THINK it APP

PROYECTO DE SÍNTESIS

Memoria

INSTITUT PUIG CASTELLAR

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Alejandro Berdún Gómez y Jordi Solà Ceada

CURSO 2017-18

Índice de contenidos

Índice de figuras

ABSTRACT

This document is a memory of our final project for the Vocational Education Training degree on Multiplatform Software Development, which consists in an application for devices with Android Operating System. It is a project developed with the Integrated Development Environment Android Studio from Google, and we describe the process and difficulties of undertaking an app creation from scratch, from devising the main idea to launching the final product.

RESUMEN

Este documento es una memoria de nuestro proyecto final para el curso de Formación Profesional en Desarrollo de Aplicaciones Multiplataforma, que consiste en una aplicación para dispositivos con sistema operativo Android. Es un proyecto desarrollado con el Entorno de Desarrollo Integrado de Google Android Studio, y describimos el proceso y las dificultades de emprender la creación de una aplicación desde cero, desde el diseño de la idea principal hasta el lanzamiento del producto final.

RESUM

Aquest document és una memòria del nostre projecte final per al curs de Formació Professional en Desenvolupament d'Aplicacions Multiplataforma, que consisteix en una aplicació per a dispositius amb sistema operatiu Android. Es un projecte desenvolupat amb l'Entorn de Desenvolupament Integrat de Google Android Studio, i descrivim el procés i les dificultats d'iniciar la creació d'una aplicació des de zero, des del disseny de la idea fins al llançament del producte final.

1. Introducción

En un mundo en que gran parte de la sociedad cuenta con un smartphone en su bolsillo, el sector del desarrollo de software se encuentra en una fase de cambios, con una tendencia clara hacia el desarrollo de aplicaciones móviles y el uso de arquitecturas multiplataforma como la web.

Esto, por supuesto, repercute también en otros aspectos de la sociedad, referidos a las maneras en las que los humanos acometemos nuestras tareas del día a día. Es algo común que uno utilice la alarma del móvil para despertarse, calcule su ruta hacia el trabajo con la app de Google Maps, arranque su lista de reproducción de Spotify para amenizar el trayecto y durante la mañana envíe unos cuantos mensajes de WhatsApp a compañeros, familia o amigos. Probablemente durante la tarde consultará su cronología de Twitter o Instagram, y quizá incluso antes de dormir mire alguna serie online mediante una plataforma de televisión a la carta.

Esta situación se produce día a día en muchas de las rutinas de los habitantes de una sociedad del primer mundo, y el país donde nos encontramos no es una excepción. Aunque en el ejemplo expuesto, se utiliza la rutina frecuente de un adulto, esta tendencia también ha supuesto un cambio de hábitos a los más jóvenes. No es poco habitual encontrar a niños entreteniéndose con su tablet, o bien adolescentes iniciándose en el mundo de las redes sociales. Es por ello que su rutina diaria, o lo que es lo mismo, el mundo de la educación, también ha experimentado un gran cambio en los últimos años.

Figura 1: Jóvenes utilizando dispositivos móviles

Las aulas han pasado de tener pizarras con tiza a proyectores y pizarras interactivas. Los centros educativos han pasado de usar libretas y lápices, a ofrecer (o exigir) un ordenador portátil a cada alumno. Y los libros de texto han dado paso, en muchas materias, a los archivos PDF. Por ello es casi obligatorio ofrecer a las nuevas generaciones, nuevas formas de aprender, utilizando así el recurso de la tecnología como un medio para acercarles el aprendizaje y hacer que su percepción sobre la cultura del conocimiento sea agradable.

De este modo, los smartphones pueden convertirse en la plataforma perfecta para realizar este cometido, puesto que muchos jóvenes, independientemente del tipo de restricciones que apliquen sus padres o tutores, disponen de un smartphone. Esto también supondría un cambio de percepción en los padres, respecto a la utilización de un smartphone por parte de sus hijos.

El peligro de un smartphone depende del uso que se le dé, y en el caso de su utilización para educar a los más jóvenes, su propósito tendría buenas intenciones. Puede incluso motivar una mejora en la relación paterno/materno-filial y entre compañeros o amigos, en el caso de utilizar el marco de los videojuegos y los retos para fomentar el conocimiento.

2. Finalidad del proyecto

El objetivo principal de nuestro proyecto final es el desarrollo de una app para sistemas Android. La idea de esta app fue originada por una petición inicial por parte del Instituto Puig Castellar, para cubrir la necesidad de presentar una aplicación para el **Proyecto Switch**, que organiza nuestro instituto junto otros institutos de Europa dentro del marco de Erasmus+.

Nuestra primera fecha importante fue la presentación de la aplicación en el **evento "Multiplier"** de este año, ante profesores de los diferentes institutos colaboradores y alumnos de nuestro instituto, en febrero de 2018. Tras la presentación, definimos unas líneas de futuro basadas en nuevas funcionalidades de mediana y gran envergadura, tanto para la presentación de la aplicación como proyecto final del grado superior, como para futuras presentaciones y para un futuro desarrollo más detallado, con una distribución planificada.

La compilación realizada para el evento “Multiplier", con algunos ligeros cambios no sustanciales, fue además utilizada para presentarnos al concurso anual y autonómico que organiza la Fundació puntCAT, llamado WebsAlPunt.cat, que premia a las mejores páginas webs y aplicaciones móviles.

Finalmente, la aplicación también se presentaría en un evento final del Proyecto Switch, realizado a finales de junio en Birmingham, en el Halesowen College, para el cual el equipo de desarrollo y parte del profesorado impulsor de Erasmus+ viajarían una vez finalizado el curso.

Este proyecto nos ha permitido y nos permitirá aprender a fondo el desarrollo de aplicaciones para Android con el lenguaje de programación Java, utilizando las clases necesarias para todos los elementos multimedia y de conexión requeridos, aprovechando las herramientas que incorpora Android Studio para la integración de ficheros y de traducciones, así como obtener experiencia en las diferentes fases del desarrollo de una aplicación, desde el comienzo hasta el lanzamiento.

Figura 2: Logotipo del Proyecto Switch

3. Requisitos y objetivos de la aplicación

3.1 Requisitos de la aplicación

➔ **Matemáticas:**

Uno de los requisitos planteados en la fase de planificación para la compilación inicial para el proyecto Switch, era desarrollar una aplicación que **favorezca el aprendizaje de matemáticas**. Esto desembocó en la idea de un juego basado en sencillas operaciones matemáticas que el jugador debe completar. La idea principal es que aparecieran en pantalla operaciones como una suma, una resta, una multiplicación o una división, y que el usuario tenga que completarla. Más adelante se idearon mecanismos para dinamizar esta mecánica y añadirle un punto competitivo, que se explicarán en siguientes apartados.

➔ **Multilenguaje:**

Otro requisito que debía cumplir el producto, y con bastante lógica tratándose de un complemento de un proyecto multicultural, es permitir elegir un idioma de entre los siguientes cuatro:

- **Castellano**
- **Catalán**
- **Inglés**
- **Italiano**

Una vez elegido el idioma deseado, la interfaz debe cambiar, así el usuario de la aplicación podrá sentirse más cómodo y entenderá más fácilmente el producto. Si bien puede servir además de herramienta para aprender otro idioma, aunque no es el enfoque principal de nuestra app por el momento.

➔ **Jóvenes**:

Finalmente, se definió como público objetivo a los niños y adolescentes, tal y como se explica en la introducción de este documento: gente joven que se encontrara en una etapa de aprendizaje de matemáticas básicas. Tratándose de un proyecto realizado entre distintos centros educativos del mundo, era lógico que este fuera nuestro público objetivo. No obstante, no hay nada que impida a usuarios de otras edades utilizar nuestra app para mejorar su cálculo mental.

3.2 Objetivos de la aplicación

Para cumplir con los requisitos mencionados, establecimos una serie de objetivos a nivel de producto:

- **Mejora del cálculo mental:** Con un uso seguido de la app, cualquier usuario tiene que llegar a la conclusión de que su habilidad de cálculo mental ha mejorado y es capaz de realizar operaciones matemáticas básicas en menos tiempo.
- **Atractivo visual:** La aplicación debe tener unos gráficos y una gama de colores apropiada para ser un juego, aunque debemos cumplir ciertos mínimos de usabilidad.
- **Atracción del público joven:** Es importante añadir características que supongan retos para el público objetivo, propiciando opiniones favorables y una predisposición a instalar la app en sus dispositivos.
- **Entretenimiento:** El juego debe ofrecer una mecánica que guste al público, que permita sesiones de juego con una duración decente, y que les genere un sentimiento de satisfacción.
- **Inclusividad tecnológica:** La aplicación debe poder ser ejecutada desde un amplio rango de dispositivos, incluyendo una escalabilidad de la interfaz y un soporte tecnológico del código utilizado.
- **Documentación y ayuda:** Para hacer más fácil los primeros pasos con la aplicación, se debe incluir un apartado de ayuda, un tutorial de iniciación y diferentes pistas eventuales durante el juego.

4. Plataforma

Como se ha acordado con el profesorado encargado de Erasmus+, se ha decidido desarrollar la app para sistemas Android. Tras analizar las estadísticas de uso de las diferentes versiones de Android, y poner en una balanza las ventajas y desventajas del apoyo a versiones antiguas, hemos decidido fijar una versión mínima de **Android 4.4**.

Según datos oficiales de Google[1], se calcula que un 94,8% de los usuarios de Android utilizan una versión igual o superior a la 4.4. También, las estadísticas señalan[2] que un 75% de los usuarios de teléfonos inteligentes utilizan Android, en vez de otros sistemas como iOS o Windows Phone / Mobile. Esto son razones suficientes para decidir Android como sistema de salida, aunque no cerramos la puerta a un futuro lanzamiento de la aplicación en otros sistemas operativos.

Respecto al lenguaje utilizado para el desarrollo, hemos decidido programar en Java, en vez de en Kotlin, al ser un lenguaje más familiar para nosotros, teniendo en cuenta los contenidos del ciclo formativo que hemos puesto en práctica durante este año.

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.4%
4.1.x	Jelly Bean	16	1.7%
4.2.x		17	2.2%
4.3		18	0.6%
4.4	KitKat	19	10.5%
5.0	Lollipop	21	4.9%
5.1		22	18.0%
6.0	Marshmallow	23	26.0%
7.0	Nougat	24	23.0%
7.1		25	7.8%
8.0	Oreo	26	4.1%
8.1		27	0.5%

Figura 3: Porcentaje de distribución de las versiones de Android

Figura 4: Logotipo de Android 4.4 (KitKat)

1 Android Developers: Paneles de control (consultado 01/05/2018)

2 StatConter: Mobile Operating System Market Share Worldwide (consultado 01/05/2018)

5. Aspectos críticos de implementación

5.1 Soporte para Android 4.4 o superior

Uno de los aspectos más críticos del desarrollo, es que la aplicación funcione correctamente en versiones de Android poco recientes como la 4.4 (2013), ya que supone la falta de funcionalidades más recientes (como por ejemplo clases de versiones más recientes de Java, y propiedades relacionadas con los *guidelines* de **Material Design** de Google).

Esto dificulta claramente el proceso de implementarlas, ya que no funcionan en dispositivos más antiguos y se deben utilizar ciertas alternativas más complejas para conseguir el mismo resultado.

5.2 Cambio de idioma

Android no facilita el cambio de idioma fuera del menú de configuración del sistema operativo, ya que las aplicaciones establecen por defecto el idioma del dispositivo. Esto puede ser un problema en el caso de querer ofrecer un sistema integrado de cambio de idioma explícito dentro de la aplicación. Como el origen de nuestra aplicación es la colaboración entre diferentes países y el aprendizaje, esta funcionalidad es un requisito indispensable.

Esto nos obliga a implementar ciertas clases y mecanismos para almacenar el idioma como parámetro de configuración de la aplicación, así como cambiar el lenguaje mostrado y recargar la interfaz para mostrar inmediatamente el cambio de lenguaje.

5.3 Audio y música

El SDK de Android no es un entorno especialmente propicio para el desarrollo de videojuegos, es por eso que la reproducción intensiva de diferentes archivos de audio de forma concurrente resulta habitualmente en una fuente de problemas y *bugs,* que pueden desencadenar situaciones en las dejen de funcionar la música o los efectos de sonido.

También relacionado con el audio, es obligatorio implementar mecanismos para controlar la reproducción de ficheros según el ciclo de vida de Android. Es decir, detener cualquier sonido cuando la app se encuentra en segundo plano, y reanudar la reproducción cuando se vuelva a abrir, así como evitar la reproducción simultánea de dos ficheros que no deberían reproducirse a la vez.

5.4 Modo multijugador

Uno de los objetivos a largo plazo de cara a la presentación definitiva para el proyecto de síntesis es la implementación de determinadas funcionalidades que permitan jugar a nuestra app con otros jugadores en tiempo real, formando una nueva experiencia de juego competitiva. En el lado técnico, esto requiere la implementación de una arquitectura en red que incorpore diferentes roles de cliente-servidor e incluso *Peer-to-Peer*.

Este modo supone la utilización de clases de Java para la comunicación TCP y UDP, y es una de las partes más críticas de implementación debido a la dificultad para depurar el programa y evitar situaciones de excepción. Las aplicaciones en red deben ejecutar una lógica muy estricta para evitar que un movimiento no esperado interrumpa el flujo normal de la app, teniendo que controlar tanto el tráfico entrante como el saliente.

5.5 Almacenamiento persistente

Uno de los puntos clave para generar en el jugador una sensación de reto y continuidad, es diseñar un sistema que almacena las puntuaciones de éste, y que pueda así compararlas. Existen varias posibilidades para implementar esto, como almacenar los datos en el archivo de preferencias de la app (clase SharedPreferences de Android SDK), o bien implementar una base de datos.

Finalmente se ha decidido diseñar una base de datos para guardar información de forma persistente, de tal forma que se mantiene guardada aunque se cierre la aplicación. Se implementará una base de datos de tipo relacional y las puntuaciones almacenadas se mostrarán de forma descendiente en un ranking de puntuaciones.

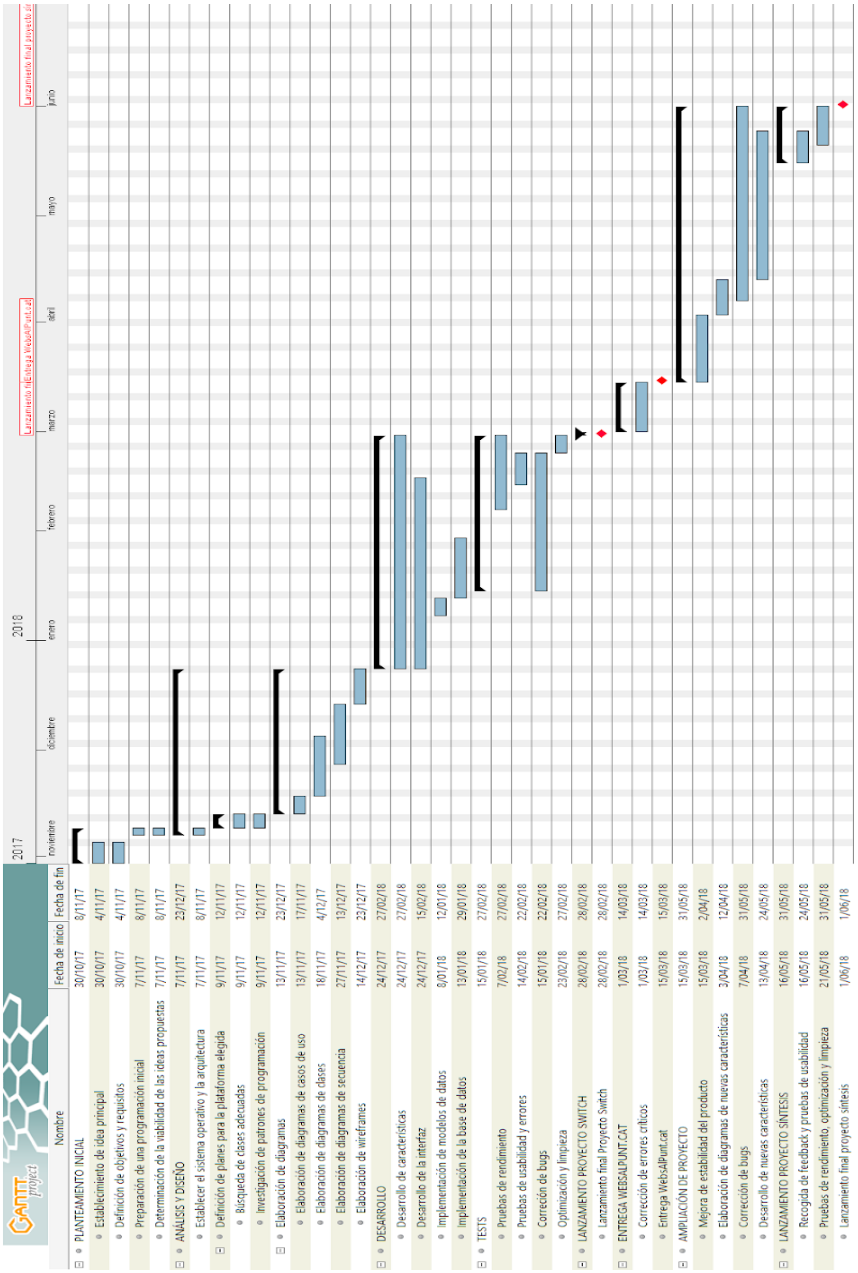
Figura 5: Logotipo de Android Studio

6. Planificación - Diagrama de Gantt

Nuestra planificación incluye hasta 3 *milestones*:

- **28/02/2018** → Evento "Multiplier" del Proyecto Switch.
- **15/03/2018** → Entrega para WebsAlPunt.cat (presentación: 11/05/2018).
- **01/06/2018** → Entrega para proyecto de síntesis de DAM.

Además, hay un cuarto milestone que se producirá el **28/06/2018**, que consiste en la presentación de la app en el evento final del Proyecto Switch, aunque no se recoge en el diagrama puesto que se utilizará la compilación final entregada en el proyecto de síntesis.



