además, esta clase va a crear una view personalizada, con lo que parte del código estará orientado a que esta clase pueda utilizarse como view.

En primer lugar, para representar el metrónomo se van a dibujar los elementos necesarios en un canvas. Por tanto, en el constructor se inicializa *drawPaint* de tipo *Paint*, que será el "pincel" que dibuje sobre el "lienzo". Se establecen parámetros del pincel tales como color, transparencia o grosor inicial. Por supuesto, también se inicializa *canvasPaint* de tipo *Canvas* sobre el cual se va a dibujar. La variable drawX será la encargada de determinar la posición en la que el pincel va a dibujar, en un principio 0.

Para mover la línea que índica el avance del metrónomo se utiliza la función *draw()*. Para entender esta función hay que saber que se coloca el pincel englobando la posición que se le da a este, es decir, no dibuja a partir de la posición dada sino en torno a ella. Así, para que desde el primer momento dibuje con el grosor deseado y a partir de la posición dada por *drawX* se escribe `drawPaint.drawLine(drawX+(paintWidth/2), 0, drawX+(paintWidht/2, this.getHeight(), drawPaint))`. El primer y tercer parámetro se refieren a la posición inicial en el eje X, que es fija, y el segundo y cuarto parámetro al eje Y, que se recorre completo de arriba abajo para crear la línea.

En un primer momento, se establece el pincel en modo borrado para borrar la línea dibujada en el tick anterior, y luego se retoma el modo dibujar para pintar la nueva línea y, que de este modo, avance.

Una vez se ha llegado al final de la barra que representa el metrónomo hay que borrar lo dibujado otorgar a *drawX* la posición inicial para reiniciar el dibujado del metrónomo. En *cleanCanvas()* se lleva a cabo de la misma manera que en *draw()*, con la diferencia de que el grosor del pincel en modo borrado es igual a todo el ancho de la view para así borrar todo.

Para establecer el grosor de *paintWidth* se utiliza una función estilo set.

Es necesario insertar el método *onMeasure()* para poder dimensionar correctamente la view. Las variables locales *widthMode* y *heightMode* sirven como selectores para establecer la manera en la que otorgar un valor al ancho y la altura respectivamente. En este caso, se quiere que esta view tenga un valor de *match_parent* en ambos casos, así que las variables mencionadas serán igual a *MeasureSpec.AT_MOST*, que establece el *match_parent*.

Finalmente, es el método *setMeasureDimension()* el encargado de establecer el ancho y el alto. En este caso podrían fijarse directamente las dimensiones sin necesidad de utilizar los selectores, pero es conveniente mantener la estructura de *onMeasure()* para facilitar hipotéticos cambios futuros.

Los dos últimos métodos son necesarios para dibujar sobre el canvas.

El primero, *onSizeChanged()*, simplemente crea un mapa de bits e inicializa el canvas con este mapa de bits.

El segundo método es *onDraw()* y se ejecuta en cada tick de reloj de la aplicación. Su función es dibujar aquella que se indique en el propio método. En este caso, se dibujan varias líneas verticales a distancia equidistante, dividiendo la barra del compás en secciones, uno por cada beat que forma el compás. También se dibujan los márgenes superior, inferior, izquierdo y derecho. En definitiva, se dibuja el fondo sobre el cual se va a ir desplazando la barra del compás.

SquareTrack

Esta clase también crea una view personalizada que, a su vez, dibujará sobre un canvas y lo irá rellenando, por lo tanto será bastante parecida a *MetronomeBar* y algunas de sus funciones no se explicarán con tanto detalle dado que ya han sido explicadas.

De nuevo, hay una función *draw()* encargada de rellenar el cuadrado, con la diferencia de que, en este caso, no se elimina lo dibujado con anterioridad, dejándose un "rastro".

Cuando se rellena el cuadrado por completo, se usa la función *cleanCanvas()*, que borra todo lo dibujado y restablece la posición inicial de *drawX*.

En ciertos momentos, se requerirá variar la opacidad de lo dibujado en el cuadrado. Para ello es necesario redibujar con la nueva opacidad o "alpha" lo ya dibujado. La función *update()* se encarga de ello.

En primer lugar borra todo lo dibujado hasta ese momento, utilizando la misma manera deproceder que *cleanCanvas()*. Tras esto, simplemente se vuelve a dibujar hasta donde había llegado el pincel pero con el nuevo nivel de opacidad.

Las siguientes funciones son de estilo set/get para establecer u obtener el valor del color del pincel, *paintColor*, el grosor del pincel, *paintWidth* y el alpha del pincel, *paintAlpha*.

De nuevo es necesaria la aparición de *onMeasure()* que, en este caso, da un valor fijo en píxeles *squareSide*. A pesar de ser un valor en píxeles, está preparado para los distintos dispositivos pues toma su valor en base al ancho de la pantalla, siendo dicho valor la mitad de la pantalla menos la suma de los márgenes.

Los métodos *onSizeChanged()* y *onDraw()* funcionan tal y como se ha explicado en el apartado anterior. En este caso, *onDraw()* no dibuja nada.

ColorManager

Cada cuadrado que representa un track ha de tener un color distinto al resto. Dado que el color de los cuadrados puede variar de orden dependiendo de cómo se borran los tracks, es necesaria una clase que gestione dichos colores para facilitar la tarea de asignar nuevos colores a ciertos tracks.

Esta clase se basa simplemente en dos arrays en los que se almacenarán ocho colores distintos, uno por cada track. El primer array, *colors[]*, es para almacenar los colores estándar. El segundo, *lightColors[]*, almacenará colores más claros para crear un contraste entre el estándar y estos y dar la sensación de que el track parpadea cuando sea necesario.

En el constructor simplemente se almacenan los colores en sus posiciones correspondientes en cada array. Un color está formado por ocho dígitos en hexadecimal. Los dos primeros se corresponden con la opacidad del propio color, con lo que en ciertas ocasiones se utilizará un valor de 00 para hacer transparente cierta parte de un elemento si se precisa. Los demás valores corresponden con un sistema de colores RGB. Para encontrar los colores deseados se ha utilizado la página web Colors Hex, en la cual se puede buscar un color en una paleta y conocer su valor en varios sistemas distintos.

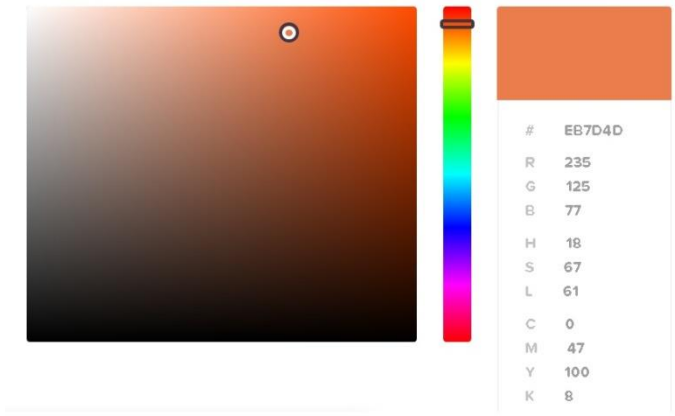


Figura187: Interfaz de htmlcoloreshex.com

Además, esta clase ha de ser global, pues la posición de algunos colores cambiará durante el uso de la aplicación y se debe mantener la coherencia en todas las clases que utilicen estos colores.

Para que otras clases puedan obtener el valor de los colores estándar y claros se utilizan unas funciones de estilo get que devuelven el valor solicitado.