Nombre	Fecha de inicio	Fecha de fin
PLANTEAMIENTO INICIAL	30/10/17	8/11/17
Establecimiento de idea principal	30/10/17	4/11/17
Definición de objetivos y requisitos	30/10/17	4/11/17
Preparación de una programación inicial	7/11/17	8/11/17
Determinación de la viabilidad de las ideas propuestas	7/11/17	8/11/17
ANÁLISIS Y DISEÑO	7/11/17	23/12/17
Establecer el sistema operativo y la arquitectura	7/11/17	8/11/17
Definición de planes para la plataforma elegida	9/11/17	12/11/17
Búsqueda de clases adecuadas	9/11/17	12/11/17
Investigación de patrones de programación	9/11/17	12/11/17
Elaboración de diagramas	13/11/17	23/12/17
Elaboración de diagramas de casos de uso	13/11/17	17/11/17
Elaboración de diagramas de clases	18/11/17	4/12/17
Elaboración de diagramas de secuencia	27/11/17	13/12/17
Elaboración de wireframes	14/12/17	23/12/17
DESARROLLO	24/12/17	27/02/18
Desarrollo de características	24/12/17	27/02/18
Desarrollo de la interfaz	24/12/17	15/02/18
Implementación de modelos de datos	8/01/18	12/01/18
Implementación de la base de datos	13/01/18	29/01/18
TESTS	15/01/18	27/02/18
Pruebas de rendimiento	7/02/18	27/02/18
Pruebas de usabilidad y errores	14/02/18	22/02/18
Corrección de bugs	15/01/18	22/02/18
Optimización y limpieza	23/02/18	27/02/18
LANZAMIENTO PROYECTO SWITCH	28/02/18	28/02/18
Lanzamiento final Proyecto Switch	28/02/18	28/02/18
ENTREGA WEBSALPUNT.CAT	1/03/18	14/03/18
Corrección de errores críticos	1/03/18	14/03/18
Entrega WebsAlPunt.cat	15/03/18	15/03/18
AMPLIACIÓN DE PROYECTO	15/03/18	31/05/18
Mejora de estabilidad del producto	15/03/18	2/04/18
Elaboración de diagramas de nuevas características	3/04/18	12/04/18
Corrección de bugs	7/04/18	31/05/18
Desarrollo de nuevas características	13/04/18	24/05/18
LANZAMIENTO PROYECTO SÍNTESIS	16/05/18	31/05/18
Recogida de feedback y pruebas de usabilidad	16/05/18	24/05/18
Pruebas de rendimiento, optimización y limpieza	21/05/18	31/05/18
Lanzamiento final proyecto síntesis	1/06/18	1/06/18

Figura 6: Diagrama de Gantt

7. Planificación - Diagrama de casos de uso

En el siguiente diagrama se definen las diferentes acciones que el jugador (que en este caso es el único actor) puede realizar en toda la aplicación, incluyendo las dependencias y extensiones de cada acción respecto a otra.

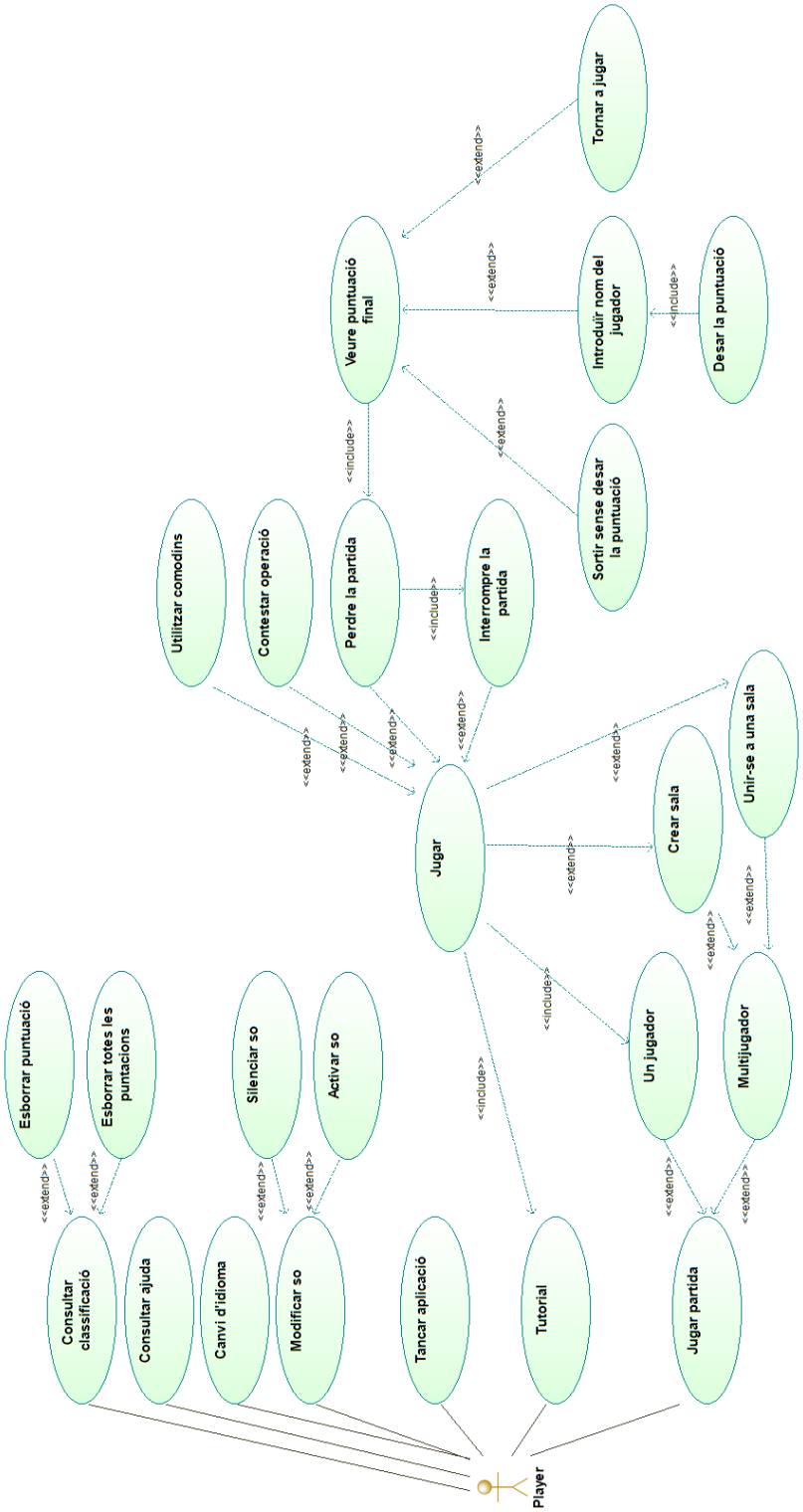


Figura 7: Diagrama de casos de uso

8. Planificación - Diagrama de actividad

En el siguiente diagrama se definen las diferentes acciones que el jugador puede hacer o a las cuales se somete, en el flujo de actividad del modo de juego individual:

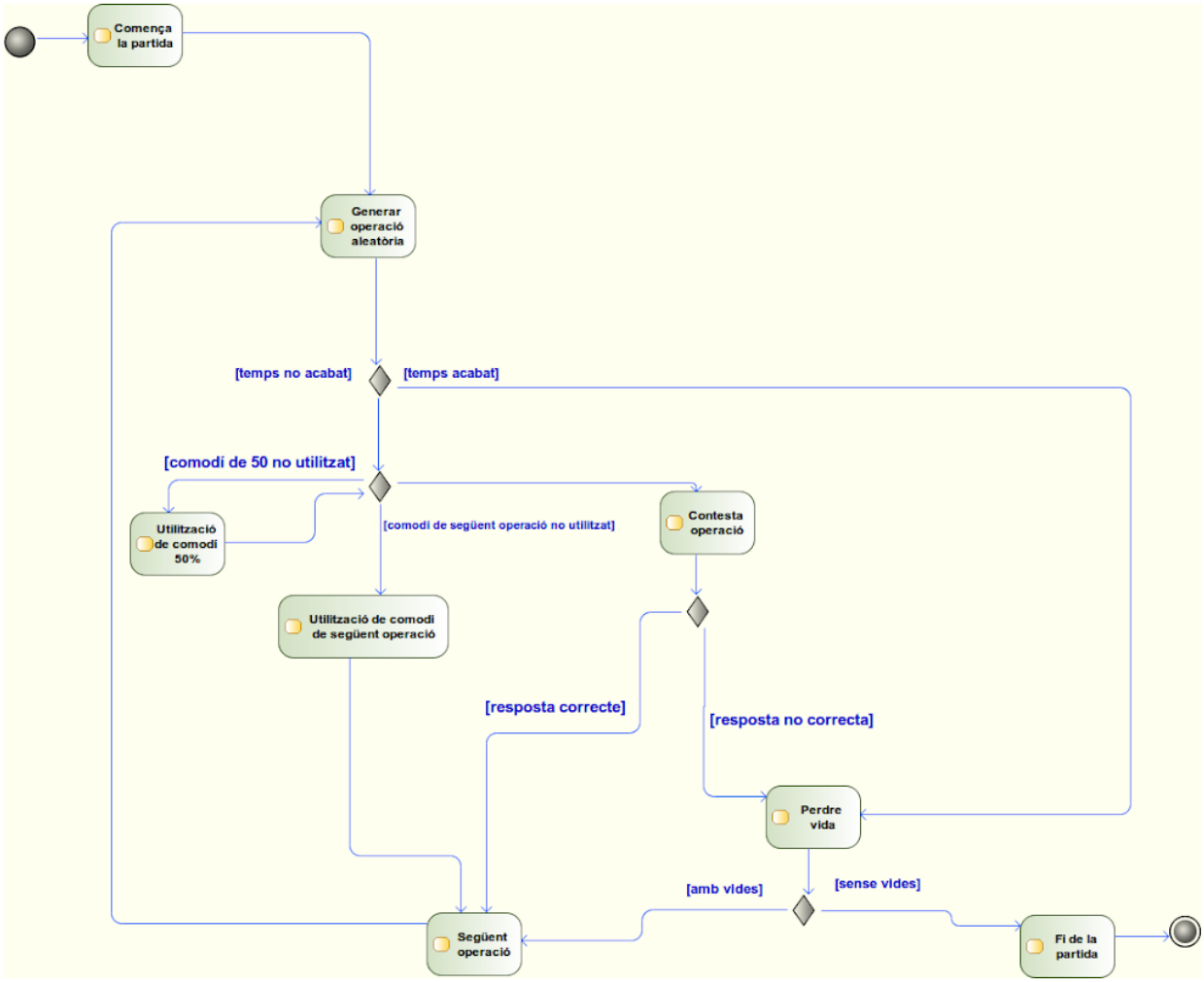


Figura 8: Diagrama de actividad del juego

9. Implementación

9.1 Menú principal

El menú principal, representado en la actividad MainActivity, contiene el logo de la aplicación (mediante un ImageView), un botón de jugar (mediante otro ImageView con cambios de imagen al pulsarlo), y todos los FAB (Floating Action Button) que permiten navegar por las funcionalidades que la aplicación ofrece al jugador.

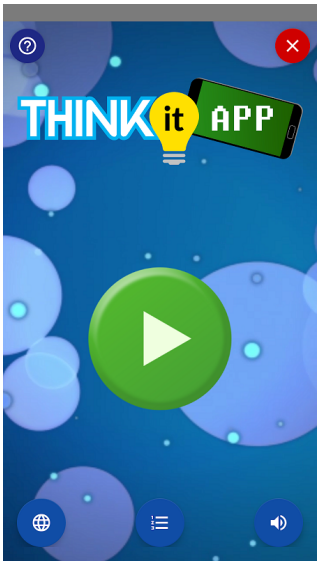


Figura 9: Interfaz del menú principal (MainActivity)

En este caso, y para permitir un ajuste dinámico de los elementos de la interfaz en diferentes resoluciones de pantalla, hemos utilizado un **ConstraintLayout** como ViewGroup del layout.

Los layouts en Android se basan en ficheros XML, que, como todo XML bien formado, requiere de una etiqueta raíz. En este caso, la etiqueta raíz corresponde al grupo principal que engloba las vistas (o ViewGroup), y define la estructura principal de la interfaz (si es lineal, si es una cuadrícula...). Las vistas que incluye el ViewGroup principal puede ser otros ViewGroups anidados, o bien widgets y controles como botones o imágenes.

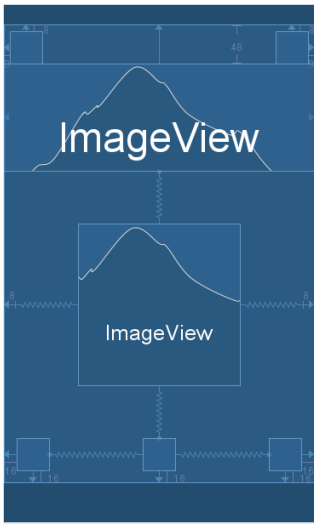


Figura 10: Blueprint de la MainActivity

El tipo ConstraintLayout permite definir un posicionamiento dinámico, basado en puntos de anclaje entre elementos del layout y entre los elementos y el borde de la pantalla, pudiendo crear espacios dinámicos entre elementos, que se amplian o reducen dependiendo del tamaño de la pantalla.

Volviendo a la implementación del código, hemos incluido también un fondo animado, que reproducimos en bucle utilizando la clase **VideoView** y para la música utilizamos otra clase llamada **MediaPlayer**. En el caso del vídeo, se trata de un fichero MP4, y en el caso de la música es un fichero MP3.

Este es el código utilizado para inicializar las variables:

```
// Inicializar MediaPlayer con el fichero a reproducir
musicPlayer = MediaPlayer.create(this, R.raw.modern_theme_nicolai_heidlas);

// Inflar VideoView (obtener elemento del layout)
bgVideo = findViewById(R.id.bg_video);

// Indicar el fichero de vídeo a reproducir
bgVideo.setVideoURI(Uri.parse("android.resource://net.xeill.elpuig.thinkitapp/" +
R.raw.background2));
```

Para reproducirlos, utilizamos sendos métodos *start()*.

```
bgVideo.start();
musicPlayer.start();
```

9.1.1 El ciclo de vida de las actividades

Debemos controlar la reproducción multimedia cuando la actividad cambia de estado, ya que tanto el vídeo o la música pueden dejar de funcionar correctamente si no se gestionan los cambios de estado.

El ciclo de vida de las actividades define una serie de estados por los que pasa una actividad durante su ejecución. Por ejemplo, si otra actividad pasa a primer plano, la anterior se pausa, si la actividad no es visible, se detiene, y si el sistema la destruye, se elimina de memoria. Otras acciones que pueden desencadenar un cambio de estado es dejar la app en segundo plano (pulsando el botón *Home* de Android) o bien girar el dispositivo para pasar a modo horizontal.

La clase Activity (o en su defecto, AppCompatActivity en caso de utilizar librerías de soporte) ofrece una serie de métodos, que al implementar una actividad se pueden sobreescribir, y definir sentencias que se ejecutarán cuando la actividad pase a un determinado estado.

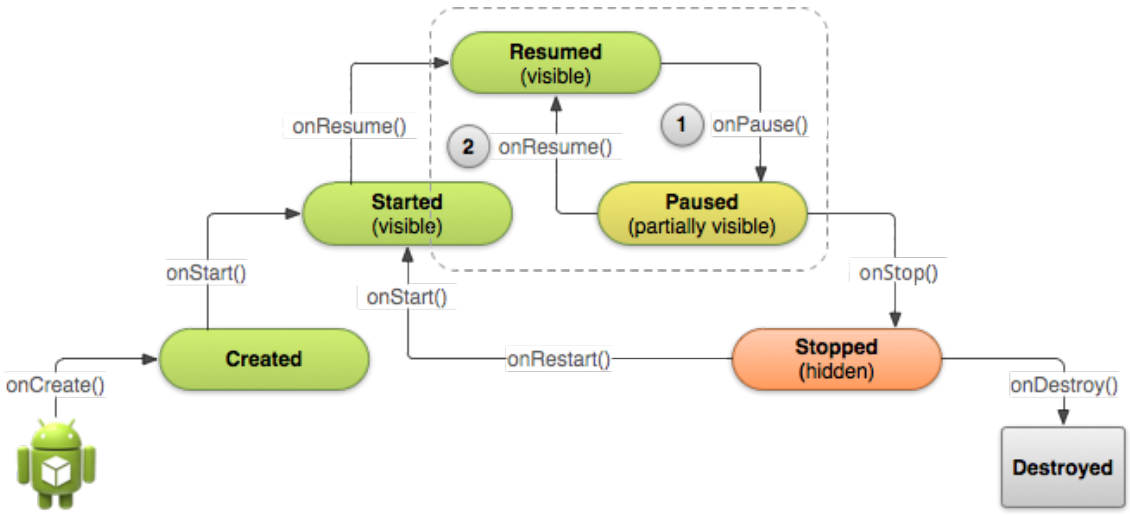


Figura 11: Ciclo de vida de una actividad (Fuente: Android Developers).

Para gestionar estos estados, sobreescribimos los métodos *onPause()* y *onResume()* y añadimos una serie de flags y comprobaciones con tal de pausar los vídeos cuando la actividad pase a estado 'Pause', y para reactivarlos cuando vuelve al estado 'Resume'.

```
// Cuando el estado de la actividad cambie a Paused, se ejecutará éste método
@Override
protected void onPause() {
    super.onPause();
    // Pausa la música si no estaba pausada ya
    if(!musicPaused){
        musicPlayer.pause();
    }
    // Pausa la reproducción del video si se está reproduciendo
    if(bgVideo!=null && bgVideo.isPlaying()){
        bgVideo.pause();
    }

// Cambiamos el valor del flag musicPaused a true ya que las anteriores sentencias
de control han pausado mediaPlayer
    musicPaused = true;
}
```

```
// Cuando el estado de la actividad cambie a Resumed, se ejecutará éste método
@Override
protected void onResume() {
    super.onResume();
    // Reproducimos la música si se encuentra pausada
    if(musicPaused){
        musicPlayer.start();
    }
    // Reproducimos el video si no se está reproduciendo
    if(bgVideo!=null && !bgVideo.isPlaying()){
        bgVideo.start();
    }
    // Cambiamos el valor del flag musicPaused a false
    musicPaused = false;
}
```

9.1.2 Selector de idioma

En esta pantalla permitimos al jugador elegir el idioma que desee, permitiendo así que nuestra app sea utilizada por un conjunto más amplio de personas, ya que facilita la lectura y la comprensión de todo el texto que contiene la aplicación.

Esta actividad está basada también en un ConstraintLayout, incluyendo un GridLayout anidado para crear una table con las banderas de cada idioma.

Al iniciar la aplicación por primera vez, nos preguntará cuál de los idiomas preferimos, y a partir de entonces mostrará las cadenas de texto en el idioma que el jugador haya seleccionado.

Para gestionar el cambio de idioma, hemos creado una clase específica con este propósito (LocaleManager).

Figura 12: Pantalla de selección de idioma (LanguageActivity)

```
public class LocaleManager {
    public static void setLocale(Context context, String lang) {
        //Crear Locale
        Locale myLocale = new Locale(lang);

        //Obtener resources actuales
        Resources res = context.getResources();
        DisplayMetrics dm = res.getDisplayMetrics();
        Configuration conf = res.getConfiguration();

        //Establecer locale y actualizar resources
        conf.locale = myLocale;
        res.updateConfiguration(conf, dm);

        //Guardar preferencia
        SharedPreferences preferences = context.getSharedPreferences("prefs", 0);
        preferences.edit().putString("language",lang).apply();
    }
}
```

Desde LanguageActivity, asignamos un nuevo *View.OnClicListener* a cada bandera, y sobreescribimos el método *onClick()* para añadir un código semejante a este:

```
        es_flag.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                LocaleManager.setLocale(LanguageActivity.this,"es");
                menuIntent(getResources().getString(R.string.es_flag));
            }
        });
```

Nótese que inmediatamente después de cambiar el idioma, ejecutamos el método *menuIntent()*, el cual incluye un Intent[3] y la utilización del método heredado *finish()*, que harán que salgamos de esta actividad y se abra la MainActivity (menú principal).

```
private void menuIntent(String language){
    Toast.makeText(this,getResources().getString(R.string.language_change_toast) +
language, Toast.LENGTH_SHORT).show();
    if (settings.getBoolean("isFirstRun",true)) {
        settings.edit().putBoolean("isFirstRun",false).apply();
        finish();
        Intent firstMain = new Intent(LanguageActivity.this,MainActivity.class);
        startActivity(firstMain);
    } else {
        LanguageActivity.this.finish();
    }
    playSoundPlayer.start();
}
```

Cabe destacar también la utilización de la clase **SharedPreferences** para gestionar una variable booleana, para diferenciar en el código cuándo es la primera vez que se accede a la aplicación y cuándo no.

En caso de ser la primera vez que se accede, se da por hecho que cuando el usuario selecciona un idioma, ya se debe considerar que no es la primera vez que accede. En caso de no llegar a seleccionar un idioma, las próximas veces que se ejecute la app, se consideraría "primera vez" hasta que se seleccione uno.

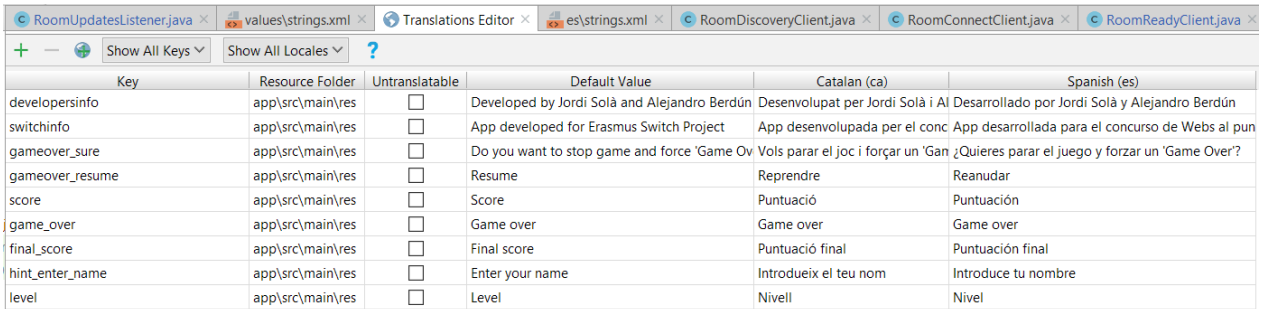
```
private SharedPreferences settings;

[...]

if (settings.getBoolean("isFirstRun",true)) {
        settings.edit().putBoolean("isFirstRun",false).apply();
```

Traducción de 'strings'

El soporte de múltiples idiomas es posible gracias a que Android Studio permite generar diferentes ficheros de localización para cada idioma distinto. Google facilita el desarrollo de apps para distintas localizaciones, ofreciendo el **Translations Editor**, una herramienta que permite la edición de cadenas de texto para diferentes idiomas de forma gráfica, a partir de una tabla.



Key	Resource Folder	Untranslatable	Default Value	Catalan (ca)	Spanish (es)
developersinfo	app\src\main\res	☐	Developed by Jordi Solà and Alejandro Berdún	Desenvolupat per Jordi Solà i Al	Desarrollado por Jordi Solà y Alejandro Berdún
switchinfo	app\src\main\res	☐	App developed for Erasmus Switch Project	App desenvolupada per el conc	App desarrollada para el concurso de Webs al pun
gameover_sure	app\src\main\res	☐	Do you want to stop game and force 'Game Ov	Vols parar el joc i forçar un 'Gan	¿Quieres parar el juego y forzar un 'Game Over'?
gameover_resume	app\src\main\res	☐	Resume	Reprendre	Reanudar
score	app\src\main\res	☐	Score	Puntuació	Puntuación
game_over	app\src\main\res	☐	Game over	Game over	Game over
final_score	app\src\main\res	☐	Final score	Puntuació final	Puntuación final
hint_enter_name	app\src\main\res	☐	Enter your name	Introdueix el teu nom	Introduce tu nombre
level	app\src\main\res	☐	Level	Nivell	Nivel

Figura 13: Translations Editor de Android Studio

3 - Un "intent" en el SDK de Android representa un acceso a otra actividad (ventana) de la aplicación.

9.1.3 Ranking de puntuaciones

Cuando finaliza una partida, se le da la opción al jugador de guardar la puntuación obtenida. Una vez guardada una o más puntuaciones, se pueden consultar en una clasificación accesible desde el menú principal (aunque también se accede automáticamente justo después de guardar una puntuación). Esta lista muestra el nombre y la puntuación de los jugadores.

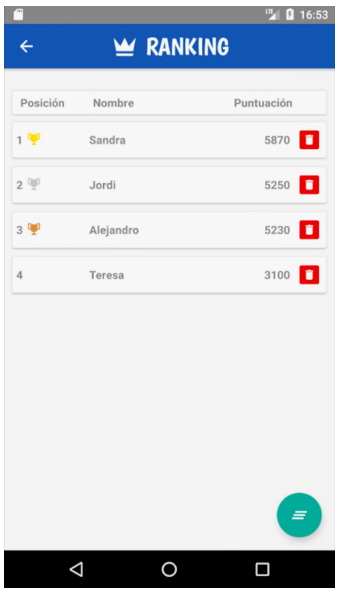


Figura 14: Pantalla de ranking de puntuaciones.

Para permitir una utilización eficiente de recursos, y que ello repercuta en un desplazamiento vertical fluido, utilizamos un **RecyclerView**, que es un tipo de lista más avanzado que los ListView. La utilización de un RecyclerView está pensada en listas que se componen de una gran cantidad de datos o en las que sus datos se actualizan constantemente.

Antes de mostrar las puntuaciones, primero las ordenamos de forma descendente y a continuación se muestran al usuario mediante un **CardView** como base del *view holder*[4], para mejorar así la consistencia de la lista y la visibilidad de cada elemento.

```
// Cuando haya cambios en la lista, se ejecutará éste método
@Override
public void onChanged(@Nullable List<Score> scores){
    [...]

    // Ordena de forma descendente la lista de puntuaciones
    Collections.sort(scoreList);

    // Notifica al adapter del RecyclerView que los datos de la lista han cambiado
    scoreRecyclerAdapter.notifyDataSetChanged();

    [...]

}
```

El jugador puede eliminar todas las puntuaciones pulsando el FAB verde, y también eliminar una puntuación individual de la lista pulsando el botón de la papelera.

4 - Un *view holder* es una clase que agrupa y convierte a objetos Java ("infla") aquellos elementos del XML que representan un item de la lista, y que son susceptibles de cambiar dependiendo de cada item.

La base de datos

ThinkItApp utiliza una base de datos relacional para el almacenamiento de las puntuaciones de los jugadores. Esta base de datos se ha implementado mediante la librería de persistencia **Room**, que Google incluye dentro de la colección "Android Architecture Components", que ofrece a los desarrolladores clases que facilitan la gestión del ciclo de vida de las apps, y robustas abstracciones de clases muy utilizadas.

En este caso, Room es una abstracción de la API del sistema gestor de bases de datos **SQLite**. Permite la vinculación de POJOs[5] con entidades de la base de datos mediante anotaciones de Java (los llamados Data Transfer Objects), así como la creación de un Data Access Object (DAO), donde se incorporan sentencias SQL vinculadas a métodos, que otras clases pueden ejecutar de forma aislada.

De este modo, se intenta dividir el código en las capas que establece el patrón de Modelo-Vista-Modelo de Vista, donde el **modelo** serían la base de datos, el DAO y los POJOs, el **modelo de vista** ("ViewModel") sería una clase encargada de ejecutar los métodos del DAO, y las **vistas** serían los ficheros XML que definen las interfaces y la lógica tras las actividades (esta última parte es la que llama a los métodos del ViewModel y se suele llamar **"View Controller"**).

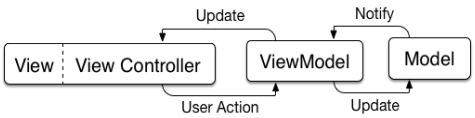


Figura 15: Esquema del patrón de Modelo-Vista-Modelo de Vista. Fuente: Medium.

En nuestro proyecto, disponemos de una clase abstracta denominada **ScoreDatabase**, que no es más que un *Singleton*[6] que se encarga de instanciar la base de datos de forma estática. Esta clase utiliza la anotación **@Database**, proveniente de la librería Room. Con ella se establece las entidades que se usarán en la base de datos, y la versión de esta (es un número que habrá que incrementar en sucesivos cambios de esquema).

Esta clase también sirve de intermediario para obtener la clase del DAO.

```
@Database(entities = {Score.class}, version = 1)
public abstract class ScoreDatabase extends RoomDatabase {
    private static ScoreDatabase INSTANCE;
    public static ScoreDatabase getInstance(Context context) {
        if (INSTANCE == null) {
            INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
ScoreDatabase.class, "db")
                            .fallbackToDestructiveMigration()
                            .build();
        }
        return INSTANCE;
    }
    public abstract ScoreDao getScoreDao();
}
```

5 - POJO: Plain Old Java Object. Clase de Java que apenas incorpora atributos, constructor, *getters* y *setters*.

6 - Patrón que se utiliza para que una clase solo tenga una implementación en toda la aplicación.

En el proyecto también tenemos una clase llamada **Score**, que actúa de DTO. Esta clase también utiliza diferentes anotaciones provenientes de la librería Room:

- **@Entity:** Marca la clase como una entidad de la base de datos. Esto generará una tabla de SQLite con los atributos definidos en ella.
- **@PrimaryKey (autoGenerate = true):** Establece el atributo de la línea inferior (en este caso, "id") como clave principal de la entidad. También le indicamos que sea autogenerada, lo que corresponde con un campo de tipo "serial" en SQL.

Como se puede comprobar, los tipos de los atributos deben ser compatibles con los que soporta SQL: podemos utilizar int, long, String o Date, entre otros.

```
@Entity
public class Score implements Comparable<Score>, Serializable {
    @PrimaryKey (autoGenerate = true)
    private long id;
    private String user;
    private int score;
    private int level;
    private int correctAnswers;
    private int mistakes;
```

Esta clase debe definir también los correspondientes *getters* y *setters* para que funcione correctamente:

```
public String getUser() {
    return user;
}
public void setUser(String user) {
    this.user = user;
}
public int getScore() {
    return score;
}
public void setScore(int score) {
    this.score = score;
}

[···]
```

Después, tenemos una **interfaz de Java que actúa de DAO**, y que define las firmas de los métodos que se comunicarán con la base de datos, así como vincular con ellas las sentencias SQL.

En esta interfaz se utilizan otras anotaciones de Room:

- **@Dao:** Marca la clase como un Data Access Object. Esta debe ser o bien una interfaz (como es el caso) o bien una clase abstracta.
- **@Query("select * from score"):** Sirve para vincular el método de abajo con la sentencia SQL definida literalmente en entre paréntesis. Puede ser una sentencia de cualquier tipo, aunque se suele utilizar para SELECTs, ya que el resto de sentencias se pueden realizar más fácilmente mediante otras anotaciones, como las que veremos a continuación.
- **@Insert(onConflict = 1):** Vincula el método de abajo con una sentencia de inserción (INSERT). El objeto a insertar será aquel enviado por parámetro al método (que también puede ser una lista de objetos) y la entidad correspondrá a su tipo, mientras que Room se encargará de construir la sentencia. El parámetro "onConflict" en 1 establece que se reemplazarán los datos en caso de ya existir.
- **@Delete:** Vincula el método de abajo con una sentencia de eliminación (DELETE). Funciona del mismo modo que @Insert.

```
@Dao
public interface ScoreDao {
    @Query("select * from score")
    LiveData<List<Score>> getScores();
    @Insert(onConflict = 1)
    long insertScore(Score score);
    @Delete
    void deleteScore(Score score);
    @Delete
    void deleteAllScores(List<Score> scoreList);
}
```

También tenemos una clase encargada de la parte del **"modelo de vista"** (*viewmodel*). Esta clase extiende de la clase AndroidViewModel, que permite que los accesos a persistencia sean "context-aware", lo cual permite evitar fugas de memoria en casos en los que cambie el estado de la aplicación, según el ciclo de vida explicado con anterioridad.

Para que esto sea posible, se debe definir un constructor que solo reciba por parámetro un objeto de tipo Application. En este constructor, además se obtiene la implementación del DAO a partir de la instancia de ScoreDatabase.

```
public ScoreViewModel(Application application) {
    super(application);
    scoreDao = ScoreDatabase.getInstance(this.getApplication()).getScoreDao();
}
```

Es en esta clase donde se definen los métodos que utilizarán las vistas para la gestión de la base de datos. Los métodos de esta clase, por su parte, incluyen tareas asíncronas en las que se llama a los métodos definidos en el DAO.

Por ejemplo, este es el código para la llamada a la eliminación de una sola puntuación. Nótese que este método recibe por parámetro un Score, y se lo pasa al método del DAO, siendo así su intermediario.

```
public void deleteScore(final Score score){
    //Tarea asíncrona
    new AsyncTask<Void, Void, Void>() {
        @Override
        protected Void doInBackground(Void... voids) {
            //Elimina un objeto Score
            scoreDao.deleteScore(score);
            return null;
        }
    //Ejecutamos la tarea en un sólo thread
    }.execute();
}
```

Este método, muy similiar al anterior, elimina todas las puntuaciones:

```
public void deleteAllScores(final List<Score> scoreList){
    //Tarea corta en segundo plano
    new AsyncTask<Void, Void, Void>() {
        @Override
        protected Void doInBackground(Void... voids) {
            //Elimina toda la lista de puntuaciones
            scoreDao.deleteAllScores(scoreList);
            return null;
        }
    //Ejecutamos la tarea en un sólo thread
    }.execute();
}
```

Para insertar una nueva puntuación se usa el siguiente método. En este caso, retorna el id de la instancia insertada, que luego se utilizará en el código del "View Controller". El retorno se hace de forma encapsulada en un tipo MutableLiveData. Esta clase, junto con LiveData, permiten el uso de Observers, objetos que notifican y retornan de forma asíncrona los resultados de una tarea.

```
public MutableLiveData<Long> insertScore(final Score score){
    final MutableLiveData<Long> id = new MutableLiveData<>();
    new AsyncTask<Void, Void, Void>() {
        @Override
        protected Void doInBackground(Void... voids) {
            id.postValue(scoreDao.insertScore(score));
            return null;
        }
    }.execute();
    return id;
}
```

Vamos a echar un vistazo a la parte del código de la actividad (el "view controller" dentro del patrón que estamos utilizando). El siguiente extracto de código se ejecuta cuando el jugador pulsa el botón de guardar puntuación. Se instancia un objeto de tipo Score, se establecen atributos, y se llama al método insertScore() del *viewmodel*.

```
Score score = new Score();
score.setUser("" + nameEdit.getText());
score.setScore(mScore);

scoreViewModel.insertScore(score).observe(ResultActivity.this, new Observer<Long>()
{
    @Override
    public void onChanged(@Nullable Long aLong) {
        Intent scoreIntent = new Intent(ResultActivity.this,ScoreActivity.class);
        scoreIntent.putExtra("scoreId",aLong);
        startActivity(scoreIntent);
    }
});
```

Es importante remarcar el método .observe() que se ejecuta de forma encadenada. Con este método, avisamos al *viewmodel* de la actividad (contexto) en la que nos encontramos, y le pasamos una implementación de la interfaz Observer.

El *viewmodel* llamará al método onChanged() de esta interfaz en cuanto la ejecución del INSERT haya acabado, pasando por parámetro el id del Score insertado. Con ese dato, y dentro de ese método, se ejecuta el cambio de actividad (al ránking de puntuaciones). Pasamos por el *intent* el id recibido con el propósito de remarcarlo en la lista que se muestra en el ránking.

9.1.4 Ayuda y acerca de

A la hora de desarrollar una aplicación, se debe tener en cuenta facilitar su uso lo máximo posible, con el objetivo de mejorar la experiencia del usuario y hacerla más accesible.

ThinkItApp cuenta con una sección de ayuda, en la que se puede acceder fácilmente desde el menú principal. Ésta sección contiene toda la información sobre la aplicación, además también incluye la posibilidad de repetir el tutorial que sólo aparece cuando se ejecuta la aplicación por primera vez.

Para organizar de forma clara y estructurada los apartados de ayuda, se ha creado una lista de elementos que se despliegan cuando el usuario pulsa la sección de ayuda deseada. Una vez pulsada, el resto de secciones se cerrarán, para mejorar la lectura de la sección abierta.

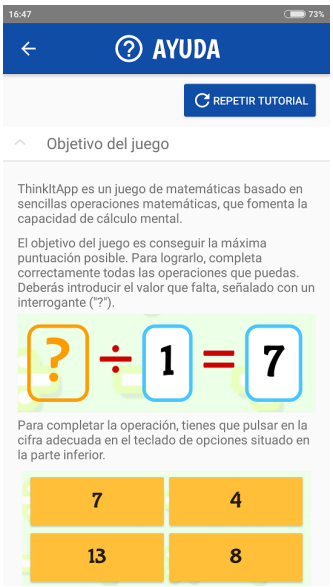


Figura 16: Pantalla de ayuda - Ejemplo de un tema expandido

Cada sección incluye su propio contenido y su propio diseño, cada una es independiente a las demás. En cada sección se explica mediante texto, imágenes y botones de muestra, información de la aplicación, separada en 7 secciones:

- **Créditos:** Contiene información de los desarrolladores y de la aplicación. También incluye el copyright. Es la sección que se abre por defecto.
- **Menú principal:** En esta sección se explican con una breve descripción, cada botón del menú principal y su funcionalidad.
- **Objetivo del juego:** Describe detalladamente el objetivo del juego, así como la dificultad progresiva y cuándo finaliza.
- **Ayudas y bonificaciones:** En este apartado se muestran las ayudas y bonificaciones del juego, así como consejos para obtener la puntuación máxima posible.
- **Ranking de puntuaciones:** Contiene información para el usuario sobre el ranking de puntuaciones.
- **Cambiar idioma:** Explica cómo cambiar el idioma de la aplicación.
- **Atribuciones:** Incluye todas las atribuciones por la utilización de obras de las que los desarrolladores de ThinkItApp no son autores.