Para finalizar, *reorganize()* se encarga de recolocar los colores. Esto ocurrirá cuando se elimine un track y se recoloquen los siguientes tracks al eliminado, con lo cual, el color asociado a una posición ahora se deberá asociar a una posición menos. Teniendo esto en cuenta, la función recibe dos parámetros *first* y *last*, que hacen referencia a la posición del track eliminado y la última posición en la que hay un track, respectivamente. El bucle for coloca los colores en una posición anterior. Para colocar el color correspondiente a la posición *first* y no perderlo es necesario almacenarlo en una variable auxiliar para, una vez finalizado el bucle, colocarlo en la última posición. Esta reorganización debe ocurrir tanto con los colores estándar como con los colores claros para mantener el mismo tono de color en la misma posición. A continuación se muestra esta parte del código para ilustrar el funcionamiento de *reorganize()*:

```
public void reorganize(int first, int last) {
auxColor = colors[first];
auxLightColor = lightColors[first];
    for (int colorPos = first; colorPos < last; colorPos++) {
colors[colorPos] = colors[colorPos+1];
```

```
lightColors[colorPos] = lightColors[colorPos];
}
colors[last] = auxColor;
lightColors[last] = auxLightColor;
}
```

Tracks

Esta clase gestiona los objetos tracks. En primer lugar, hay que saber que cada uno de estos tracks está asociado a un número para así poder hacer referencia al track requerido desde la clase que gestiona los tracks. Además, en la clase *Tracks* se crea visualmente el track, con sus botones y demás views correspondientes, así como gestionar las propiedades de los tracks.

Como constructor se tiene la función *createTrack()*, para poder instanciar un track sin crearlo. Dado que los tracks se crean de manera dinámica, pues no hay un número fijo de ellos, las views que conforman cada track deben colocarse desde el constructor y no desde XML.

Un track tiene varias apariencias que varían según el momento en que se encuentre ese track a las que llamaremos estados. Para clarificar este ciclo se ha diseñado el siguiente diagrama de estados:

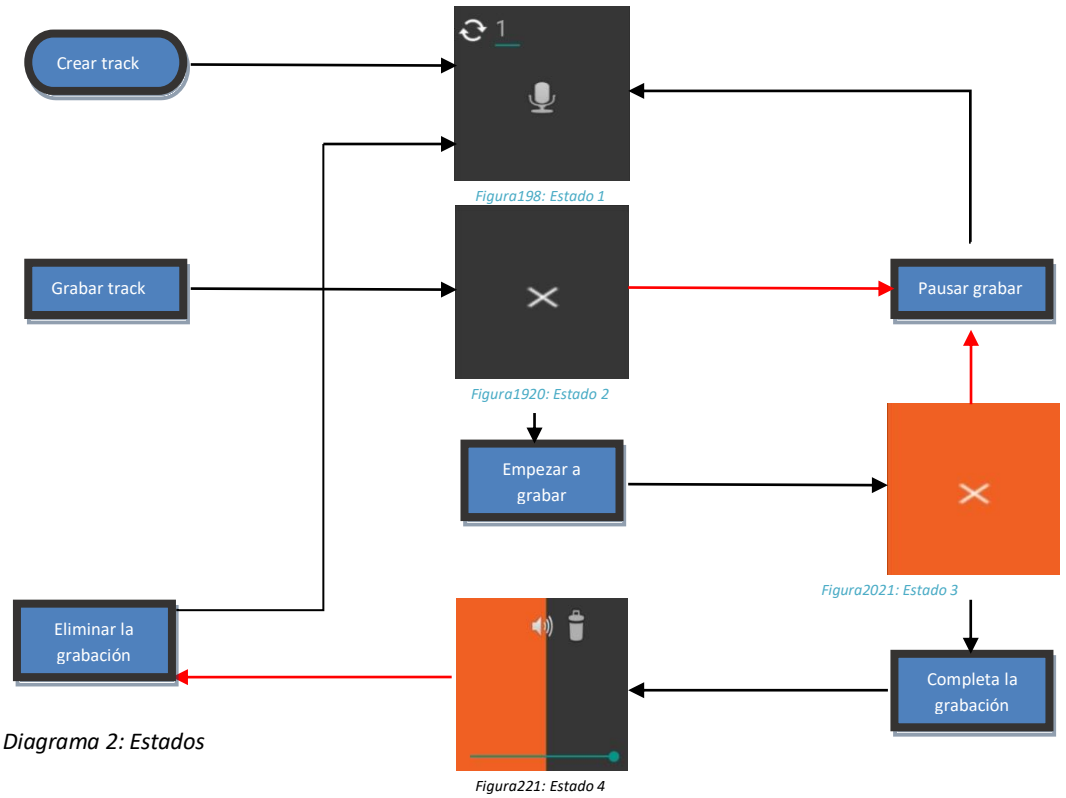


Diagrama 2: Estados

Las líneas de color rojo indican que sólo se puede pasar de ese estado a esa acción si el usuario pulsa un botón. Como puede verse en el diagrama, cuando se crea un track éste se muestra con un EditText para introducir en él el número de compases que se van a grabar en el track y un botón que inicia la grabación al ser pulsado. Una vez se pulsa este botón, si no se realiza ninguna acción el track se mostrará en espera mientras se reproduce el compás de entrada a grabar, seguido de un parpadeo que ocurrirá mientras dure toda la grabación y, finalmente, aparecerán nuevas views y se empezará a rellenar el cuadrado para indicar que se está reproduciendo el audio grabado. Si en algún momento se pausa la pre-grabación, la grabación o se elimina el audio, el track volverá a su estado inicial. Durante la grabación y pre-grabación desaparece el EditText pues no puede modificarse ya el número de compases, y el botón que antes iniciaba la grabación ahora la interrumpe. Mientras se reproduce el audio se muestran dos botones para silenciar el audio o eliminarlo y una SeekBar para regular el volumen. Además, hay que tener en cuenta que los track se dividen en dos tipos: iniciales y añadidos. Hay cuatro tracks iniciales que se crean automáticamente al iniciar la aplicación y son siempre visibles y hasta cuatros tracks más añadidos que pueden ser creados uno por uno por el usuario pulsando el botón de la esquina superior izquierda. Lo relevante de estos tracks añadidos respecto a las views es que siempre tendrán un botón extra en su esquina superior derecha para eliminar completamente el track, haciéndolo desaparecer de la pantalla en cualquiera de los cuatro estados vistos.

Pues bien, en el constructor van a colocarse todos estos elementos, aunque algunos de ellos serán invisibles, pues no deben mostrarse todos al mismo tiempo. Además, para organizar las posiciones de las views se utilizan distintos layouts y un sistema de pesos. Para poder introducir los valores de las dimensiones y el peso se utiliza la clase *TableLayout* para definir los parámetros necesarios gracias a su función *layoutParams()* y, más tarde, establecer las dimensiones y el peso de la view con la función propia de las views *setLayoutParams()*. La función *setMargins()* de *TableLayout* permite introducir el valor de los márgenes. Todas las views de cada track se crean desde esta función *createTrack()*, en cambio, existen cuatro LinearLayouts, uno por cada fila, que se crean en el código XML de la activity en la que se muestran los tracks. En cada una de estas LinearLayout horizontales se introducen dos tracks. Dentro de estas LinearLayouts se organizan los views de un track de la siguiente manera:

- Un ConstraintLayout que englobará todos los demás views del track para permitir que se solapen, pues deben ir encima del cuadrado que se rellena. Esta ConstraintLayout contiene de manera directa dos views:

  - Un *SquareTrack* para representar el avance del audio.