Para realizar esta lista desplegable que se expande y se reduce, se han utilizado diferentes clases que Android proporciona. Cuando queremos tener una gran cantidad de datos agrupados en grupos o categorias, **ExpandableListView** es la opción ideal. Para que este tipo de elementos funcione, se requiere un **ExpandableListAdapter**.

ExpandableListAdapter

Este adaptador, que extiende de **BaseExpandableListAdapter**, tiene la finalidad de proporcionar datos a ExpandableListView de forma rápida. Al ser una interface, a partir de los métodos que proporciona se obtienen los elementos para luego ser mostrados en ExpandableListView, y también prepara los layout de cada elemento.

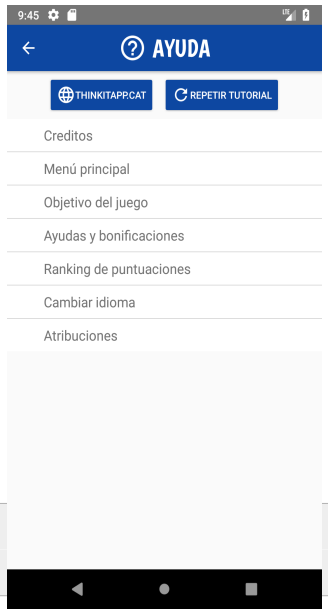


Figura 17: Pantalla de ayuda - Lista expandible cerrada

Antes de generar el adapter, debemos preparar los datos que servirán de fuente, esto se realiza desde la clase Java de la actividad de Android. Cada elemento dispone de dos niveles, el *header* (título) y su *child* (hijo). Estos elementos se almacenan en una lista y un mapa de hash, respectivamente.

```
List<String> listDataHeader;
HashMap<String, List<String>> listDataChild;
```

El *header* es aquella cadena de texto dónde el usuario tocará para abrir su contenido, mientras que el mapa *child* incluye el elemento header como clave y una lista de cadenas de texto que va a contener el child.

```
listDataChild.put(listDataHeader.get(0), creditsHelp);
listDataChild.put(listDataHeader.get(1), menuHelp);
listDataChild.put(listDataHeader.get(2), goalsHelp);
listDataChild.put(listDataHeader.get(3), bonusHelp);
[...]
```

En nuestro caso, los Strings que hemos establecido como datos del HashMap son Strings vacíos, puesto que no utilizaremos este sistema para poblar la lista, sino que utilizaremos recursos XML individuales para cada item. Esto se explica a continuación.

```
List<String> creditsHelp = new ArrayList<>();
creditsHelp.add("");
List<String> menuHelp = new ArrayList<>();
menuHelp.add("");
[...]
```

Una vez preparados los datos, se crea el *adapter* pasando por parámetro estos atributos, y se establece este *adapter* como *adapter* de nuestro ExpandableListView.

```
listAdapter = new ExpandableListAdapter(this, listDataHeader, listDataChild);
expListView.setAdapter(listAdapter);
```

En la definición de nuestro ExpandableListAdapter, se preparan las vistas de cada child para cada uno de los elementos de la lista. Como se puede observar, retornamos en cada caso una vista inflada de elementos XML predefinidos. Esto nos permite diseñar cada apartado desde el diseñador, añadiendo tanto texto como imágenes u otro tipo de vistas, en vez de simplemente Strings.

```
@Override
public View getChildView(int element, final int childPosition, boolean isLastChild,
View convertView, ViewGroup parent) {
    switch (element){
        case 0:
            LayoutInflater infalInflater = (LayoutInflater) this._context
                    .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
            convertView = infalInflater.inflate(R.layout.help_item_0, null);
            break;
        case 1:
            infalInflater = (LayoutInflater) this._context
                    .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
            convertView = infalInflater.inflate(R.layout.help_item_1, null);
            break;
[...]
```

ExpandableListView

En la instancia de ExpandableListView, a parte de indicarle el adapter a utilizar, podemos personalizar algunos comportamientos, como por ejemplo definir qué ocurrirá cuando se expanda un grupo de la lista con el método setOnGroupExpandListener().

Aprovechando este método, y una variable de clase que almacene el último elemento abierto, se puede conseguir que el elemento anteriormente abierto se cierre al expandir otro. Para ello, necesitamos un listener que esté atento a este evento:

```
expListView.setOnGroupExpandListener(new ExpandableListView.OnGroupExpandListener()
{
    @Override
    public void onGroupExpand(int groupPosition) {
        if (lastPosition != -1 && groupPosition != lastPosition) {
                expListView.collapseGroup(lastPosition);
         }
         lastPosition = groupPosition;
         expListView.smoothScrollToPosition(groupPosition);
     }
  });
}
```

Para mostrar la lista, hemos de añadir la vista al fichero XML que define la interfaz de la actividad:

```
<ExpandableListView
    android:id="@+id/lvExp"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

9.1.5 Modificador de sonido

El menú principal también incluye un FAB para activar o desactivar el sonido en toda la aplicación.

Figura 18: FAB de modificación de sonido

Cada vez que se ejecuta la actividad MainActivity, se lee la preferencia de nombre "mute" (que nosotros mismos creamos[7]), y según si está en true o no, ejecutamos el método *setMute()* o *setUnmute()*.

```
if(settings.getBoolean("mute",true)) {
    setMute();
} else {
    setUnmute();
}
```

Para gestionar el comportamiento del botón, usamos el booleano 'activated' que incorporan todos los FABs. Cuando se hace clic en él, se comprueba si está activado o no. Si está activado, ejecutará *setMute()*, y en caso contrario, *setUnmute()*.

```
volumeFAB.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (volumeFAB.isActivated()) {
            setMute();
        } else {
            setUnmute();
        }
    }
});
```

Para entender lo que hacen esos dos métodos, vamos a ver el contenido de setMute(). Lo que hace es, a grandes rasgos, cambiar el volumen del MediaPlayer de esta actividad, establecer el FAB en 'no activado', cambiar el estilo del botón y guardar la preferencia. El método *setUnmute()* es idéntico pero a la inversa.

```
private void setMute() {
    // Establecer volumen a 0
    volume=0;
    musicPlayer.setVolume(0,0);

    // Cambiar estado del FAB a activado
    volumeFAB.setActivated(false);

    // Cambiar background tint programáticamente
    volumeFAB.setBackgroundTintList(ColorStateList.valueOf(
            getResources().getColor(R.color.color_grey_disabled)) );

    // Guardar en preferencias
    settings.edit().putBoolean("mute",true).apply();
}
```

7 - Las preferencias devuelven siempre false si se intentan leer y no existen. Esto es útil para las primeras veces que se ejecuta la aplicación.

En el caso de otras actividades, estas leerán la preferencia "mute" cuando se inicien, y ejecutarán sus propios métodos de activación/desactivación de sonido, para cada archivo de audio que reproduzcan. Es importante que este proceso se haga inmediatamente después de cargar la actividad, para evitar que la reproducción de ficheros emita algún sonido en caso de estar el parámetro "mute" activado.

9.2 Tutorial de iniciación

A pesar de que el juego cuenta con una mecánica muy simple, sí que cuenta con distintas funcionalidades que pueden modificarla en cualquier momento. Eso, sumado al factor tiempo, puede hacer que un usuario que nunca ha jugado a nuestra app se sienta abrumado la primera vez que arranque.

Por ello, la primera vez que se pulsa el botón de jugar, se le da la oportunidad al jugador de probar un tutorial de iniciación para conocer los conceptos básicos del juego. Un acceso a este tutorial también está disponible desde el apartado de **Ayuda**.

La interfaz del tutorial es exactamente la misma que la de la pantalla del juego, mientras que para implementar el paso a paso y la interacción con el usuario, se ha utilizado la librería **FancyShowCaseView**, desarrollada por **faruktoptas**.

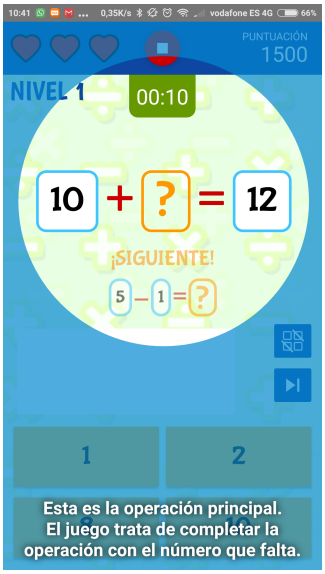


Figura 19: Pantalla del tutorial

Con la clase homónima FancyShowCaseView podemos crear pantallas donde aparece un texto, y un círculo engloba la parte de la interfaz que queramos enfocar especialmente, gracias al método focusOn(), al que le pasamos una vista inflada.

Como se puede observar, estos *showcases* son muy personalizables:

```
final FancyShowCaseView fancyShowCaseViewMainOperation = new
FancyShowCaseView.Builder(this)
        .title(getResources().getString(R.string.tutorial_main_operation))
        .titleStyle(R.style.tutorial_title_style,Gravity.BOTTOM | Gravity.CENTER)
        .focusOn(op1)
        .backgroundColor(getResources().getColor(R.color.color_showcase))
        .enterAnimation(animation)
        .build();
```

Además, la clase FancyShowCaseQueue nos permite crear colas de showcases, en las que, cuando el usuario pulse la pantalla y desaparezca, aparezca inmediatamente el siguiente.

```
mQueue = new FancyShowCaseQueue()
        .add(fancyShowCaseViewWelcome)
        .add(fancyShowCaseViewWelcome2)
        .add(fancyShowCaseViewMainOperation)
        [...]
mQueue.show();
```

9.3 El juego

El objetivo principal de ThinkItApp es conseguir la máxima puntuación a medida que se completan las operaciones matemáticas que el juego ofrece al jugador. Esta operación puede ser una **suma**, una **resta**, una **división** o una **multiplicación**, y existe la posibilidad de que se tenga que completar el **primer operando**, el **segundo operando** o bien el **resultado**.

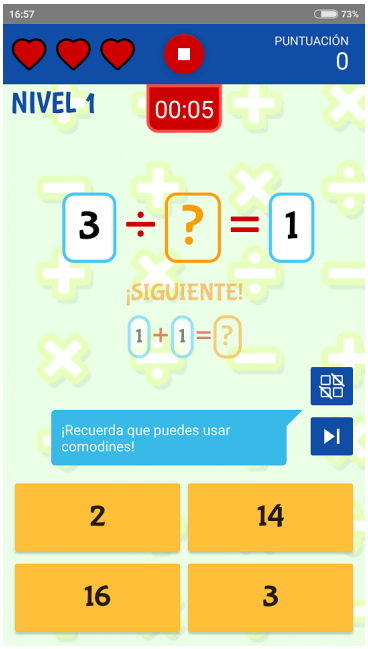


Figura 20: Pantalla principal del juego (nivel 1)

Tanto el tipo de operación como el campo a completar se decide de forma aleatoria en el momento de la carga de la operación. La operación en sí también se genera de forma aleatoria, aunque en este caso entran en juego otros factores como los **niveles**.

Cada 10 preguntas acertadas, se sube de nivel. Esto implicará que los rangos de números a partir de los cuales se calculan los operandos, cada vez serán más grandes, y por tanto las operaciones más difíciles.

Para completar la operación, se le ofrece al usuario una serie de posibles opciones, siendo todas ellas cercanas a la respuesta correcta, excepto, claro está, la respuesta correcta. Una vez el jugador pulse una de las opciones, se informará si ha sido correcta o no, y esto se reflejará en la puntuación. En ese momento cargará la siguiente operación y el jugador tendrá que responderla.

Siendo esta la mecánica principal del juego, a continuación se describen de forma detallada las funcionalidades secundarias con las que se intenta añadir cierta diversificación y escalabilidad.

Figura 21: Logotipo de ThinkItApp

9.3.1 Generación de operaciones y jugabilidad escalable

Para gestionar las operaciones aleatorias en tiempo de ejecución, se ha definido una clase **Operation**, que tiene los siguientes atributos:

- Primer operando
- Segundo operando
- Tipo de operación
- Resultado
- Un entero que determina el campo a esconder (y que el usuario tendrá que completar)

```
public class Operation {
    public enum OpType {
        SUM,SUBST,DIV,MUL
    }
    private int op1;
    private int op2;
    private OpType opType;
    private int res;
    private int hiddenField;
    public Operation(int op1, int op2) {
        this.op1 = op1;
        this.op2 = op2;
    }
    public Operation() {
    }

[···]
```

En **MathsActivity.java**, la clase principal de la actividad del juego, y que define la lógica tras el juego, disponemos de un método llamado **loadOperations()**, en la que se llama una o dos veces al método **calculateOperation()**, que devuelve un objeto de tipo Operation. Como se puede ver, en la primera pregunta se calculan dos operaciones, mientras que en sucesivas preguntas, la segunda operación pasa a ser la primera, y se calcula una nueva operación para la segunda.

```
if (firstTime) {
    [...] // Se omiten métodos de cambio de estilos para simplificar la explicación
    op1 = calculateOperation();
    op2 = calculateOperation();
} else {
    op1 = op2;
    op2 = calculateOperation();
}
```

¿Por qué se calculan dos operaciones? Pues esto es debido a que, en todo momento, se muestra al jugador dos operaciones: la operación principal que tiene que responder en el momento actual, y la siguiente operación que aparecerá una vez responda la actual.

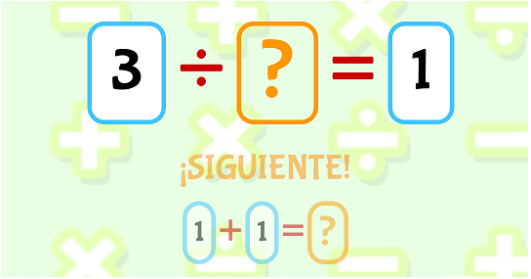


Figura 22: Operación actual (arriba) y operación siguiente (abajo)

Esto permite al jugador anticiparse al cálculo de la siguiente operación, aprovechando esos segundos entre que responde y aparece la siguiente, con tal de ganar algo de tiempo. El factor tiempo se explicará en el siguiente apartado.

Volviendo al código, vamos a ver cómo funciona el método **calculateOperation().** Primero, seleccionamos el tipo de operación. Esto se hace con un *random* que vaya del 0 al 3 (no se incluye el 4). El setter funcionará bien, ya que los valores de un enum corresponden internamente a números enteros.

```
//Seleccionar operación
operation.setOpType(Operation.OpType.values()[(int) (Math.random() * 4)]);
```

Seguidamente calculamos los dos operandos. Esta parte no es trivial, ya que es donde debemos ajustar cuidadosamente la dificultad de las operaciones, teniendo en cuenta que un mismo rango de números puede suponer una dificultad distinta dependiendo de si se trata de una división o de una suma (por ejemplo).

En este caso, se ha decidido controlar manualmente los 5 primeros niveles, y apartir de entonces aplicar valores incrementales diferentes a los rangos, dependiendo del tipo de operación. Un ejemplo del ajuste manual de los primeros niveles:

```
switch (level) {
    case 1:
        if (operation.getOpTypeStr().equals("÷")) {
            operand1Range = (10 - 1) + 1;
            operand2Range = (10 - 1) + 1;
        } else {
            operand1Range = (5 - 1) + 1;
            operand2Range = (5 - 1) + 1;
        }
        break;

[···]
```

En este caso, se aumenta el rango de números a 1-10 en las divisiones (a diferencia del resto de operaciones donde el rango es 1-5), para evitar la repetición de divisiones extremadamente fáciles, como 2x1 o 3x2.

Sin embargo, al llegar al nivel 5 se hace justamente lo contrario: si se trata de una multiplicación o una división, los rangos serán menores a los de las sumas y las restas. Es más, en el primer caso, se establece un rango más pequeño en el segundo operando respecto al primero. Esto se hace para evitar operaciones difíciles como 40/35 o 38x26, lo cual no es problema en caso de suma o resta: 40+35 o 38-26 son operaciones fáciles de calcular al vuelo.

```
case 5:
    if (operation.getOpTypeStr().equals("x") ||
operation.getOpTypeStr().equals("÷")) {
        operand1Range = (40 - 1) + 1;
        operand2Range = (20 - 1) + 1;
    } else {
        operand1Range = (50 - 1) + 1;
        operand2Range = (50 - 1) + 1;
    }
    break;
```

En el default del switch definimos el comportamiento escalable en los niveles posteriores. Para el incremento del rango, utilizamos cuatro variables acumuladoras: dos para los dos operandos en sumas y restas, y otras dos para los dos operandos en multiplicaciones y divisiones.

Cada una de ellas se incrementa progresivamente en un determinado valor (en este momento se incrementa en 10 en todos los casos, excepto en el operando 2 de multiplicaciones y divisiones, donde se incrementa en 5). El código está estructurado de manera que cada caso se pueda ajustar a gusto del desarrolador.

```
default:
    if (operation.getOpTypeStr().equals("x") ||
operation.getOpTypeStr().equals("÷")) {
        operand1Range = (scalableRangeOperand1MultDiv - 1) + 1;
        scalableRangeOperand1MultDiv +=10;
    } else {
        operand1Range = (scalableRangeOperand1SumSubst - 1) + 1;
        scalableRangeOperand1SumSubst +=10;
    }
    if (operation.getOpTypeStr().equals("x") ||
operation.getOpTypeStr().equals("÷")) {
        operand2Range = (scalableRangeOperand2MultDiv - 1) + 1;
        scalableRangeOperand2MultDiv +=5;
    } else {
        operand2Range = (scalableRangeOperand2SumSubst +10 - 1) + 1;
        scalableRangeOperand2SumSubst +=10;
    }
    break;
```

Estas variables tienen un valor inicial en la instanciación de la clase.

```
private int scalableRangeOperand1SumSubst = 50;
private int scalableRangeOperand2SumSubst = 50;
private int scalableRangeOperand1MultDiv = 40;
private int scalableRangeOperand2MultDiv = 20;
```

Para que el juego vaya subiendo de nivel, se añade un *if* para que el atributo de clase *level* aumente en uno si el número de respuestas correctas hasta el momento es **múltiplo de 10 (cada 10 preguntas)**, si además no es la primera pregunta (cuando ésta carga) y si la última respuesta dada por el jugador fue correcta (para evitar que, tras pasar de nivel, siga subiendo de nivel al fallar en la siguiente pregunta, donde las anteriores condiciones seguirían siendo ciertas).

```
if (mCorrectAnswers %10==0 && !isFirstAnswer && mAnswerWasCorrect) {
    level++;
    delay=2000L;
    if (level == 2 || level == 3) {
        mEnabledButtons+=2;
    }
```

Otro punto que añade escalabilidad al juego, aunque en este caso limitada, es el hecho de que el teclado de posibles respuestas va aumentando su tamaño a medida que se incrementa el nivel. Puede albergar hasta 8 respuestas, en el momento en que se alcanza el nivel 3, aumentando de 2 en 2 desde el nivel 1.

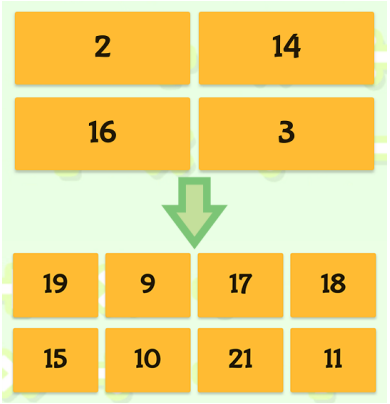


Figura 23: Teclado de opciones. Arriba, nivel 1; abajo, nivel 3

Esto es posible si separamos en el código el hecho de que unos botones sean visibles o no, del proceso de rellenar los botones con posibles respuestas.

El control de visibilidad se realiza mediante un simple *switch* para los tres primeros niveles (teniendo en cuenta que la visibilidad por defecto es GONE (invisible y sin ocupar espacio).

Nótese que answerButtons es un ArrayList que agrupa todos los botones del teclado, y la posición va de izquierda a derecha, de arriba a abajo, haciendo zig-zag. Este posicionamiento en el array es clave para su escalabilidad. Para empezar, como van cambiando su visibilidad de 2 en 2, permite mantener una consistencia visual sin demasiados esfuerzos.

```
switch (level) {
    case 1:
        answerButtons.get(0).setVisibility(View.VISIBLE);
        answerButtons.get(1).setVisibility(View.VISIBLE);
        answerButtons.get(2).setVisibility(View.VISIBLE);
        answerButtons.get(3).setVisibility(View.VISIBLE);
        break;
    case 2:
        answerButtons.get(4).setVisibility(View.VISIBLE);
        answerButtons.get(5).setVisibility(View.VISIBLE);
        break;
    case 3 :
        answerButtons.get(6).setVisibility(View.VISIBLE);
        answerButtons.get(7).setVisibility(View.VISIBLE);
        break;
    default:
}
```

Figura 24: Posiciones de los botones del teclado en el ArrayList

Tras esto, se decide aleatoriamente una posición válida del teclado para albergar la respuesta correcta. Obviamente no puede ser un botón que no sea visible en ese momento. Por ello la variable de clase mEnabledButtons almacena el número de botones activados en un momento dado.

```
mCorrectButtonIndex = (int) (Math.random() * mEnabledButtons);
```

Posteriormente, obtenemos la respuesta correcta de la operación calculada, dependiendo de cuál sea su campo oculto, y **establecemos ese número como texto** del botón elegido anteriormente.

```
int answer=0;
switch (op1.getHiddenField()) {
    case 0:
        answer=op1.getOp1();
        break;
    case 1:
        answer=op1.getOp2();
        break;
    case 2:
        answer=op1.getRes();
        break;
}

answerButtons.get(mCorrectButtonIndex).setText(answer + "");
```

Una vez se ha establecido la respuesta correcta, se debe **poblar el resto de botones**. Para ello, primero se genera un rango de respuestas posibles (incorrectas). Por lo general, las posibles respuestas oscilarán entre *respuesta + 10* y *respuesta – 10*.

No obstante, y para evitar que aparezcan números negativos y el 0, hacemos que cuando la respuesta correcta sea igual o inferior a 10, establecemos manualmente el mínimo del rango al 1, y el máximo a respuesta+10 sumándole la diferencia entre 10 y la respuesta, obteniendo así el mismo número de respuestas que un número > 10, pero sin 0 ni negativos.

```
if (answer <= 10) {
    min=1;
    max=answer+10+(10-answer);
} else {
    min=answer-10;
    max=answer+10;
}

int answerRange = (max - min) + 1;
```

Finalmente, se recorren los botones activos, calculando en cada iteración un número dentro del rango calculado (aplicando un filtro para que no lo haga si el botón ya tiene una respuesta, para ello es necesario limpiar los *strings* antes). Después, se recorren todos los botones anteriores a la iteración actual, con tal de comprobar que el número generado no es idéntico a uno ya colocado. En ese caso, o en caso de que el número generado sea la respuesta correcta, vuelve a calcular otro número y a realizar el mismo proceso. Así hasta que todos los botones estén poblados con números diferentes.

```
//Recorrer botones activos
for (int i = 0; i < mEnabledButtons; i++) {
    //Solo aplica si está vacío todavía
    if (answerButtons.get(i).getText()=="") {
        do {
            //Calcular respuesta errónea aleatoria
            option = ((int)(Math.random() * answerRange) + min) + "";
            found = false;
            //Comprobar si en los anteriores ya puestos se repite el número random
            for (int j = 0; j < i; j++) {
                if (answerButtons.get(j).getText().toString().equals(option)) {
                    found=true;
                    break;
                }
            }
        //Si está repetido o corresponde a la respuesta correcta, volver a calcular
        } while (found ||
option.equals(answerButtons.get(mCorrectButtonIndex).getText().toString()));
        answerButtons.get(i).setText(option);
    }
}
```

Tras generar ambas operaciones y rellenar los botones, el usuario puede pulsar en uno de ellos para responder. No sin antes establecer un listener para el evento de *click*. Esto es fácilmente realizable si directamente se implementa la interfaz View.OnClickListener en la clase, y se sobrescribe el método onClick().

En este punto, se comprueba si la vista en la que se ha hecho clic se encuentra en un índice de la lista de botones igual al índice donde se encuentra la respuesta correcta (variable mCorrectButtonIndex que calculamos anteriormente). Si coinciden, es respuesta correcta, si no, es incorrecta. En tal caso, se hace lo oportuno (aumentar variables, reproducir sonidos, desactivar botones…) hasta que en un momento dado se vuelve a llamar al método **loadOperations()**, y todo vuelve a comenzar otra vez (eso sí, con distintos niveles, rangos, u otros parámetros de dificultad progresiva).

Nótese que se desactiva el listener para evitar dobles clics. Esto nos obliga a realizar un *view.setOnClickListener(this)* al recargar los botones de opciones.

```
//ACIERTA
if (answerButtons.indexOf(view) == mCorrectButtonIndex) {
    correctAnswer();
} else { //FALLA
    [...]
    view.setOnClickListener(null);
    [...]
    incorrectAnswerOrTimeOut();
}
```

9.3.2 Bonificaciones y ayudas

Para añadir un punto de extensibilidad al juego y una motivación extra al jugador, se incluyen diferentes ayudas y bonificaciones que entran en juego durante el trascurso de la partida.

Puntuación

Para empezar, el juego cuenta con un sistema de puntuaciones, que es en lo que se basa el objetivo a largo plazo: conseguir la máxima puntuación posible. El jugador empieza con 0 puntos, y a medida que completa correcta o incorrectamente una operación, se suman o restan puntos a la puntuación. Más específicamente, se suman **100 puntos** por cada respuesta correcta, y se restan **50 puntos** por respuesta incorrecta (con el límite de 0).

Sin embargo, la puntuación también puede cambiar mediante otro factor: la cuenta atrás. El juego dispone de una cuenta atrás, y el jugador debe procurar que esta nunca llegue a 0 (se explica en detalle en el siguiente punto). Por cada respuesta correcta, se añade una bonificación de puntos que sigue esta fórmula:

```
puntuacionExtra = segundosRestantes * 10 + 10;
```

Esto permite que las puntuaciones finales no sean muy parecidas o sean siempre números redondos. Cuando se responde correctamente, se muestra al jugador durante unos breves segundos la puntuación conseguida, diferenciando qué puntos fueron provinientes de la bonificación por tiempo restante.

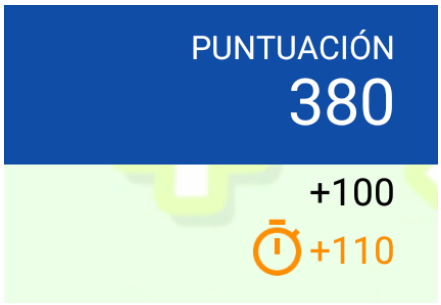


Figura 25: Puntos añadidos diferenciados según su origen

Tiempo extra

Se han añadido algunas bonificaciones que ofrecen al jugador recompensas que le permiten alargar la duración de su partida: Cada vez que el jugador complete una operación correctamente, se evalúa si lo ha hecho en menos de 10 segundos desde que apareció la operación en la zona principal. En caso afirmativo, se añaden segundos extra a la cuenta atrás, siguiendo la siguiente fórmula:

```
tiempoExtra = (10 - tiempoTranscurrido) / 2;
```

Es importante mencionar que hay un límite de 59 segundos en el tiempo acumulado. No queremos que sea excesivamente fácil acumular tiempo y que el juego sea un “paseo".

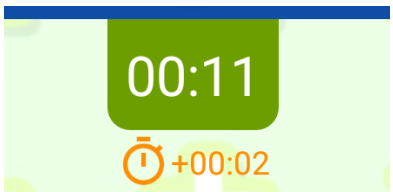


Figura 26: Tiempo extra mostrado bajo el contador

En relación a la interfaz, se utiliza el mismo estilo que en las puntuaciones para remarcar que se trata de una bonificación surgida por el tiempo.

Comodines

Por último, el jugador tiene la posibilidad de usar hasta 2 comodines, en caso de que se vea en apuros para contestar una operación, ya sea por la dificultad o por falta de tiempo. Son los siguientes:

- **Comodín del 50%:** Este comodín reduce a la mitad las posibles respuestas ofrecidas al jugador, lo cual aumenta las posibilidades de acertar. Eso sí, a medida que el número de opciones sea mayor, menos efectividad tendrá este comodín (con 4 opciones quedarán 2, pero con 8 opciones restarán 4). Las respuestas que se ocultarán se deciden de manera aleatoria en el momento de utilizar el comodín. Nótese que la cuenta atrás no se verá afectada, así que el tiempo para pensar será el mismo.

Figura 27: Botón del comodín del 50%

Figura 28: Teclado de opciones tras aplicar el comodín del 50%

- **Comodín de omitir operación:** Este otro comodín es un verdadero as en la manga, ya que permite al jugador omitir completamente la operación actual, y pasar a la siguiente como si la actual hubiera sido respondida correctamente. El jugador deberá meditar si guardarse o no este comodín para preguntas difíciles.

Figura 29: Botón del comodín de omitir operación

Los comodines solo pueden ser utilizados una vez por partida, y en el momento en que se utilizan, se deshabilitan y se marcan como tal.

Figura 30: Comodines agotados

9.3.3 Obstáculos y fin de partida

Por supuesto, no todo iban a ser ayudas y bonificaciones. Para añadirle un punto de reto y emoción a la mecánica, hay determinados obstáculos de los que el usuario debe ser consciente y evitar que arruinen su partida. Se trata de **la cuenta atrás y las vidas.**

Cuenta atrás

El factor tiempo siempre ha sido un elemento recurrente en los videojuegos, y ThinkItApp no es una excepción. La barra superior del juego alberga una cuenta atrás, que se convertirá en el mayor enemigo del jugador, ya que éste deberá procurar que nunca llegue a cero.

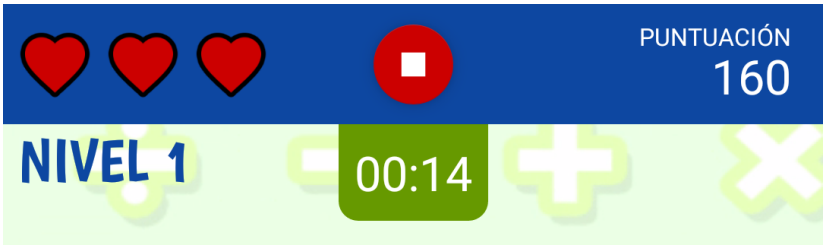


Figura 31: Barra superior: vidas, nivel, cuenta atrás, puntuación y botón de parar

Cuando el juego comienza, el contador arranca con 10 segundos, como es de esperar, irá decrementando a cada segundo. Si se responde correctamente, cabe la posibilidad de que se sumen algunos segundos (como se ha explicado anteriormente), o en caso de no haber bonificación, se mantendrán los segundos para la siguiente operación. Esto último ocurrirá también en caso de fallar una operación.

Independientemente de si se falla o no, si no hay ningún tipo de bonificación, y se respondió cuando quedaban menos de 10 segundos, se sumarán 10 segundos (a los restantes) para la siguiente operación. Al ocurrir solamente cuando se responde con menos de 10 segundos, esto no se puede considerar una bonificación (no es acumulativo realmente) sino por cortesía para que pueda continuar con el juego… si es capaz de estar a la altura de la dificultad.

¿Y qué ocurre cuando el contador llega a 0? Pues que el jugador pierde una **vida**. Además, el juego se ha concebido como una "carrera", en la que no hay forma de poner en pausa el juego, simplemente el jugador debe seguir hasta que fracase.

Solo hay un momento en que el juego puede ponerse en pausa (de forma automática), y es cuando la app pasa a segundo plano por haber abierto otra o haber vuelto al *launcher*. Con una **excepción**: el contador seguirá descontando en segundo plano, y cuando llegue a 0, se pondrá en pausa hasta que el jugador vuelva a la app, momento en que aparecerá un diálogo, que el jugador deberá aceptar para continuar el juego.

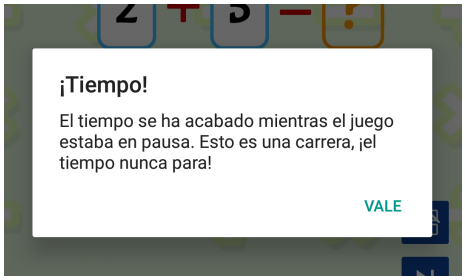


Figura 32: Diálogo cuando el contador llega a cero en segundo plano

Esto evita que haga trampas utilizando otra aplicación para calcular la operación. Aunque no podemos controlar que use otros dispositivos, reducimos las posibilidades de trampa.

Así y todo, el jugador tiene en todo momento a su disposición el botón de *stop*. Pero cuidado, no es una pausa. Esto desencadenará un *Game Over* como si hubiese perdido todas las vidas. Este botón se incluyó por el hecho de que los jugadores deben tener siempre el control sobre si salir o continuar en la app. Y en este caso, se le da la oportunidad de guardar la puntuación si quiere (en la pantalla de *Game Over* que se explicará en el siguiente apartado).

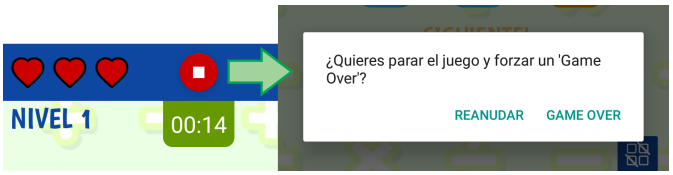


Figura 33: Utilización del botón de 'stop'

Vidas

Son la única razón por la que el jugador puede seguir completando operaciones aunque falle… y también son la razón por la que se acabará el juego. La partida empieza con 3 vidas, que aparecen en la parte superior izquierda de la pantalla, y hay dos maneras de perder una: agotando la cuenta atrás (como se ha comentado) o contestando erróneamente.

Figura 34: Las tres vidas iniciales

A diferencia de algunos juegos, éste no tiene un estado de "sin vidas". Es decir, se cuentan las vidas como oportunidades, no como salvavidas. De este modo, cuando quede 1 vida y el jugador falle, se habrá acabado el juego.

Las vidas no son recuperables, así que el usuario deberá pensárselo dos veces antes de responder sin tenerlas todas consigo. Para evitar el riesgo, puede usar un comodín (explicados anteriormente). Tras acabar con sus vidas o darse por rendido, un cartel anunciará el fin de la partida.

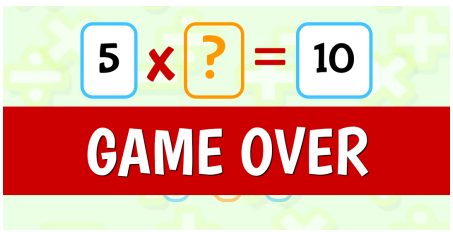


Figura 35: Un cartel anuncia el fin de la partida

9.4 Pantalla de 'Game Over' (fin de partida)

Al finalizar la partida, se le muestra al jugador la pantalla de *Game Over*, que contiene la puntuación final que ha conseguido y se le pregunta si quiere jugar de nuevo, salir sin guardar o guardar la puntuación. La pantalla dispone de tres botones de tipo **Floating Action Button** en la parte inferior con estos propósitos.

El primero de ellos permite al usuario **volver a jugar**. Al ser pulsado, preguntará al usuario mediante un diálogo de tipo AlertDialog, si desea salir de esta pantalla sin guardar la puntuación. Si el jugador lo rechaza, no se realizará ninguna acción (ya que para eso debe usar el botón de guardar puntuación), en caso contrario, iniciará otra partida.

Figura 36: Botón

El segundo botón cierra la pantalla de *Game Over* y **devuelve al usuario al menú principal**. De nuevo preguntará mediante un diálogo si desea salir sin guardar la puntuación.

Figura 37: Botón de salir sin guardar

El tercer y último botón, sirve para **guardar la puntuación**. En ese caso, será obligatorio que el jugador haya introducido su nombre en el campo de texto de la zona central (máximo 20 caracteres) para identificar al jugador que realizó esa puntuación a partir de su nombre en el ránking de puntuaciones guardadas.

Figura 38: Botón de guardar puntuación

Específicamente, se comprueba que la cadena de texto tiene una longitud de, al menos, 1 carácter. Además, se utiliza el método *trim()* para quitar espacios al principio y al final (y así, además de "limpiar" la cadena, evitamos un falso negativo en caso de que el texto solo contenga espacios).

En caso que esto no se cumpla, mostrará un error (utilizamos el método *setError()* del EditText) y no guardará la puntuación.