- Una LinearLayout vertical *lyt1*. En esta LinearLayout se colocan los siguientes elementos uno debajo del otro:

- Un ToggleButton que alterna entre grabar o abortar la grabación.

- Un SeekBar para regular el volumen.

- Otra LinearLayout *lyt2* horizontal que contiene varios views. Esta LinearLayout y sus views se encuentran en la parte superior de *lyt1* pero en esta explicación se han mencionado los últimos por conveniencia a la hora de exponer los elementos. Las views de *lyt2* son:

- Una ImageView para indicar qué hace el siguiente EditText.

- Un EditText correspondiente al número de compases.

- Un ToggleButton para silenciar o quitar el silencio del track.

- Un Button para eliminar el audio grabado.

- Otro Button para eliminar por completo el track.

Hay que tener en cuenta un problema que ocurre con las dimensiones de los elementos y es que, aunque se especifique una medida como *squareSide* y se otorgue un peso, si no hay otros elementos que limiten al view, éste tiende a expandirse llenando por completo el layout. Por este motivo se crean todos los views al mismo tiempo y siempre se mantienen dos tracks en la misma línea. Debido a esto, cuando se crea un track se añaden de dos en dos. Más adelante en la clase *MainActivity*se verá cómo se gestiona esto, pero por el momento hay que saber es por este motivo por el que, al crear los tracks impares cuyo id es mayor de 3 (es añadido) la visibilidad de sus views se establece en *INVISIBLE*. Por el contrario, *removeBtn*, que es el botón encargado de eliminar el track, será invisible en los tracks menores de 3 dado que estos tracks no pueden borrarse. A continuación se muestra como ejemplo la manera de implementar el Button *recBtn*, pues todos los views se agregan prácticamente de igual manera:

```
recBtn = new ToggleButton(ctx);
recBtn.setLayoutParams(recParams);
recBtn.setBackgroundResource(R.drawable.ic_voice_search);
recBtn.setTextOn("");
recBtn.setTextOff("");
recBtn.setTextColor(0xFFAAA9A9);
recBtn.setChecked(false);
recBtn.setTag(pos);
if (pos%2 != 0 &&pos >3) recBtn.setVisibility(View.INVISIBLE);
lyt1.addView(recBtn);
```

Cabe destacar que el EditText *compass*, encargado de especificar el número de compases que se van a grabar, se apoya en la variable entera *compassNum*. En esta variable se almacena el contenido de *compass* cuando el usuario termina de editar el EditText. Así se evita que, si en algún momento el usuario borra todo el contenido de *compass*, alguna función acceda a un valor vacío y se produzca un error. Si el usuario intenta dejar un valor vacío en *compass*, se restablece el último valor de *compassNum*.

Por otro lado, el SeekBar *volumeBar* utiliza la función *onProgressChanged* para regular el volumen del audio grabado, reproducido por un MediaPlayer, con respecto al nivel de progreso de la propia barra.

```
public void onProgressChanged(SeekBar seekBar, int progress, boolean
fromUser) {
mediaPlayer.setVolume((float) progress/100, (float) progress/100);
square.setPaintAlpha((progress*2)+ 55);
volume = (float)volumeBar.getProgress()/100;
square.update(square.getPaintAlpha());
}
```

El parámetro de entrada*progress* es el porcentaje al cual se encuentra la barra del SeekBar, con lo cual éste será el volumen. Para variar la opacidad del cuadrado se empieza desde 55 pues no se quiere hacer completamente transparente nunca aunque el volumen esté en cero. Como el valor máximo de alpha es 255, cada unidad de porcentaje en la barra de *volumeBar* equivaldrá a dos unidades para alpha.Por último, la variable *volume* almacena el valor del volumen actual para futuros usos y se llama a *update* para actualizar el alpha del cuadrado.

La función *activateTrack()* forma parte de la gestión de los tracks añadidos. En este caso, hace visible el track en su estado inicial, preparado para ser utilizado por el usuario.

Para gestionar la clase *MediPlayer* encargada de reproducir el audio grabado se usan las clases *createMP()*y *destroyMP()*. La primera simplemente recoge el archivo de audio resultante de la grabación y lo carga en *mediaPlayer*. También activa la opción de reproducir en bucle el contenido de *mediaPlayer*. Por el contrario, *destroyMP()* elimina el archivo de audio. No obstante, su eliminación debe de ir precedida de una serie de acciones tales como detener la reproducción del archivo y liberar a *mediaPlayer*. La variable *currentComp* lleva

la cuenta del compás en el que se encuentra la grabación. Dado que se elimina dicha grabación, se restaura su valor a cero.

A la hora de eliminar un track añadido se debe tener en cuenta si se encuentra a la izquierda (su id *pos* será par) o la derecha (*pos* será impar). Esto es importante porque cuando se elimina un track par en realidad no se elimina sino que se oculta para mantener las dimensiones correctas del otro track, tal y como se ha explicado antes. Para ocultar el track se utiliza la función *hideTrack()*, que oculta todas las views del track. En cambio, si se requiere eliminar completamente el track se utiliza *destroyTrack()*, que estable la propiedad *visibility* de las views en *GONE* para eliminarlas. En ambas funciones se llama a *destroyMP()* en base a si el propio *mediaPlayer* que va a ser eliminado contiene algo y en base a si se recibe como parámetro de entrada en la función un *false*. Esto es así porque en ocasiones se requerirá eliminar también el archivo de audio grabado y en cambio, en otras ocasiones será necesario guardarlo.

Para controlar el pintado del cuadrado se tienen las funciones *startSquare()* y *stopSquare()*. Laprimera se apoya en la clase *PreciseCountdown* para ejecutar de manera precisa y periódica el pintado de una línea en el cuadrado para rellenarlo de forma constante. El tiempo total será el mismo tiempo que tarde en reproducirse toda la grabación correspondiente al track, mientras que los intervalos serán de cinco milisegundos. Para pintar el número de veces exacto y rellenar de manera sincronizada el cuadrado, se debe calcular el grosor del pincel que pintará dichas líneas.

```
length =
metronome.getBeats()*(Integer.parseInt(compass.getText().toString()))*
        (int)(60000/metronome.getBpm());
int maxCont =  (int)Math.floor((float)length/ 5); // Veces que hay que
pintar para llenar el cuadrado completamente
square.setPaintWidth((float)squareSide / maxCont);
//Grosor de cada franja = Tamaño del square / (Duración del track /
Periodo del thread)
```

En estas líneas de código se muestra, en primer lugar, la fórmula para saber cuánto tiempo ha de durar el pintado del cuadrado. Se almacena en la variable *length* y es la multiplicación del número de beats en un compás, por el número de compases grabados por el tiempo que transcurre entre cada beat. Para el grosor de *paint* primero se calcula cuántas líneas habrá que dibujar, *maxCont*. Esto es sencillo, será el resultado de dividir el tiempo total entre lo que dura un

intervalo, cinco milisegundos. Se redondea a la baja porque con un tick más se produciría un retardo. En este caso el número de ticks no variará por el cálculo de *maxCont*, pero sí el grosor de *paint*, con lo cual se rellenaría el cuadrado antes de tiempo. Finalmente, el grosor de *paint* será el resultado de dividir el ancho total del cuadrado, *squareSide*, entre el número de líneas a dibujar, *maxCont*. Cuando finaliza la cuenta atrás se limpia lo dibujado, dejando el cuadrado preparado para volver a empezar a pintarlo.

Para detener el dibujado se utiliza la función *stopSquare()*, que simplemente consiste en detener el *PreciseCountdown*, destruirlo y borrar lo que se haya dibujado. Se hace en un hilo aparte dado que esto es necesario para dibujar o, en este caso borrar, sobre un canvas.

La función *mute()* silencia o devuelve el volumen a la pista de sonido. La acción que lleve a cabo dependerá del estado del ToggleButton *muteBtn*. Se utilizan las variables *volume* y *lastVolume*para poder reorganizar los tracks cuando uno de ellos se borra sin perder el estado del volumen de la pista que se recoloca. Además, *lastVolume* también sirve para recuperar el volumen que tenía el audio antes de silenciarlo con *mute*, por ejemplo, si se encontraba al ochenta por ciento y se desactiva su volumen, al restaurarlo debe volver a tener un ochenta por ciento de volumen y no el cien por cien. Para mantener la coherencia en la interfaz también se modifica la posición de *volumeBar*. Como cada vez que se modifica el nivel de *volumeBar* se almacena en *volume* ese valor, es por esto por lo que se usa *lastVolume* para almacenar el anterior volumen.

Por último, se puede acceder a algunas views desde otra clase a través de unas funciones get que devuelven una instancia de la view correspondiente. Estas views son *recBtn*, *muteBtn*, *compass*, *deleteBtn* y *removeBtn*. También existe una serie de funciones set/get para acceder a las propiedades de la clase *length*, *volume*, *lastVolume*, *currentComp* y *oddTick*, así como para *mediaPlayer*.

MainActivity

Esta es la activity inicial y su interfaz se gestiona desde *activity_main.xml*. En *MainActivity* se gestiona el ciclo de vida de los tracks, se añaden nuevos tracks, se accede al menú de ajustes y se gestiona el metrónomo.

En primer lugar se va a explicar el archivo XML. En la parte superior de la pantalla, de izquierda a derecha, hay un Button para añadir tracks, un *MetronomeBar* para indicar el progreso del metrónomo y otro Button para desplegar el menú de opciones, llamando a la activity de *MetronomePopUp*. Debajo de esta fila hay cuatro LinearLayouts, uno debajo del otro, para incluir tracks de dos en dos. Estos LinearLayouts se encuentran dentro de un ScrollView, ya que no cabrían todos los tracks en la pantalla de un dispositivo. Gracias al ScrollView se puede deslizar la pantalla arriba y abajo para acceder a todos los tracks.

Respecto al archivo .java, el constructor se va a encargar de crear los cuatro tracks iniciales que van a estar siempre visibles, inicializar algunas variables y hacer referencia en otras a las views de *activity_main*.

Para gestionar los tracks, estos van a ser almacenados en un array de tracks llamado *tracks[]*, desde la posición 0 a la 3, en este caso. Se utiliza un for para crear los cuatro tracks, pues el proceso es el mismo para todos. Como se colocan en distintos LinearLayouts, se utiliza un Switch para distribuirlos correctamente. Para conseguir que los botones de cada track llamen a una función de *MainActivity* se utilizan las siguientes líneas de código:

```
auxBtn = tracks[posTrack].getRecBtn();
auxBtn.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
        checkRecBtn((Integer) v.getTag());
}
});
```

En este caso, cuando recBtn sea pulsado se llamará a la función *checkRecBtn()*. Para el resto de botones se procede de igual manera con su función correspondiente.

La variable *activatedTracks* es de vital importancia para la gestión de tracks pues cuenta el número de tracks activos. Se utiliza esta variable y no toma la cantidad de elementos del array porque no siempre va a coincidir el número de tracks activos, los que el usuario puede usar, con los que están creados en el array *tracks[]*.

Cuando se pulse el botón de añadir un nuevo track se llamará a la función *generateTrack()*. Esta función procede de manera muy parecida al constructor

a la hora de crear un nuevo track, colocándolo en su LinearLayout correspondiente mediante un Switch y asignando funciones a los views tal y como se ha visto. La diferencia es que, en este caso, los tracks se crean de dos en dos para evitar la deformación de las dimensiones de los tracks, tal y como se explicó anteriormente. Puesto que el usuario sólo va a añadir los tracks de uno en uno, en un principio se añaden dos tracks pero el último se oculta en pantalla. Es en este caso cuando hay un elemento más en *tracks[]* que el valor de *activatedTracks*. Cuando se añade un track más, se hace visible el track oculto gracias al método *activateTrack()* de la clase *Tracks*. Gracias a la variable *activatedTracks* se puede hacer referencia la posición nueva en la que se va a colocar el track dentro de *tracks[]*. Cuando ya existan ocho tracks, *activatedTracks* es ocho, se deshabilita el botón de añadir tracks con la línea de código `if(activatedTracks == 8) generateBtn.setEnbaled(false)`.

La función *checkRecBtn()* es la función que se llama al pulsar el botón *recBtn* de la clase *Tracks*. Como este botón es un ToggleButton, se usa *isChecked()* para conocer en qué estado se encuentra el botón y realizar una acción u otra dependiendo de esto. Este botón puede comenzar la grabación o interrumpirla de manera manual. Para entender mejor el proceso de grabación hasta que se obtiene el archivo de audio, todo ello provocado simplemente por pulsar el botón *recBtn*, se ha diseñado el siguiente diagrama de flujo:

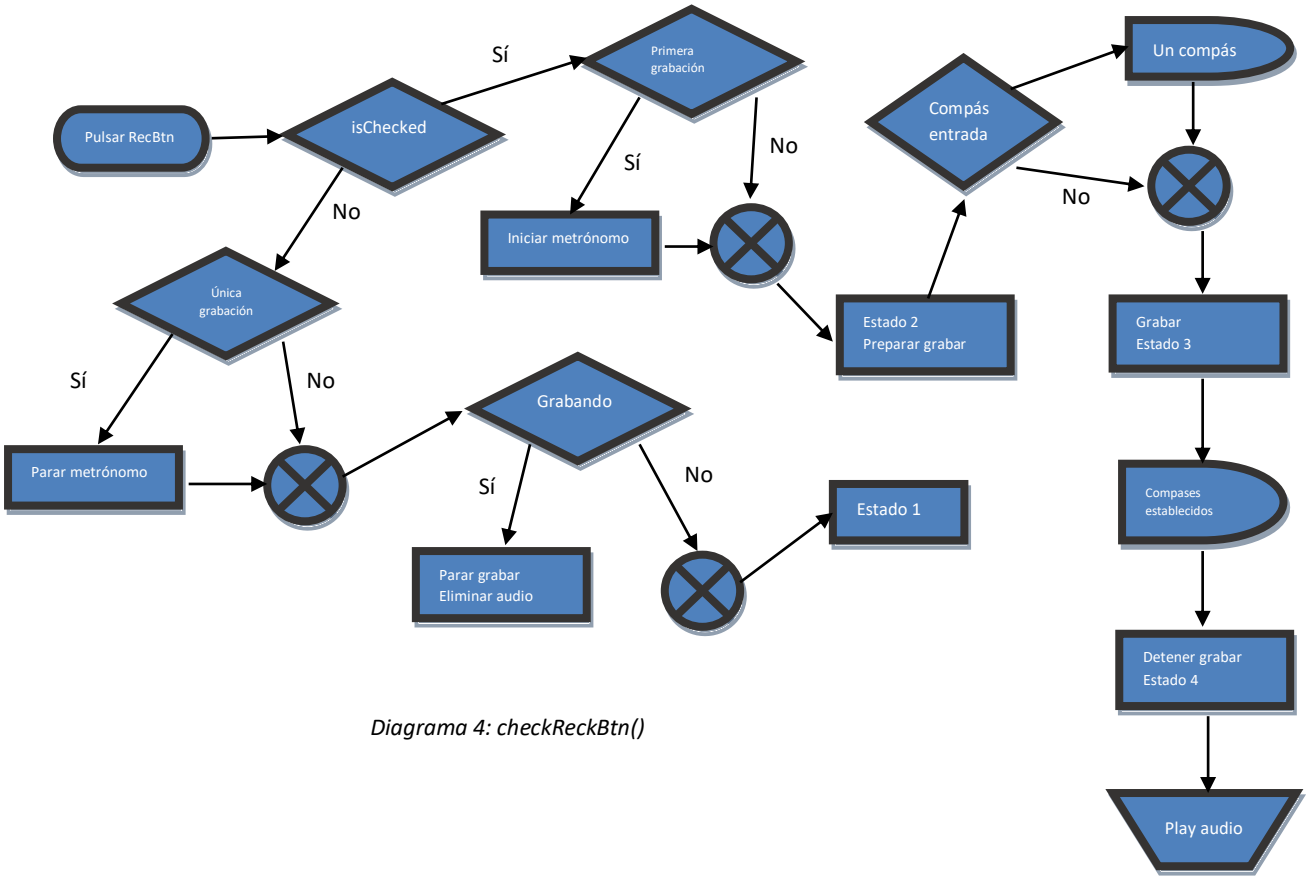


Diagrama 4: checkReckBtn()

Los estados son las distintas apariencias por las que pasa un track, expuestos en el diagrama de estados incluido en la clase *Tracks*. Como se puede ver en este diagrama, es importante controlar si el track que se graba o se elimina es el único activo, pues habrá que iniciar el metrónomo o detenerlo.

En el caso de que al pulsar el ToggleButton se entre en modo grabar, hay que llevar el track a su estado de apariencia 2, pero también se modifica el aspecto de los demás tracks y botones de añadir track y abrir el menú, deshabilitándolos temporalmente. Esto se hace para que no ocurran otros procesos que puedan interferir de manera nociva en la grabación. Sin embargo, no es en esta función donde se empieza a grabar, sino que se llama a *prepareReck()* para preparar la grabación, para ahorrar tiempo y sincronizar de manera perfecta las pistas cuando se empiece a grabar. Es la función *playMetronome()* la encargada de comenzar a grabar. Esto es así para poder coordinar el comienzo de la grabación con el comienzo de compás. Puesto que, una vez iniciada la función *playMetronome()* ésta se ejecuta en bucle, la variable *soundMetronome* con un valor *true* es la encargada de indicar a *playMetronome()* que empiece a grabar cuando empiece el siguiente compás o tras un compás de entrada. También es importante la variable *compassCont*. Esta variable también es utilizada por *playMetronome()*, pero es en *checkReckBtn()* donde se le otorga el valor correspondiente. Si esta variable vale -1, esto quiere decir que la futura grabación aún no ha entrado en el compás de entrada, 0 es dicho compás de entrada que utiliza el músico para "coger" el ritmo y 1 indica el compás en el que se empieza a grabar.

Si la función *checkRecBtn()* es llamada por *recBtn* pero con *isChecked()* devolviendo false, quiere decir que se va a abortar la grabación. Lo único que se hace aquí es cambiar el estado del ToggleBtn y llamar a la función *interruptRec()*, así que vamos a ver qué hace esta nueva función.

Lo primero que se hace en *interruptRec()* es detener el sonido del metrónomo, que no el metrónomo, si acaso estuviese sonando. Como el metrónomo sólo suena cuando se está grabando y sólo puede grabarse un track a la vez, no entra en conflicto con otros tracks. Como se ve en el diagrama, se debe retomar el estado 1 del track, cambiando la apariencia de sus elementos, pero también la de los demás elementos en la interfaz, los cuales estaban deshabilitados mientras se producía la grabación. En las siguientes líneas del código se detiene la grabación si ésta ya hubiese iniciado:

```
if (isRecording) records[pos].stop();
records[pos].release();
records[pos] = null;
if (isRecording) {
isRecording = false;
song = new File(Environment.getExternalStorageDirectory()
            + "/audio" + pos + ".3gp");
song.delete();
song = null;
cdRec.stop();
cdRec = null;
}
```

Como puede verse, se realiza dos veces un if comprobando el estado de la variable *isRecording*. Esto debe ser así porque no puede eliminarse el audio hasta que se detenga la grabación. Sin embargo, anteriormente se hizo un *prepare()* de el *MediaRecord* encargado de la grabación, con lo cual habrá que liberarlo siempre, incluso aunque no hubiese empezado a grabar. Después de estas líneas sólo queda comprobar si este track era el único con un audio. Si esto es así, se detiene el metrónomo.

Este era el caso de interrumpir la grabación, pero se había visto que nada más pulsar el ToggleButton con intención de grabar se llamaba a la función *prepareRec()*, que se encargará de realizar todas las tareas previas a una grabación mediante un MediaRecorder para agilizar el proceso. Lo único relevante de esta función es saber si existirá un MediaRecorder para cada track, y que se almacenarán en el array *records[]*.

Antes de explicar las funciones de gestión de la grabación va a exponerse el funcionamiento del metrónomo interno que rige todas estas funciones desde *playMetronome()*. Como se ha visto, esta función sólo es llamada cuando se inicia la grabación de una pista sin que hubiera ninguna más funcionando. Por lo tanto, sólo puede haber un metrónomo activo al mismo tiempo. Este metrónomo sirve para asegurar que las acciones relativas a las grabaciones solamente ocurren al comienzo de un compás, asegurando la coherencia del tempo.

El bucle del metrónomo está regido por un *PreciseCountDown* que se ejecuta cada compás. Este tiempo se calcula de la siguiente manera: `timeBeat = (int)(60000/metronome.getBpm())`. Básicamente es el resultado de dividir los milisegundos de un segundo, 60000, entre el número de beats que ocurren en un minuto, bpm, y es el tiempo entre beats. Ahora, al multiplicar *timeBeat* por el número de beats se obtiene el tiempo que dura un compás. De esta manera,

las acciones de las siguientes líneas de código se ejecutarán solamente al comienzo de cada compás si procede:

```
if (remove) removeTrack(trackToRemove);
if (soundMetronome) {
if (mThread == null && (metronome.getMetronomeSound() ||
metronome.getPrevious())) {
copyMetronome = metronome.copyMetronome();
mThread = new mThread();
mThread.start();
}
if (compassCont == 1) {
isRecording = true;
isPlaying = true;
metronome.setIsPLaying(true);
        if (mThread != null &&metronome.getPrevious() &&
                !metronome.getMetronomeSound()) {
            stopMetronome();
soundMetronome = true;
}
        startRec(activity,
        posRecording);tracks[posRecording].startSquare(activity);
}
}
syncTracks();
```

En primer lugar, si un track está listo para ser eliminado, se procede a su eliminación. Seguidamente, si un track está solicitando hacer sonar el metrónomo, esto no está ocurriendo y la opción del sonido del metrónomo esta activa, se hará sonar el metrónomo gracias a la clase *Metronome*. El thread *mThread* es el hilo encargado de ejecutar paralelamente el sonido del metrónomo, por lo tanto sirve para comprobar si éste está activo. Tras esto, comienza una grabación si es necesario. La manera de comprobarlo es preguntando el valor de *compassCont*. Si dicho valor es 1, tal y como se ha visto antes, un track necesita empezar a grabar en este compás. En caso de que la opción del sonido del metrónomo esté desactivada, es aquí donde se desactiva el sonido para no interferir en la grabación. Los flags *isRecording* e *isPlaying* determinan si algún track se está grabando o está siendo reproducido. Por último, la función *syncTracks()*ejecuta otra serie de tareas que pueden desfasarse poco a poco unos milisegundos unas de otras y necesitan ser reiniciadas a comienzo de compás para igualar sus tiempos de nuevo.