```
if (nameEdit.getText().toString().trim().equals("")) {
      nameEdit.setError(getResources()
            .getText(R.string.result_name_required));
}
```

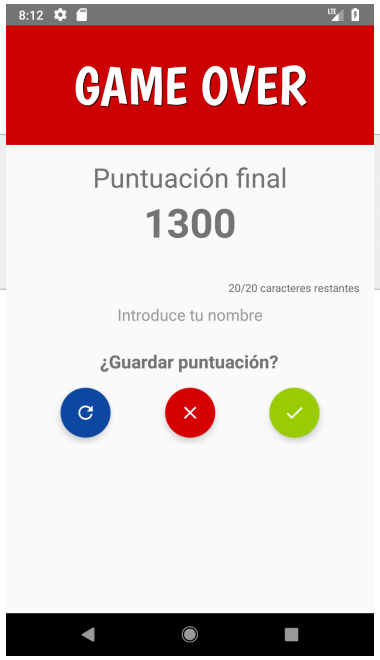


Figura 39: Pantalla de 'Game Over'

Si decide guardar la puntuación final, y el usuario ha introducido un nombre válido (1-20 caracteres), se insertará la puntuación en la base de datos, utilizando el método *insertScore()* del *viewmodel*, tal y como se ha explicado en la sección **"La base de datos"**.

Una vez guardada la puntuación, se mostrará el ránking con todas las puntuaciones guardadas anteriormente y la puntuación de la partida que justo ha terminado, con un color de fondo distinto al resto. (Véase: **"Ránking de puntuaciones"**).

A continuación se muestra un ejemplo de creación de un **AlertDialog**. Mediante *.setPositiveButton()* y *.setNegativeButton()* establecemos el comportamiento en cada uno de los dos botones, "Sí" o "No".

En el listener del botón de "Sí", utilizamos una variable *mode*, ya que utilizamos el mismo método *confirmExit()* tanto para cuando el usuario quiere volver a jugar (sin guardar) como si quiere salir sin guardar. Con esta variable sabemos desde dónde se llamó.

Dependiendo del botón pulsado, o bien finalizará la actividad actual (lo que hará volver al menú principal según la jerarquía de actividades[8]) o bien iniciará de nuevo la actividad del juego (además de finalizar también la actividad actual).

```
private void confirmExit(final int mode) {
    new AlertDialog.Builder(ResultActivity.this)
         .setMessage(R.string.dialog_result_exit_withoutsave)
         .setPositiveButton(R.string.dialog_yes, new
DialogInterface.OnClickListener() {
             public void onClick(DialogInterface dialog, int id) {
                 if (mode == MODE_EXIT) {
                     finish();
                 }
                 if (mode == MODE_REPLAY) {
                     Intent intent=new Intent(ResultActivity.this,
                             MathsActivity.class);
                     startActivity(intent);
                     finish();
                 }
             }
         })
         .setNegativeButton(R.string.dialog_no, new
DialogInterface.OnClickListener() {
             public void onClick(DialogInterface dialog, int id) {
                 //NOTHING
             }
         }).create().show();
     }
}
```

8 - Se pueden establecer jerarquías de actividades de Android para definir el flujo de navegación a través de ellas. De este modo, cuando se hace finish() en una actividad, se volverá a su actividad padre. Esto se define en el fichero AndroidManifest.xml

Con el objetivo de mejorar la accesibilidad y la comodidad del usuario a la hora de introducir su nombre, indicamos en todo momento la cantidad de caracteres restantes. Para conseguir esto, cada vez que el usuario modifique el texto (introduzca un carácter mientras escribe, por ejemplo), se calcula la resta entre la cantidad de caracteres permitidos y la cantidad de caracteres introducidos hasta ahora.

Esto es posible con la interfaz llamada **TextWatcher** utilizada en el método *addTextChangedListener()* del **EditText**. Los métodos de nuestro TextWatcher anónimo serán llamados cuando el usuario modifique el EditText:

```
nameEdit.addTextChangedListener(new TextWatcher() {
    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count) {
        // Ningún comportamiento establecido
    }
    @Override
    public void beforeTextChanged(CharSequence s, int start, int count, int aft){
        // Ningún comportamiento establecido
    }
    @Override
    public void afterTextChanged(Editable s)
    {
        // Esto muestra los caracteres restantes
        countTextView.setText((nameMaxLength - s.toString().length()) + "/" +
nameMaxLength + " " + getResources().getString(R.string.result_characters));
    }
});
```

Para controlar que el jugador introduzca un máximo de 20 caracteres, en el fichero XML correspondiente a la actividad ResultActivity.java, indicamos que la longitud máxima del EditText es de 20.

```
<EditText
    [...]
    android:maxLength="20"
    [...]
/>
```

También, hay que evitar que el usuario pueda introducir saltos de línea, ya que un nombre no contiene saltos de línea. Con estos atributos lo conseguimos:

```
<EditText
    [...]
    android:lines="1"
    android:maxLength="20"
    android:maxLines="1"
    android:singleLine="true"
    [...]
/>
```

9.5 Modo multijugador LAN

El juego dispone de un modo multijugador local (coloquialmente "LAN"), es decir, un modo competitivo contra otros jugadores que se encuentren en la misma red. Desde el ládo técnico, este modo se basa en dos fases: un sistema de creación-unión de salas de juego y un sistema de intercambio de mensajes basado en eventos.

La primera fase se basa principalmente en una **arquitectura cliente-servidor** mediante mensajes broadcast y unicast UDP, enviando y recibiendo palabras clave para el descubrimiento de servidores y la unión a ellos. En una fase intermedia, también se utiliza TCP con objetos, justo en el momento anterior de arrancar el juego. A partir de ese momento, se utiliza exclusivamente TCP con objetos, pero en este caso se diseña una **arquitectura P2P**, en la que todos los jugadores conocen las direcciones IP de todos los jugadores, y en el momento en que algún dato de la partida de un jugador cambie, el jugador avisará a todos los jugadores a la vez, que estarán también constantemente escuchando. En otras palabras, todos los jugadores son servidores y todos los jugadores son clientes.

Cabe mencionar que, tanto para TCP como para UDP, se ha decidido utilizar el puerto de escucha **15151**, que no es el puerto de ningún servicio conocido ni entra dentro del rango de "puertos bien conocidos" (0-1023). Así se evita la utilización de un puerto ya abierto por otro servicio.

Para hacer posible este modo, se ha diseñado una serie de actividades extra para la selección del modo, y para las funcionalidades necesarias para cada rol en el momento de la creación de la sala (unas para el que la crea, que llamaremos "servidor", otras para el que se une, que llamaremos "cliente").

- **Por parte de ambos (al inicio):**
  - **GameModeActivity** → Permite seleccionar el modo de juego (individual o multijugador).
  - **LocalMultiplayerActivity** → Permite decidir si crear una sala o unirse a una existente.

- **Por parte del servidor:**
  - **CreateRoomActivity** → En este punto el jugador debe introducir el nombre de la sala y su nombre de jugador.
  - **ServerWaitingRoomActivity** → Con la sala ya creada, aquí el servidor puede observar cómo los jugadores se van uniendo. El servidor espera a que todos estén listos, y acciona el botón de comenzar.

- **Por parte del cliente:**
  - **JoinRoomPlayerNameActivity** → El cliente debe introducir su nombre de jugador.
  - **JoinRoomActivity** → Se escanea la red en busca de salas, y se muestran. El jugador puede unirse a una.
  - **ClientWaitingRoomActivity** → El jugador deberá pulsar el botón de "Listo" para informar al servidor que está listo. Esperará hasta que el servidor arranque el juego.

En los diagramas de secuencia de las siguientes páginas se muestra un resumen esquemático del flujo de ejecución a través de los distintos *threads* que se crean en todo el proceso.

Mecánica

La mecánica del modo multijugador es prácticamente la misma que la del modo individual, con la diferencia de que mientras jugan, los jugadores pueden observar una lista en tiempo real con el estado del juego de cada uno de ellos:

- Posición
- Nombre
- Indicador de comodín del 50% utilizado
- Indicador de comodín omitir operación utilizado
- Puntuación
- Indicador de Game Over

Esto añade un factor competitivo que incentivará a los jugadores a intentar quedar en lo más alto de la tabla, pudiendo ver la puntuación del resto de jugadores. Así y todo, aquí no gana el más rápido, sino el más hábil, puesto que mientras no se pierdan todas las vidas, los jugadores pueden seguir respondiendo preguntas para acumular puntos.

Si pierde todas las vidas, el jugador se quedará esperando a que el resto de jugadores también "mueran". Será en ese momento en que se sabrá quién ha sido el ganador.

9.5.1 Diagramas

Diagrama 1: Gestión de salas

El siguiente diagrama muestra el flujo de comunicación entre threads de distintos dispositivos en el proceso de creación y unión a salas, siendo p1 el dipositivo (o jugador) que busca y se une a una sala (“cliente"), y p2 el que la crea (“servidor").

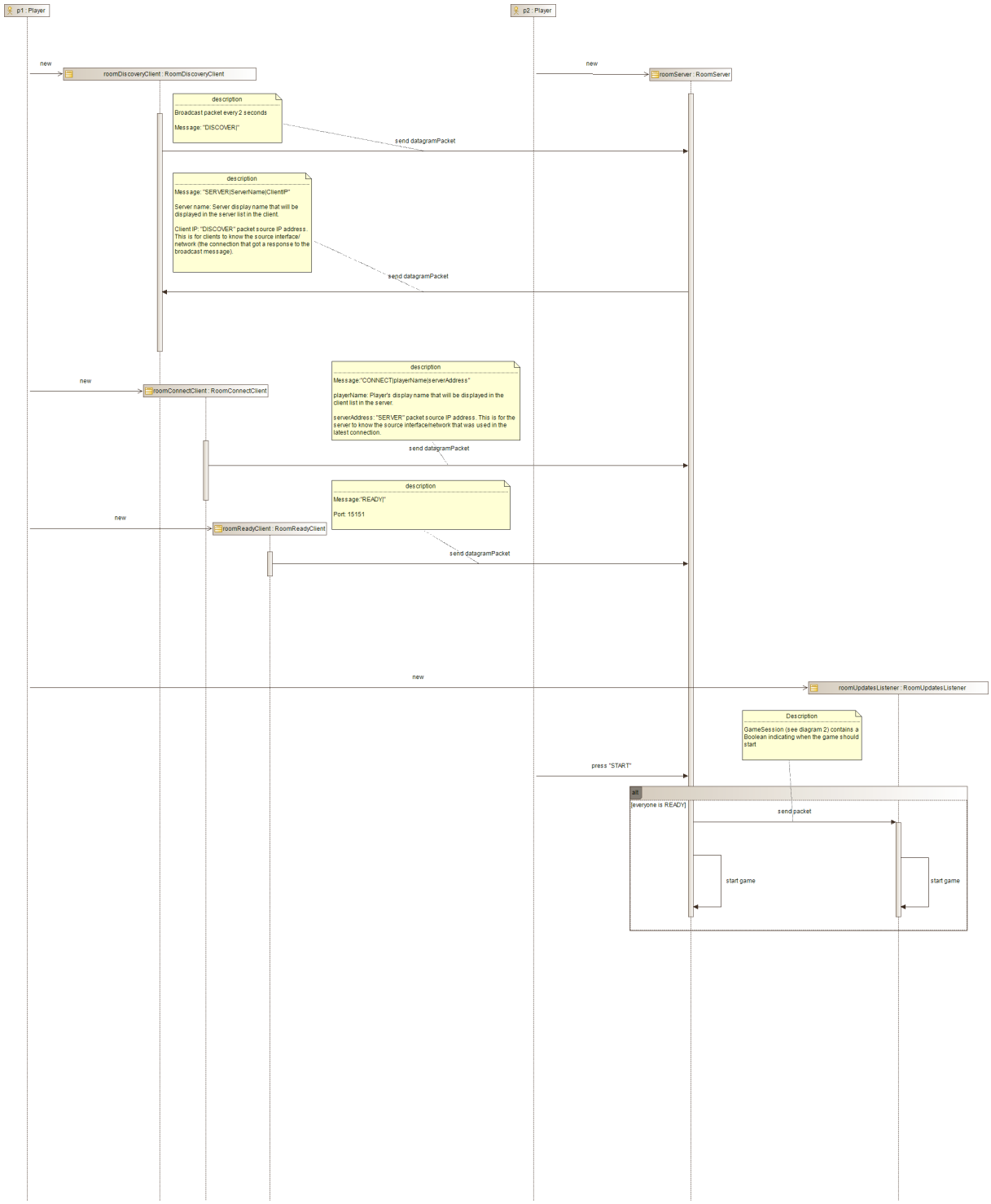


Figura 40: Diagrama de la gestión de salas

Diagrama 2: Comunicación *in-game*

El siguiente diagrama representa la etapa posterior al anterior diagrama: la comunicación entre jugadores, con el juego ya comenzado. Es una representación de la arquitectura P2P que hemos implementado.

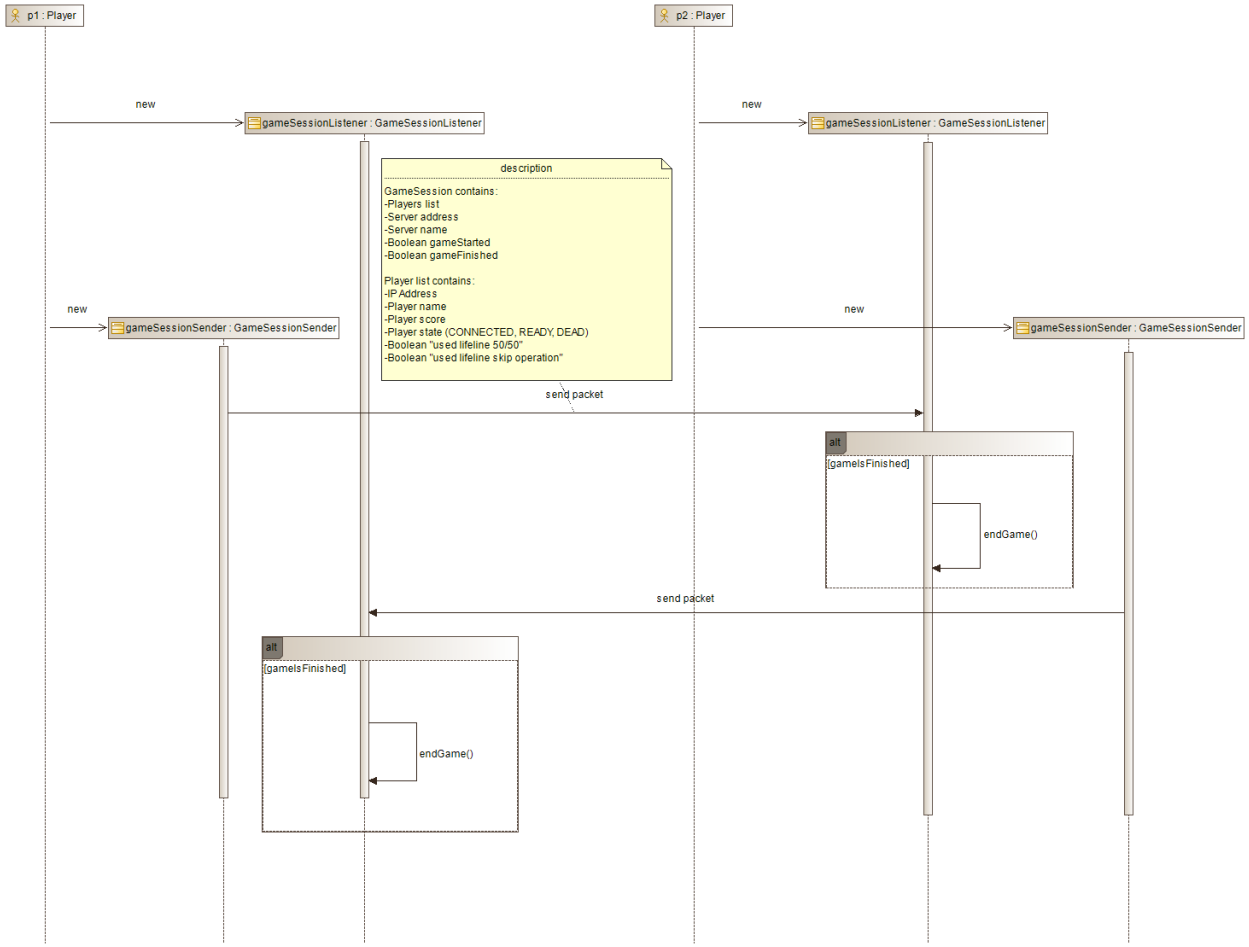


Figura 41: Diagrama de la comunicación in-game

Diagrama 3: Flujo de actividades del modo multijugador

Aquí se muestra la navegación por las diferentes pantallas del modo multijugador, con dos bifurcaciones dependiendo de si es quien crea o se une a la sala, y la "reunión" en la pantalla del juego.

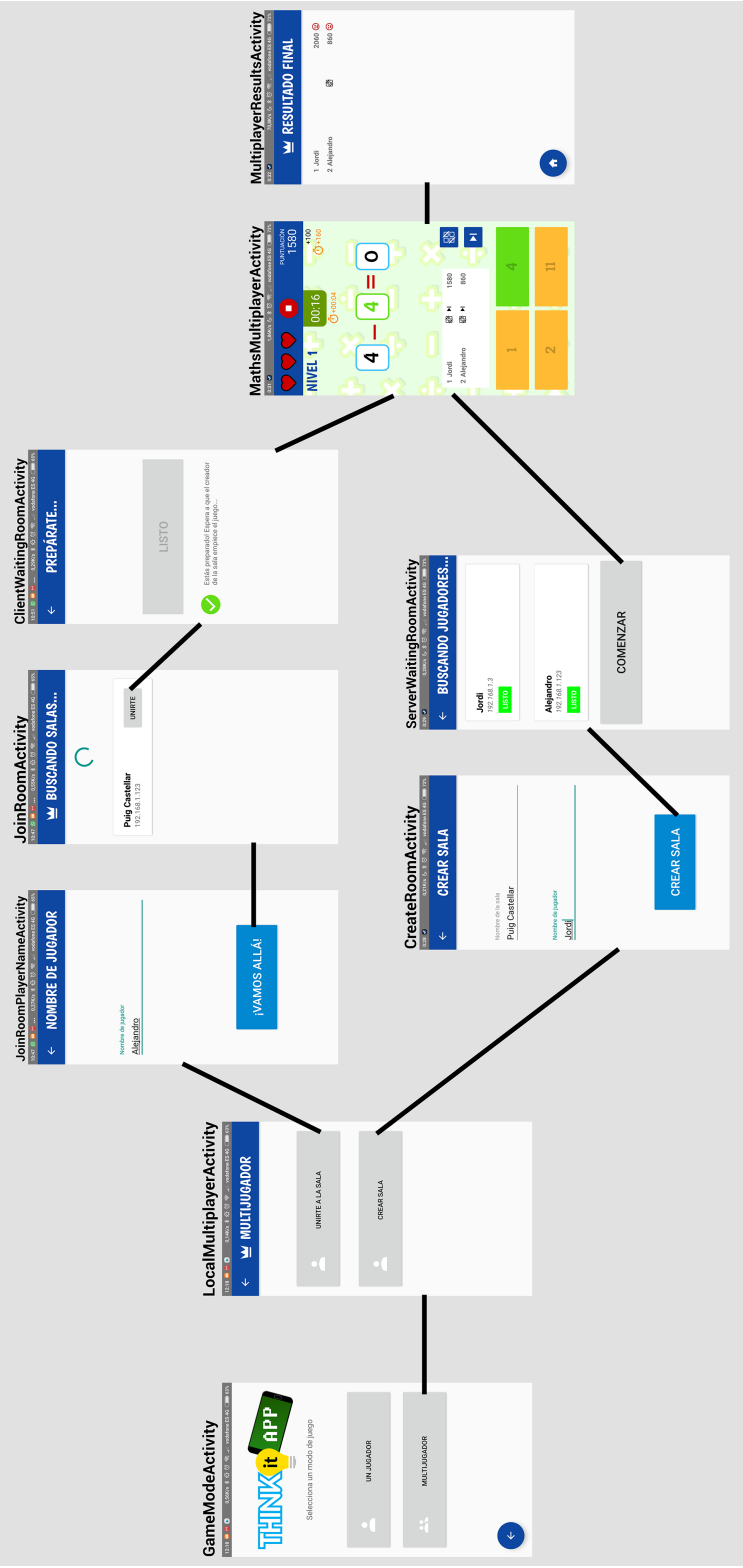


Figura 42: Flujo de actividades del modo multijugador

9.5.2 Explicación detallada

Creación de sala

Nos ponemos en el caso de que un jugador decide crear una sala de juego para poder jugar con otros jugadores. Accederá al submenú de modo de juego, seleccionará el modo multijugador, y en el siguiente menú accederá a "Crear sala".

A continuación, el juego le pedirá dos datos: el nombre que le pondrá a la sala (y que otros jugadores podrán ver cuando busquen salas) y el nombre o apodo que quiere él como jugador.

En el momento en el que pulse **"Crear sala"**, se ejecutará un *thread* llamado **RoomServer**, que se encargará de escuchar paquetes UDP de todas las interfaces. Dependiendo del contenido de los mensajes, se ejecutarán diferentes secuencias.

PAQUETES UDP	
MENSAJE	**DESCRIPCIÓN**
"DISCOVER"	Mensaje de tipo *broadcast* que "escanea" la red en busca de salas. El servidor responderá con un mensaje "SERVER".
"SERVER"	Respuesta por parte de un servidor a un mensaje de tipo "DISCOVER". Incluye información del servidor.
"CONNECT"	Mensaje del cliente al servidor, para conectarse a este después de descubrirlo. Incluye información del cliente. Lo acciona el jugador seleccionando la sala.
"READY"	Mensaje del cliente al servidor, tras unirse a la sala, para indicar que el jugador está listo. Lo acciona el jugador mediante el botón de "Listo".

Para escuchar este tipo de mensajes UDP, utilizamos las clases **DatagramSocket** y DatagramPacket de Java. Con **DatagramPacket** declaramos el paquete en el que se insertarán los datos, mientras que el método ***receive()*** de DatagramSocket se podrá a escuchar en el puerto que hayamos definido en su constructor y almacenará los datos recibidos en el DatagramPacket que le pasemos por parámetro.