En el método *onFinish()* del *PreciseCountDown* se vuelve a llamar al propio contador para ejecutarlo en bucle y se suma uno en *compassCont*.

Las acciones que se sincronizan en la función *syncTracks()* son:

- Resetear la barra del metrónomo. Se detiene, limpia lo dibujado y se empieza a dibujar desde cero la barra del compás.

- Actualizar el contador de compases propio de cada track. Este contador sirve para poder grabar más de un compás en un track si es necesario sin que se detenga automáticamente antes de tiempo.

- Reproducir el audio de cada track, si lo hubiese, desde cero.

- Detener el pintado del cuadrado si no se ha detenido automáticamente, borrar lo pintado y empezar a pintarlo desde el principio, si hubiese una pista de audio reproduciéndose en ese track.

Esta función es necesaria pues Java no es un lenguaje especialmente preciso en materia de tiempos y, en ocasiones, aparecen retardos al ejecutar algunas acciones.

Continuando con el orden en que aparecen las funciones en el código, la siguiente función a *prepareTrack()* es *startRec()*. Esta función utiliza un *PreciseCountdown* para ejecutar la grabación en un hilo paralelo. Es útil un contador porque, en cada beat, se alterna el color del track entre color estándar y color claro para dar la sensación de parpadeo y denotar que se está grabando. Se utiliza la propiedad de la clase *TracksoddTick* para alternar entre los colores. El tiempo de la cuenta atrás es el tiempo que dura un compás por el número de compases específico para ese track, ejecutándose un tick cada beat, tal y como se ha dicho.

El método *onFinish()* detendrá la grabación y devuelve el color genérico al fondo del cuadrado.

La función *changeBtnsState()* recibe un parámetro booleano *state*. Este parámetro se utilizará para establecer el estado de todos los elementos de la interfaz excepto de los propios del track que ha llamado a *changeBtnsState()*. Si *state* es *false*, se desactiva la posibilidad de interactuar con esos elementos, y si es *true* se vuelven a activar. Además, cambia el color de los elementos deshabilitados a gris claro para hacer notar que no pueden utilizarse. Esta función es útil para evitar que se ejecuten otras acciones mientras se lleva a cabo una tarea que no permite que ocurran otras al mismo tiempo.

Por el contrario, para deshabilitar o habilitar las funciones de un track específico se utiliza la función *changeTrackState*, que funciona de la misma manera que *changeBtnsState* pero para un track concreto únicamente.

Como se ha visto, es necesario comprobar si un track es el único funcionando en varias ocasiones. Se encarga de ello *checkLastTrack()*, que modifica el estado de la variable *isPlaying* en función de si se cumple o no dicha premisa. Para ello, simplemente se comprueba el estado de todos los Mediarecord almacenados en *records[]*. Si alguno de ellos contiene una grabación, no es *null*, la variable *isPlaying* cambiará su valor a *true*, indicando que al menos un track tiene una grabación y, por lo tanto, está activo.

Tal y como se vio al explicar *startReck()*, se llama a una función *stopRec()* para detener la grabación. Esta función sólo se utiliza cuando la grabación se detiene de manera natural y automática, es decir, no ha sido el usuario quien ha decido abortarla, sino que se ha completado el número de compases requeridos en la grabación.

De nuevo, se tiene en cuenta si el metrónomo estaba sonando, en cuyo caso se detiene pues el metrónomo no suena mientras se reproducen las pistas, sólo mientras se graba o en la pre-grabación. Para finalizar la grabación simplemente se detiene el MediaRecord correspondiente y se libera. También se cambia el estado del track al estado 4.

Es esta misma función la encargada de comenzar la reproducción del archivo de audio nada más cortar la grabación, lo cual es la base de un looper. Para ello se utiliza la clase MediaPlayer y se crea un nuevo archivo de audio mediante la función *createMP()*, tal y como ya se vio en la explicación de su clase *Tracks*.

La función *muteTracks()* es muy sencilla, pues utiliza la función *mute()* de la clase *Tracks* para silenciar o reactivar el volumen de un track específico. Para evitar fallos en la ejecución de la aplicación, sólo se llamará a *mute()* en caso de que el track que lo solicite tenga un MediaPlayer con un archivo de audio.

Cuando sea necesario eliminar una grabación se llamará a *deleteSong()*. Esta función simplemente detiene y elimina el archivo de audio asociado al track que ha llamado a dicha función, devolviéndolo a su estado 1, pero no elimina el track en sí.

Para eliminar el archivo de audio se utiliza la función *destroyMP()* de la clase *Tracks* y para detener y vaciar el cuadrado se utiliza *stopSquare()* de la misma clase. Es importante vaciar la posición correspondiente de *records[]*. Al

devolver la apariencia del track al estado 1 también se restablecen los estados de sus elementos, como por ejemplo el nivel de *volumeBar* o el estado del ToggleButton *recBtn*.

De nuevo se comprueba si éste era el último track que estaba reproduciendo un audio mediante *checkLastTrack()*. Si es así, se detiene el metrónomo interno. La función *removeTrack()* se encarga de eliminar por completo un track y se verá más adelante. Sin embargo, desde *deleteSong()* se la llama en caso de que este fuese el último track de la siguiente manera `if(remove) removeTracks(pos)`. Esto es así porque, tal y como se ha visto, *removeTrack()* es una de las funciones que se ejecutan solamente a principio de compás. No obstante, si el metrónomo se elimina nunca llegará a eliminarse ese track que se solicitó destruir. Además, esto sólo ocurrirá si es el propio track removido el que elimina su archivo de audio y a la vez es el último en estar activo. Es por esto que, en este punto, *deleteSong()* debe llamar a *removeTrack()*.

Para eliminar un track de la interfaz se ven involucradas tres funciones: *preRemoveTrack()*, *removeTrack()* y *transferTrack()*. Para ello, cuando el usuario pulsa sobre el *removeBtn* de un track concreto, no es ese track el que se elimina, sino que en realidad se elimina el último y reordena los demás, aunque visualmente parecerá que se ha eliminado el track elegido y los siguientes a éste se han movido una posición hacia adelante. Como pueden darse muchos casos distintos al borrar un track y deben tenerse todos en cuenta, se ha diseñado el siguiente diagrama de flujo:

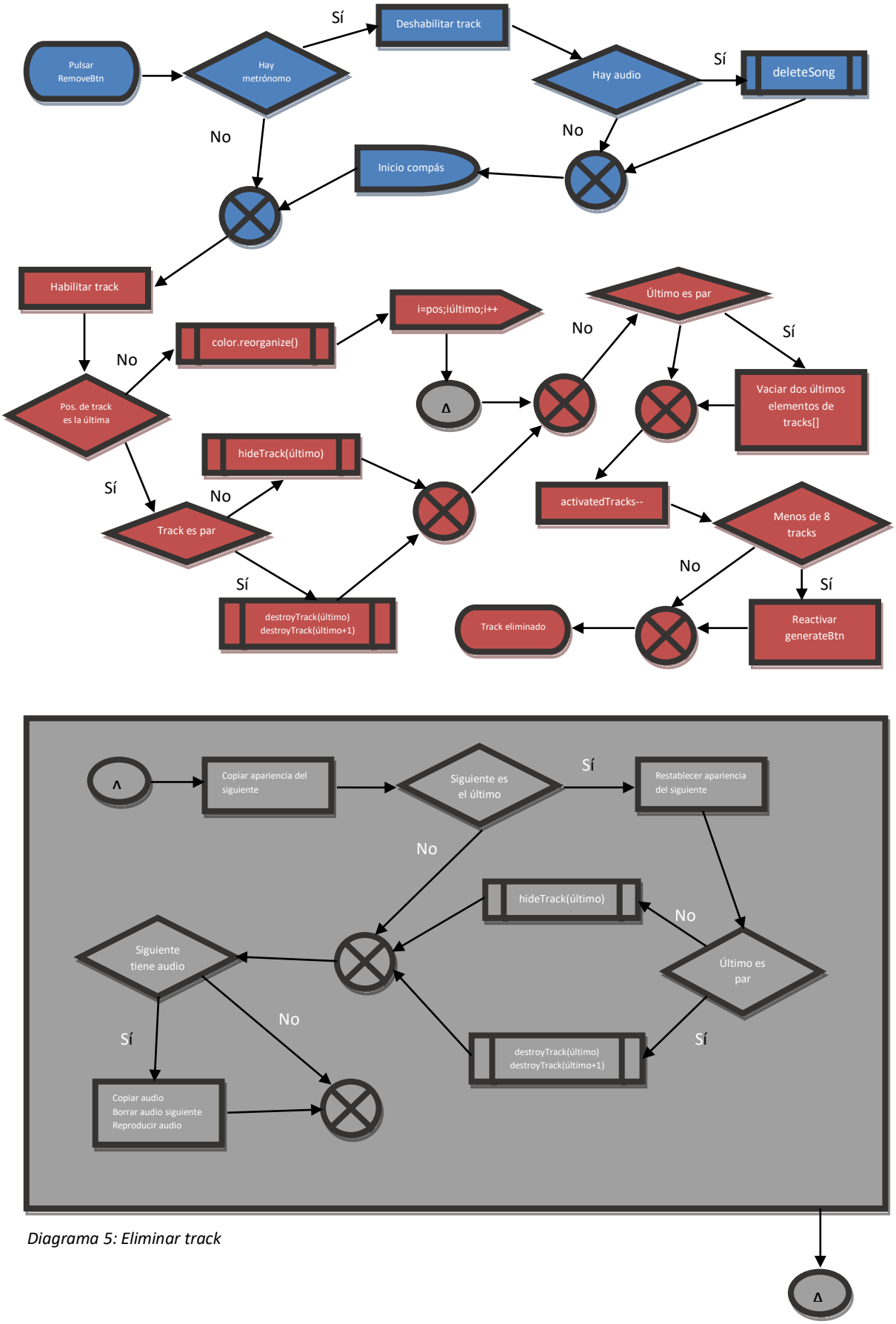


Diagrama 5: Eliminar track

En azul se muestran las tareas que se llevan a cabo en *prepareRemoveTrack()*, en rojo las que realiza *removeTrack()*y en gris las relativas a *transferTrack()*. Cuando se habla de último track, en este diagrama se refiere al track que se encuentra en la última posición.

La función *preRemoveTrack()* sirve para que el track no se elimine súbitamente, sino que lo haga a comienzo de compás y se mantenga la coherencia en el tempo. En el caso de que no haya ningún track reproduciéndose, se elimina sin más el track deseado, pues no hay que atenerse al tempo, al no existir un metrónomo. Para no perder la posición del track que se quiere eliminar se usa *trackToRemove* y al otorgar el valor *true* a la variable booleana *remove*, se espera a que *playMetronome()* comience un nuevo compás para que esta misma función llame a *removeTrack()* y comience la eliminación del track. Si el track contiene un archivo de audio, este puede ser eliminado ya desde *preRemoveTrack()* llamando a *deleteSong()*. Además, se deshabilita el track que va a ser eliminado, consiguiendo, por otro lado, que el usuario pueda identificar visualmente que ese track está a punto de ser eliminado.

Es en *removeTrack()* donde se completa la eliminación del track. Aquí existe una diferenciación entre los tracks con un pos *par*, que se encuentran a la izquierda de su fila, y los que tienen un *pos* impar, que se encuentran a la derecha, pues el primer *pos* otorgado es 0. El track relevante en este sentido es el último, que es el track realmente eliminado, sea cual sea el track que desee eliminar el usuario. Si este último track se encuentra a la derecha no se eliminará, sino que se esconderá, mientras que si está en el lado izquierdo sí se eliminará, además del siguiente track, el cual se encuentra oculto. Además, es importante restablecer los tracks a su estado inicial para futuros usos.

Pues bien, si, además, el usuario desea eliminar el último track, no hará falta reordenar ningún otro track, con loque se puede hacer directamente. Esto es lo que se hace en estas líneas de código:

```
if (pos == activatedTracks-1) { //Si el track borrado es el último
if (pos % 2 != 0) {
tracks[pos].hideTrack(false);
} else {
tracks[pos].destroyTrack(false);
tracks[pos + 1].destroyTrack(false);
}
}
```

Como se ha dicho, si el track a eliminar tiene una *pos* impar, simplemente se oculta. En caso contrario, se destruye junto al siguiente track, que está oculto.

Si el track que se va a eliminar no es el último, se deberán recolocar los siguientes tracks a este. Si, por ejemplo, se elimina el track con *pos* 4 y hay siete tracks, se eliminará el último track, con *pos* 6, pero el audio que se eliminará sí será el del track con *pos* igual a 4. Por lo tanto, en el track de *pos* 4 se copiará el de *pos* 5, y en este el de *pos* 6. Para que estos tracks reordenados conserven su color, se llama a *reorganize()* de la clase *ColorManager*. Así también se asegura que el próximo track creado tendrá el color del que fue eliminado y no se repetirán los colores. Para recolocar los tracks se llama a *transferTrack()* que copiará el siguiente track en el track con la *pos* que se pasa a dicha función. Se llama a *transferTrack()* dentro de un bucle for para recorrer desde el track eliminado hasta el penúltimo track.

Dentro de *transferTrack()*, en primer lugar se copia el aspecto del siguiente track. Esto es, si el siguiente track estaba reproduciendo una pista de audio, se cambiará la visibilidad de las views del track para que cambie al estado 4. No sólo eso, sino que, por ejemplo, también se copia el nivel de la *volumeBar* y el estado de los ToggleButtons.

Si el siguiente track ya es el último, éste no tendrá ningún track del que copiar el aspecto, por lo tanto se reinicia al estado 1 para futuros usos. Además, siguiendo la lógica de si ese último track es par o impar, se elimina junto al siguiente track oculto o se oculta dicho track.

Por último, si el siguiente track gestionaba un archivo de audio, ahora es este track el que debe gestionar ese archivo. Para ello, se copia dicho archivo, se asocia la copia al track sobre el que se está copiando, y se elimina el viejo archivo. Además, se reproduce el nuevo archivo creado, pues su antecesor también esta reproduciéndose. Con esto, ya se ha copiado en un track el siguiente.