```
public void listen() {
    byte [] socketData;
    try {
        serverSocket = new DatagramSocket(port);
        while(continueConnected) {
            socketData = new byte[1024];
            DatagramPacket packet = new DatagramPacket(socketData,1024);
            serverSocket.receive(packet);
[...]
```

Obtenemos el mensaje creando un nuevo String a partir del retorno del método *getData()* del paquete.

```
String fullMessage = new String(packet.getData(), 0, packet.getLength());
```

En esta fase de la conexión, los mensajes que enviamos son solamente cadenas de texto. Sin embargo, para separar la información que enviamos, se ha definido un delimitador, que en este caso es la tubería "|". Para tratar los mensajes según su tipo, utilizamos el método split() del String para separar los datos enviados en un array de Strings, y observamos el primer campo, que siempre será una palabra clave diferenciadora.

```
String[] parsedMessage = fullMessage.split("\\|");
if(parsedMessage[0].equalsIgnoreCase("DISCOVER")) {
      //Tratar DISCOVER
} else if (parsedMessage[0].equalsIgnoreCase("CONNECT")) {
      //Tratar CONNECT
} else if(parsedMessage[0].equalsIgnoreCase("READY")) {
      //Tratar READY
```

Independientemente del mensaje que hayamos recibido, lo que siempre podemos hacer es obtener la **dirección IP y el puerto de origen** del paquete, algo que ofrece la clase DatagramPacket.

```
InetAddress clientIp = packet.getAddress();
int clientPort = packet.getPort();
```

Paquete DISCOVER

Cuando el servidor recibe este paquete, lo único que tiene que hacer es responder con un mensaje de tipo "SERVER", junto con el nombre del servidor y la dirección del cliente. Utilizamos el paquete send() del socket.

```
socketData = ("SERVER" + DELIMITER + gameSession.getServerName() + DELIMITER +
clientIp.getHostAddress()).getBytes();
packet = new DatagramPacket(socketData,socketData.length, clientIp, clientPort);
serverSocket.send(packet);
```

¿Por qué le enviamos la dirección del cliente al cliente? Este es un punto difícil de observar si no se sigue con detenimiento el flujo de la red. Cuando el cliente envía un paquete broadcast a toda la red, no solo lo está enviando a una red, sino a todas aquellas redes que tenga configuradas (es decir, sus interfaces). Para que el cliente sepa por qué interfaz ha encontrado el servidor, y por tanto, en qué dirección deberá abrir un socket posteriormente, el servidor debe notificarle cuál era la dirección de origen del paquete broadcast que llegó, y al cual respondió.

Paquete CONNECT

El tratamiento de paquete "CONNECT" es algo más complejo, puesto que entra en juego la creación de objetos Java. En este punto, el servidor tiene ya una instancia vacía de la clase GameSession, en la cual se encuentra la información de la partida (sesión) y de todos los jugadores. Estos mensajes lo que hacen es rellenar progresivamente esa información.

La clase GameSession cuenta con el nombre del servidor (que establece el jugador que la crea), la dirección IP del servidor, algunos booleanos de control para el flujo del juego y una lista de jugadores.

```
public class GameSession implements Serializable {
      private List<Player> players;
      private boolean gameStarted = false;
      private boolean gameFinished = false;
      private InetAddress serverAddress;
      private String serverName;
[...]
```

La clase Player, que representa a los jugadores, contiene atributos como la dirección IP de ese jugador, su nombre, puntuación, dos booleanos que indican si ha utilizado cada uno de los comodines, y un atributo *state* perteneciente al enum **PlayerState**, que indica el estado en que se encuentra el jugador (en qué etapa de todo el proceso, desde la gestión de sala hasta el *Game Over)*.

```
public class Player implements Serializable,Comparable {
    public enum PlayerState {
        CONNECTED, READY, PLAYING, DEAD
    }
    private InetAddress ipAddress;
    private String name;
    private int score;
    private PlayerState state;
    private boolean used50;
    private boolean usedSkip;
[...]
```

Los estados de un jugador pueden ser:

- **CONNECTED:** El jugador se ha unido a la sala.
- **READY:** El jugador está listo para empezar.
- **PLAYING:** El jugador está jugando.
- **DEAD:** Su partida terminó y está a la espera de que todos acaben.

Volviendo al tratamiento del paquete CONNECT, el servidor crea una instancia de Player, almacenando desde ese momento su IP y su nombre de jugador.

```
Player player = new Player(clientIp,parsedMessage[1]);
```

A continuación, el servidor aprovecha que ha recibido un paquete de un cliente, para obtener así la dirección local propia que deberá utilizar para el socket del juego. El motivo es el mismo por el que el servidor debe informar al cliente de su IP: el servidor tiene muchas interfaces y éste debe saber en cuál de ellas se está comunicando con los clientes. Para ello, los clientes se guardan la IP del servidor cuando reciben el mensaje "SERVER", y se lo envian a éste en el "CONNECT".

Utilizamos un *flag* para evitar hacerlo más de una vez (se da por supuesto que una vez se una un jugador, el resto se unirán a través de la misma interfaz de red).

```
if (!serverAddressSet) {
    serverAddress = InetAddress.getByName(parsedMessage[2]);
    gameSession.setServerAddress(serverAddress);
[...]
```

La variable gameSession de la que dispone RoomServer, realmente no la ha inicializado él, sino que se la pasa la actividad al thread por parámetro. La actividad, anteriormente, ya habrá creado una instancia de Player para representar al servidor (puesto que, una vez empiece el juego, todos serán 'players' y ninguno tendrá un rol diferente al resto). Cuando se inicializa ese Player particular, se le asigna una IP temporal 127.0.0.1. En el punto en que el servidor conoce la IP propia con la que trabajará, deberá "encontrarse" en la lista de jugadores y sustituir su dirección.

```
[...]
    Iterator<Player> iterator = gameSession.getPlayers().iterator();
    while (iterator.hasNext()) {
        Player p = iterator.next();
        if (p.getIpAddress().getHostAddress().equals("127.0.0.1")) {
            p.setIpAddress(serverAddress);
            break;
        }
    }
    serverAddressSet=true;
}
```

Independientemente de todo este proceso, añadimos a la lista de jugadores al jugador que se acaba de conectar a la sala.

```
gameSession.addPlayer(player);
```

Si queremos actualizar la interfaz de usuario desde el thread, debemos ejecutar el método ***runOnUiThread()*** de la clase Context (y es que cuando creamos el thread, debemos pasarle por parámetro el contexto -la actividad de Android- desde la que se creó)

```
((ServerWaitingRoomActivity)context).runOnUiThread(new Runnable() {
    @Override
    public void run() {
        playersRecyclerAdapter.notifyDataSetChanged();
    }
});
```

Paquete READY

Cuando un jugador nos indica que está listo para jugar, el servidor lo establece como tal en la lista de jugadores (identificándolo a partir de su IP), luego la recorre de nuevo para comprobar que todo el mundo está listo. En caso afirmativo, se activa el botón para empezar el juego. En caso contrario, el botón se mantiene desactivado.

```
//Establecer READY
for(Player p : gameSession.getPlayers()) {
    if (p.getIpAddress().equals(clientIp)) {
        p.setState(Player.PlayerState.READY);
        break;
    }
}

[...]

//Comprobar si todos están READY
boolean notReady = false;
for (Player p : gameSession.getPlayers()) {
    if (p.getState() != Player.PlayerState.READY) {
        notReady=true;
        break;
    }
}
```

En caso de que el flag no se haya activado, significará que todos están listos para empezar, se activará un booleano en la instancia de gameSession, y a continuación se ejecutará un thread que se encarga de enviar la instancia de gameSession a todos los jugadores, mediante el protocolo TCP.

```
Iterator<Player> iterator = gameSession.getPlayers().iterator();
while (iterator.hasNext()) {
    try {
        Player p = iterator.next();
        socket = new Socket(p.getIpAddress(),15151);
        OutputStream out = socket.getOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(out);
        oos.writeObject(gameSession);
        out.flush();
[...]
```

Los clientes, cuando están en la actividad ClientWaitingRoomActivity, empiezan a escuchar por TCP este tipo de objeto desde el thread **RoomUpdatesListener**. De este modo, comprobarán al recibirlo que el booleano que indica que se debe empezar la partida está activado.

Utilizamos para este cometido las clases **ServerSocket, Socket, OutputStream y ObjectInputStream**. El siguiente código pertenece al lado del cliente:

```
serverSocket = new ServerSocket(port);

while(continueConnected) {
    serverReplySocket = serverSocket.accept();
    in=serverReplySocket.getInputStream();
    out=serverReplySocket.getOutputStream();
    ObjectInputStream ois = new ObjectInputStream(in);

    try {
        gameSession = (GameSession) ois.readObject();
[...]
```

Si el booleano está activado, se desactiva el flag de control del thread, para que continúe su ejecución fuera del bucle, e inmediatamente se ejecuta un Intent a la actividad del juego multijugador. Se pasa a la actividad la instancia de gameSession para tener toda la información de partida y jugadores.

```
if (gameSession.isGameStarted()) {
    continueConnected=false;
}

Intent playIntent = new Intent(context, MathsMultiplayerActivity.class);
playIntent.putExtra("gameSession",gameSession);
context.startActivity(playIntent);
```

Unión a sala

Una vez visto el comportamiento desde el lado del servidor, vamos a ver detalladamente el lado del cliente. El jugador puede buscar salas en la red (lo cual generará un mensaje broadcast), y aquellos servidores disponibles responderán con información del mismo. Una vez el usuario elige el servidor, avisará que se encuentra listo para comenzar.

Lo primero que deberá hacer el jugador es introducir su nombre de jugador. Esto no conlleva ningún tipo de comunicación en red. Sin embargo, en la siguiente actividad ya empezará a buscar salas y mostrarlas en pantalla.

Para el envío de paquetes UDP, se han definido diferentes clases de Java (threads): RoomDiscoveryClient, RoomConnectClient y RoomReadyClient.

RoomDiscoveryClient

Con el thread RoomDiscoveryClient, enviamos un paquete broadcast con la palabra clave "DISCOVER", a través de todas las interfaces del dispositivo, cada 2 segundos. Para establecer la repetición de envío de paquetes, podemos usar la clase Timer, pasando por parámetro el nombre de una instancia de tipo TimerTask donde se introducirá el código a ejecutar, un tiempo de espera inicial, y el tiempo de repetición.

```
timer.schedule(discoverServers,0L,2000L);
```

Para ello, disponemos del método estático ***getNetworkInterfaces()*** de la clase **NetworkInterface**.

Al recorrer las instancias NetworkInterface, podemos comprobar si son loopback o si están inactivas, con lo que podemos evitar enviar el paquete en este tipo de casos. Además de recorrer todas las interfaces, debemos recorrer todas las direcciones de cada interfaz (puede haber interfaces virtuales).

```
//Recorrer interfaces
Enumeration interfaces = NetworkInterface.getNetworkInterfaces();
while (interfaces.hasMoreElements()) {
    NetworkInterface networkInterface = (NetworkInterface)
interfaces.nextElement();
    //Comprobar su tipo y estado
    if (!networkInterface.isLoopback() || networkInterface.isUp()) {
        //Recorrer direcciones de la interfaz
        for (InterfaceAddress interfaceAddress :
networkInterface.getInterfaceAddresses()) {
            InetAddress broadcast = interfaceAddress.getBroadcast();
            if (broadcast != null) {
                try {
                    DatagramPacket packetConnect = new
DatagramPacket(sendingDataConnect,sendingDataConnect.length, broadcast, port);
                    socket.send(packetConnect);
[...]
```

Inmediatamente después de enviar el paquete, el cliente deberá ponerse a escuchar, ya que el servidor responderá con un mensaje con la palabra clave "SERVER", junto con información del servidor. El cliente dispondrá de una lista de objetos Server, que es una clase creada para estos efectos, y que apenas almacena un nombre y una IP.

Al recibir el paquete, se comprueba si no está ya en la lista de servers (comparando direcciones IP), y en caso de no estarlo se añade.

```
if (parsedMessage[0].equalsIgnoreCase("SERVER")) {
    boolean serverAlreadyInList=false;
    for (Server server : serverList) {
        if (server.getServerAddress().equals(packet.getAddress())) {
            serverAlreadyInList=true;
            break;
        }
    }
    if (!serverAlreadyInList) {
        Server server = new Server(packet.getAddress(), parsedMessage[1]);
        serverList.add(server);
    }
```

A continuación, se guarda la dirección IP del cliente (que ha recibido en el paquete anterior por motivos ya mencionados), en el fichero de preferencias locales (SharedPreferences). Esto es por dos motivos: primero para evitar tener que pasar este valor a otras clases y a través de intents, y segundo para que más adelante podamos saber, de la lista de jugadores, quién corresponde al dispositivo actual. Al ser gameSession un objeto compartido, no puede incluir esa información. De este modo, se recorre la lista y se compara la IP de jugador con la guardada en preferencias. Si coinciden, ese jugador de la lista será el del propio dispositivo.

```
SharedPreferences sharedPref = context.getSharedPreferences("preferences",
Context.MODE_PRIVATE);
sharedPref.edit().putString("clientIp", parsedMessage[2]).apply();
```

RoomConnectClient

En el caso del paquete CONNECT, que se envía cuando el usuario solicita unirse a una sala, lo único que se hace es enviar dicho paquete e inmediatamente se hace un *intent* a la sala de espera (ClientWaitingRoom). En este caso, no se ha implementado un sistema de ACK, lo cual requeriría un paquete extra, pero sería una mejora más que razonable, ya que en esta fase estamos utilizando UDP (protocolo sin conexión).