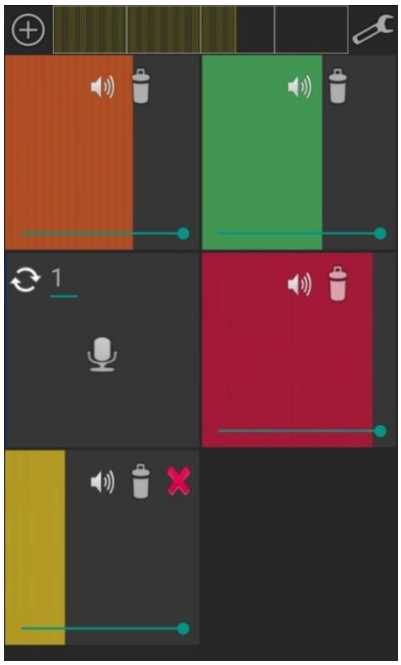
Una vez eliminado de la interfaz el último track, ya sea de manera directa o a través de *transferTrack()*, puede ser necesario eliminar los últimos tracks del array de tracks. Si el último track era impar, es decir, se encontraba a la derecha, éste no ha sido realmente eliminado, pues simplemente se ha ocultado. Por lo tanto no hay que eliminarlo de *tracks[]* ya que sigue existiendo. En cambio, si *pos* del que era el último track es par, tanto ese track como el siguiente sí han sido eliminados, por lo tanto también deben eliminarse de *tracks[]*. Sea como sea, ahora hay un track menos para el usuario, con lo cual *activatedTrack* debe contar uno menos.

Finalmente, si anteriormente había ocho tracks ahora habrá siete y, por lo tanto, puede volver a crearse un nuevo track, así que se rehabilita el botón de añadir track que debía estar inhabilitado.

Para detener el sonido del metrónomo, generado por la clase Metronome, se utiliza la función *stopMetronome()* . Aquí simplemente se detiene *copyMetronome*, que es el *Metronome* que realmente estaba sonando, y se cancela el thread *mThread*. Lo único que hace este thread es ejecutar el metrónomo en un hilo aparte. Debe realizarse una copia del metrónomo de nuevo en *copyMetronome* para un posterior uso. También es importante asignar el valor *false* a la variable booleana *metronomeSound*, pues el metrónomo ya no está sonando.

La función que gestiona el avance de la *MetronomeBar* se llama *startMBar()* y funciona de la misma manera que la función *startSquare()* de la clase *Tracks* vista anteriormente. En este caso, el PreciseCountdown que usa durará un compás y se llamará al método *onTick()* cada 5 milisegundos. En cada ejecución de *onTick()* se pinta una línea más llamando a *onDraw()* de la clase *MetronomeBar*, mientras que al finalizar llamando a *onFinish()* se limpia todo lo dibujado con *cleanCanvas()*.

Para finalizar, cuando se pulsa el botón de la esquina superior derecha este llama a *openSettings()*, encargada de abrir el menú de ajustes. En realidad se está creando una nueva activity, por lo tanto simplemente habrá que escribir esta línea de código: `startActivity(new Intent(MainActivity.this, MetronomePopUp.class))`.



En último lugar se muestra, a continuación, una captura de pantalla de la aplicación funcionando con grabaciones con distinto número de compases para ilustrar la apariencia final de SquareLoops.

7. Conclusiones

Para cerrar este proyecto, se va a revisar lo logrado durante el mismo y los objetivos que no han podido alcanzarse.

Por un lado, se ha conseguido desarrollar una aplicación funcional, capaz de grabar audio en pistas diferentes de manera independiente. El funcionamiento de estas pistas se ajusta perfectamente a los requisitos planteados, cumpliendo con lo exigible a un pedal de loop. No obstante, existen dos puntos en los que la aplicación flaquea:

- Si bien la sincronización es buena, en ocasiones existen pequeños desfases. Puesto que la intención de esta aplicación no es ser utilizada de manera profesional, sino como apoyo al músico o de un modo amateur, estos desfases de milisegundos son aceptables. Para perfeccionar la aplicación en este aspecto, podría utilizarse en un futuro el Kit de Desarrollo Nativo (NDK), que permite acceder a componentes físicos del dispositivos y actividades nativas a través de lenguaje C y C++.

- Tras finalizar una grabación, en el siguiente compás existe una desincronización. Esto es, durante ese compás no se dibuja ningún cuadrado y las pistas suenan descoordinadas entre sí. Una vez completado el compás, los audios se sincronizan y los cuadrados se pintan perfectamente coordinados. Este error no influye realmente pues basta con esperar un compás para escuchar los audios sincronizados. De nuevo, al no estar la aplicación enfocada a un uso profesional ni, desde luego, a ser utilizada en presentaciones en vivo, no es realmente relevante. Este error también se solucionaría utilizando el Kit de Desarrollo Nativo.

Por otro lado, en lo relativo a aprender a desarrollar una aplicación desde cero se han cumplido todos los objetivos propuestos, alcanzando varios conocimientos en este campo.

En definitiva, se ha conseguido desarrollar la aplicación propuesta, creando una herramienta útil y se han adquirido conocimientos valiosos para el futuro.

Bibliografía

Salazar, Jonathan (27/04/2017) "7 plataformas diferentes para desarrollar Android apps":

https://tekzup.com/7-plataformas-diferentes-desarrollar-android-apps/

(27/11/2017) "Crear apps móviles: Diferencias entre Android e iOS":

https://www.yeeply.com/blog/crear-apps-moviles-diferencias-android-e-ios/

Pascual, Juan Antonio (07/07/2018) "Android vs iPhone: la guerra de los smartphones en cifras":

https://computerhoy.com/reportajes/industria/android-vs-iphone-guerra-smartphones-cifras-271447

Music Critic (10/09/2018) "Top 10 Looper Pedals for Guitars":

https://musiccritic.com/equipment/looper-pedals/

JustinGuitar (04/08/2015) "How To Use ALooper Pedal - Guitar Lesson Tutorial - Justin Guitar [QA-004]:

https://www.youtube.com/watch?v=Gd0NhgIZWtw

ResisZienzia (16/05/2014) "Tempo, compás y ritmo: el latido de la música":

https://scirescience.wordpress.com/2014/05/16/tempo-compas-y-ritmo-el-latido-de-la-musica/

Alcalde, Alejandro (3/10/2017) "Fundamentos programación Android: Conceptos básicos y componentes":

https://elbauldelprogramador.com/fundamentos-programacion-android/

(26/08/2014) "Activity y Fragments":

https://academiaandroid.com/activity-y-fragments/

López, J. (08/03/2015) "Cómo crear las carpetas RAW y ASSETS para reproducir audio en Android Studio":

http://trucosandroidstudio.blogspot.com.es/2015/03/como-crear-las-carpetas-raw-y-assets.html

(22/03/2012) "Defining Global Variables in Android":

https://androidresearch.wordpress.com/2012/03/22/defining-global-variables-in-android/

"Interfaz de Usuario - Layout":

http://www.aprendeandroid.com/l4/interface1.htm

Revelo, James (23/12/2014) "AsyncTask: Tareas Asíncronas en Android":

http://www.hermosaprogramacion.com/2014/12/android-asynctask-hilos/

García, Fran (06/11/2015) "32. Hilos y AsincTask. (Programación Android Studio tutorial español):

https://www.youtube.com/watch?v=cxUJJh29PBg

Woodford, Chris (20/02/2019) "Sound":

https://www.explainthatstuff.com/sound.html

Smith, Sue (22/08/2013) "Android SDK: Create a Drawin App - Touch Interaction:"

https://code.tutsplus.com/tutorials/android-sdk-create-a-drawing-app-touch-interaction--mobile-19202

Wikipedia (13/03/2019) "Modulación por impulsos codificados":

https://es.wikipedia.org/wiki/Modulaci%C3%B3n_por_impulsos_codificados

Herramienta selectora de color hexadecimal:

https://htmlcolorcodes.com/es/selector-de-color/

"Símbolos de diagramas de flujo":

https://www.smartdraw.com/flowchart/simbolos-de-diagramas-de-flujo.htm