```
DatagramPacket packetConnect = new DatagramPacket(sendingDataConnect,
            sendingDataConnect.length,
            server.getServerAddress(),
            15151);
socket.send(packetConnect);
Intent intent = new Intent(context,ClientWaitingRoomActivity.class);
intent.putExtra("serverAddress",server.getServerAddress().getHostAddress());
context.startActivity(intent);
```

RoomReadyClient

Para que el juego empiece, todos los jugadores tienen que avisar que están listos y el servidor debe empezar la partida. Para ello, la sala de espera de los clientes incluye un botón a tales efectos.

Como se explica en el apartado de RoomServer, en cuanto el jugador pulsa el botón de "Listo", inmediatamente después empieza a escuchar paquetes TCP. Específicamente, se espera un paquete TCP enviado por el servidor a todos los jugadores cuando el jugador que creó la sala pulsa el botón de empezar la partida. Así, todos empiezan a la vez.

En este paquete, se envía una instancia de gameSession, que incluye un booleano llamado "gameStarted". Y en caso de estar activado, servirá para avisar a los clientes de que el juego debe comenzar, y realizar así el *intent* a la otra actividad.

Para más detalle véase el apartado **"Paquete READY"** de RoomServer.

El juego

Durante el juego, todos los dispositivos pasan a adoptar un rol híbrido de cliente-servidor, formando entre todos una arquitectura P2P. Cuando se produzca algún cambio en el estado de la jugada de cualquier jugador (pregunta correcta, pregunta incorrecta, cambio de puntuación, utilización de comodín, game over ...), el dispositivo avisará a todos los demás jugadores y actualizará un ranking a tiempo real de las partidas de los jugadores.

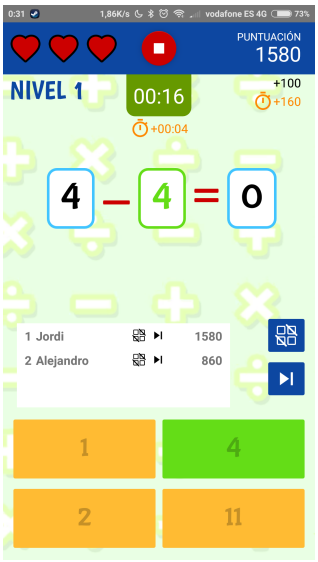


Figura 43: Pantalla del modo multijugador

Para hacer posible el modo multijugador, la herencia de clases que ofrece Java ha sido un recurso extremadamente útil. Como ya disponíamos de una clase MathsActivity donde se define todo el comportamiento del juego individual, y la lógica del modo multijugador es prácticamente la misma que la del modo individual, se ha creado una clase MathsMultiplayerActivity que extiende de MathsActivity, y que además utiliza el mismo fichero XML de interfaz que el modo individual (activity_maths.xml).

Este fichero de interfaz incluye un *include* (valga la redundancia) a otro fichero llamado online_score_list.xml donde definimos la parte de la interfaz en la que se ve la lista de jugadores en tiempo real y sus propiedades.

```
<include
    android:id="@+id/online_score_list_include"
    layout="@layout/online_score_list"
```

Fichero online_score_list.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="@android:color/white">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/online_gamesession_recycler"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Para que ahora el fichero principal de interfaz sea compatible con ambas actividades, controlamos la visibilidad de ese elemento extra dependiendo de la actividad (o modo de juego) en la que nos encontremos.

```
super.onlineList.setVisibility(View.VISIBLE);
```

Cabe decir que, para evitar tener una interfaz demasiado recargada de elementos e información, y para no tener problemas con pantallas más pequeñas, se ha decidido sacrificar el apartado de la interfaz donde se informaba de la siguiente operación.

En la nueva clase Java, sobreescribimos aquellos métodos en los que se producirán eventos que necesitarán enviar paquetes al resto de jugadores para informarles de que su status ha cambiado de algún modo. Por ello es muy importante tener la clase principal adecuadamente modularizada. Además, estos métodos deben tener visibilidad *protected* para que la clase hija pueda tener acceso a ellos.

Dentro de estos métodos sobreescritos, hacemos una llamada al método sobreescrito de la clase padre, para que se ejecuten las sentencias que se ejecutarían en el modo individual, y justo debajo añadimos el código necesario para enviar nuestra gameSession al resto de jugadores y actualizar nuestra interfaz.

Estos son los métodos que hemos necesitado sobreescribir (en casi todos ellos se llama también a la clase padre):

- **onCreate()** → Para recibir intents específicos de este modo e inicializar atributos propios de esta actividad.
- **onPause()**, **onResume()**, **onStart()** y **onDestroy()** → Para mantener los comportamientos de control del ciclo de vida.
- **useLifeline5050()** → Para avisar de que el jugador ha utilizado el comodín del 50%.
- **userLifelinePassover()** → Para avisar de que el jugador ha utilizado el comodín de omitir operación.
- **answerQuestion()** → Para avisar de que el jugador ha contestado una operación y su puntuación ha cambiado.
- **IncorrectAnswerOrTimeOut()** → Para avisar de que el jugador ha contestado una operación incorrectamente o el tiempo se le ha acabado, y por tanto ha perdido una vida y su puntuación ha cambiado.
- **gameOver()** → Para avisar de que el jugador ha perdido todas las vidas y se encuentra en estado "DEAD". Restará así hasta que el resto de jugadores terminen.
- **showLifelineHint()** → Para evitar que aparezca el bocadillo de consejo para que el jugador use un comodín (en este modo donde hay poco espacio en la interfaz, es incompatible mostrar este elemento y a la vez la lista de jugadores). Se deja el método vacío (sin llamar a *super*) para que no muestre nada.

Los métodos marcados en **rojo** son aquellos en los que se envía la sesión de juego a todos los jugadores de la sala.

Intercambio del estado de sesión

Para el envío y recibimiento de la gameSession, se utilizan dos threads llamados **gameSessionListener** y **gameSessionSender**. Al tratarse de una arquitectura de red basada en eventos, el thread gameSessionSender se ejecuta en el momento que suceda un evento, envía el paquete adecuado, y simplemente termina.

Es en los métodos mencionados anteriormente en los que se utiliza gameSessionSender. Lo primero que se hace es recorrer la lista de jugadores de gameSession para actualizar los atributos relevantes en cada caso. Como esto lo hacemos después de la llamada a la clase padre, podemos utilizar las variables de esta clase, que ya se habrán actualizado (siempre que sean *protected*). Por ejemplo, en el caso de answerQuestion() actualizamos la puntuación.

```
super.answerQuestion(view);
Iterator<Player> iterator = gameSession.getPlayers().iterator();
while (iterator.hasNext()) {
    Player player = iterator.next();
    if (player.getIpAddress().equals(clientIp)) {
        player.setScore(super.getmScore());
    }
}
sendGameSession();
```

El método sendGameSession() inicia un gameSessionSender y actualiza el RecyclerView de la lista local.

```
GameSessionSender gameSessionSender = new GameSessionSender(gameSession);
gameSessionSender.start();
gameSessionPlayersRecyclerAdapter.notifyDataSetChanged();
```

El código de gameSessionSender es exactamente el mismo que se utilizaba para enviar gameSession justo antes de empezar el juego.

```
Iterator<Player> iterator = gameSession.getPlayers().iterator();
while (iterator.hasNext()) {
    try {
        Player p = iterator.next();
        socket = new Socket(p.getIpAddress(),15151);
        OutputStream out = socket.getOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(out);
        oos.writeObject(gameSession);
        out.flush();
[...]
```

En el caso de **gameSessionListener**, sí que se utiliza un mismo thread para todo el recorrido de una partida, puesto que todos los jugadores deberán estar escuchando en todo momento a todos los jugadores. Dentro del onCreate() de la clase extendida, iniciamos dicho thread.

Cuando se recibe por red la instancia de GameSession, se actualiza el valor local de *gameFinished* con el valor recibido. También se limpia la lista de jugadores local y se utiliza el método *addAll()* del ArrayList para añadir a esta todos los jugadores recibidos por red. Esto se hace para no perder la referencia de memoria de la lista players local (utilizada por el adapter del RecyclerView de la lista de la interfaz).

```
GameSession gameSessionRead = (GameSession) ois.readObject();

l.lock();

//Actualizar campos
gameSession.setGameFinished(gameSessionRead.isGameFinished());
gameSession.getPlayers().clear();
gameSession.getPlayers().addAll(gameSessionRead.getPlayers());

//Reordenar lista
Collections.sort(gameSession.getPlayers());

//Notificar al adapter
((MathsMultiplayerActivity)context).runOnUiThread(new Runnable() {
      @Override
      public void run() {
          adapter.notifyDataSetChanged();
      }
});

l.unlock();
```

Nótese que justo antes y después de actualizar la lista, utilizamos los métodos lock y unlock de la variable estática l. Esta variable es de tipo **Lock** (con la implementación **ReentrantLock**), y la utilizamos para evitar errores de concurrencia en los que la lista de jugadores sea modificada a la vez por dos threads distintos (cuando se reciba por red y cuando el usuario local deba cambiarla para enviarla). Esta variable hace las veces de semáforo, y hará que todas las modificaciones de esa lista estén sincronizadas y se realicen una detrás de la otra, y no a la vez.

Y es en este thread donde se comprueba si el flag gameFinished de gameSession está activado. En tal caso, se hará un Intent a la actividad donde se muestra el ranking final y se activarán los flags para finalizar el thread. Esto significa que, cuando todos los jugadores reciban este paquete, todos finalizarán la actividad a la vez.

```
if (gameSession.isGameFinished()) {
    Intent resultIntent = new Intent(context,MultiplayerResultsActivity.class);
    resultIntent.putExtra("gameSession",gameSession);
    context.startActivity(resultIntent);
    shutdown();
}
```

Pero, ¿quién activa ese flag? Pues esto se hace en la sobreescritura del método ***gameOver()***. Cuando el jugador "muere", se establece como tal en gameSession, como se hace en los otros métodos, pero antes de enviar la instancia al resto de jugadores, se vuelve a recorrer la lista de jugadores para ver si todos están en estado DEAD. En caso afirmativo, eso significará que el jugador es el último que quedaba por "morir", y por tanto deberá avisar al resto de jugadores, que en este momento estarán en espera, que el juego ha concluido.

```
iterator = gameSession.getPlayers().iterator();
boolean everyoneDead=true;
while (iterator.hasNext()) {
    Player player = iterator.next();
    if(!player.getState().equals(Player.PlayerState.DEAD)) {
        everyoneDead=false;
        break;
    }
}
```

Por ello, si todos están en estado DEAD, se activa el booleano gameFinished en gameSession y se envía a todos los jugadores.

```
if (everyoneDead) {
    gameSession.setGameFinished(true);
}

sendGameSession();
```

Ranking final

Una vez terminado el juego, se muestra una lista con los resultados finales de la partida. Esta lista es la misma que se mostraba durante el juego, con la diferencia que esta vez se ve en pantalla completa, y con opciones para salir al menú principal.

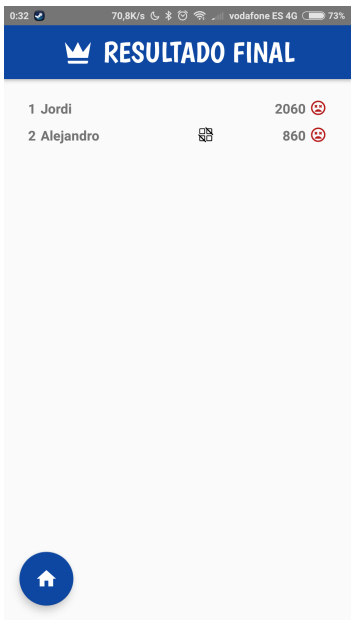


Figura 44: Ranking final del modo multijugador

10. Méritos y reconocimientos

ThinkItApp, además de ser nuestro proyecto de síntesis, es hoy en día casi un producto. Llevamos desde noviembre-diciembre desarrollando la aplicación, teniendo varios *deadlines* a través de estos pasados meses, y ha ido adquiriendo una calidad y un alto potencial de convertirse en nuestro primer proyecto personal de desarrollo a largo plazo después de terminar el ciclo formativo.

10.1 Proyecto Switch – Evento 'Multiplier'

ThinkItApp empezó como un pequeño proyecto para el evento 'Multiplier' del Proyecto Switch que se realizó en febrero de 2018. Ese fue nuestro primer *milestone*, desarrollar un juego de matemáticas que, aunque tuviera fallos, fuera presentable y una buena base a partir de la cual realizar la presentación. Cumplimos el objetivo con creces, desarrollamos el modo de un jugador, con todas sus características, y múltiples funcionalidades extra como el cambio de idioma o la ayuda. Aquel día, la presentación fue todo un éxito y los profesores británicos invitados a este evento de intercambio de experiencias educativas quedaron encantados con el resultado. El equipo de profesores de nuestro instituto también quedó satisfecho con el resultado y recibimos elogios por su parte.

Figura 45: Presentación de la app en el evento 'Multiplier' del Proyecto Switch

10.2 Proyecto Switch – Evento final

Debido al éxito que tuvo la presentación, la app será presentada en el Halesowen College, en Birmingham, centro académico que organiza el Proyecto Switch junto con otros centros de Europa (como es el caso del Instituto Puig Castellar), en el **evento final del proyecto**. Para entonces, se espera poder anunciar el lanzamiento al público en la Google Play Store.

Figura 46: Halesowen College - Birmingham

10.3 Concurso de WebsAlPunt.cat

Debido al avanzado estado de desarrollo en que se encontraba el modo individual después del evento 'Multiplier', se decidió intentar corregir fallos graves en la jugabilidad y en el funcionamiento de la app, y presentar la aplicación en el concurso de WebsAlPunt.cat que organiza la Fundació .CAT. Para podernos presentar, se requirió desarrollar un sitio web con dominio .cat en el que hacemos un breve repaso por las características del juego, además de mostrar un vídeo demostrativo y ofrecer un enlace de descarga al archivo ejecutable '.apk'. El dominio es **thinkitapp.cat**.

Figura 47: Sitio web de thinkitapp.cat

En mayo de 2018, se entregaron los premios en un evento en CosmoCaixa. Después de exponer al jurado las virtudes de nuestra app, esta resultó ganadora en dos categorías: **Votación popular** y **Mejor app**. Lo cierto es que la campaña que hizo el profesorado de nuestro instituto, y nosotros mismos, fue todo un éxito. Y además, el jurado consideró que nuestra aplicación era la mejor de las nominadas.

Sin duda fue una recompensa a varios duros meses de trabajo.

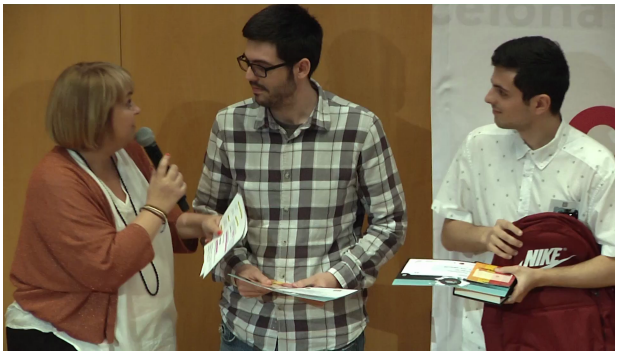


Figura 48: WebsAlPunt.cat - Recogida de premios

Figura 49: El equipo de desarrollo posando con los premios

10.4 Proyecto de síntesis

Tras pasar por todas esas etapas, se recupera una idea que siempre ha estado presente: presentar esta aplicación como proyecto de síntesis. Y después de todos estos meses, se ha querido "subir de nivel" y aprovechar las últimas semanas de curso para desarrollar un modo muy particular: el modo multijugador.

Aprovechando nuestros recientes conocimientos en la programación de Sockets en Java, creíamos que era el momento adecuado para implementar un modo de tales características. El resultado de estas últimas semanas ha sido excelente, recibiendo de nuevo elogios por parte de los profesores, y es el producto en que se ha basado este documento.

11. Líneas de futuro

El futuro de ThinkItApp todavía está por decidir, tenemos una gran cantidad de ambiciosas ideas que, o bien no se han podido implementar por falta de tiempo, o bien hemos considerado que excecen el ámbito de este proyecto.

La principal idea que tenemos en mente, y que más que idea es la finalización de una característica, trata de añadir las **cadenas de texto traducidas al inglés y al italiano**. Cuando se planeó el desarrollo de la app en colaboración con el instituto, se acordó que la traducción de la app a otros idiomas correría a cuenta del profesorado colaborador. Lamentablemente, solamente les ha sido posible entregarnos la traducción al catalán, mientras que nosotros mismos nos encargamos del castellano, que fue el idioma inicial en que se desarrolló. Se espera que estos idiomas sí que estén disponibles en la presentación que se realizará en Birmingham. No obstante, la funcionalidad de cambio de idioma está completamente desarrollada, y el único obstáculo son las cadenas de texto.

Una idea que sí que ha surgido constantemente durante el proceso de desarrollo, es el de hacer descargable la app a través de Google Play Store, la tienda oficial de apps para Android. No obstante, debido a que hemos querido aprovechar al máximo el tiempo en el último mes para implementar el modo multijugador, hemos considerado que la app aún requiere algunas semanas más de perfeccionamiento y corrección de *bugs*, máxime cuando hablamos de que la app esté disponible al público en un mercado de millones de usuarios. Esto junto con la disponibilidad de todos los idiomas, son las principales ideas y mejoras que queremos ofrecer en la presentación de Birmingham.

Aparte de estas mejoras, tenemos pensadas otras enfocadas a continuar con el desarrollo de la aplicación a largo plazo, ampliando sus funcionalidades. Una funcionalidad interesante, sería implementar un **modo de entrenamiento diario**, en el que se **guarden estadísticas** diarias de cada una de las partidas del jugador, en forma de gráficos, con la finalidad de mostrar la evolución del cálculo mental de este. También sería interesante que estas estadísticas y los resultados de cualquier tipo, se almacenen bajo un sistema de cuentas de usuario y una base de datos en la nube, como por ejemplo la base de datos en tiempo real de **Firebase**.

Figura 50: Logotipo de Firebase (Google)

Otra de las mejoras que hemos planteado, y una de las más ambiciosas, es hacer la aplicación **jugable a través de Internet**, no sólo en la misma LAN, e incluso utilizar tecnologías alternativas como el Bluetooth. También queremos **añadir otros juegos** dentro de la app, convirtiendo la aplicación en un conjunto de minijuegos que requieran pensar para lograr la puntuación máxima posible, combinando las dos ideas con las que iniciábamos este documento: conocimiento y tecnología.

Hablando del modo multijugador, este actualmente tiene una limitación, y es que al estar basado en mesajes broadcast para el descubrimiento de salas de juego, existe la posibilidad de que esto no sea técnicamente posible, debido a que el router local descarte por defecto ese tipo de paquetes. Para usar este modo en esos casos, habría que crear una red WiFi personal y hacer que los usuarios se conecten a esa red. Sin embargo, consideramos que no es una solución apropiada para jugadores menos experimentados en el mundo de la informática. Por ello hemos pensado varias alternativas más *user-friendly*, como por ejemplo un sistema de códigos QR que el usuario pueda escanear para obtener información del servidor y conectarse a él, o bien la generación de códigos a partir de su dirección IP y la introducción manual en el lado del cliente.

Otra idea interesante no enfocada a la ampliación de características pero sí de público potencial, sería desarrollar versiones de la app para **otros sistemas operativos** además de Android, como por ejemplo iOS o Windows. Esto conllevaría a la utilización de frameworks, SDKs o motores de juego específicos de cada plataforma.

Además de las ideas anteriores, nos gustaría obtener ingresos a través de la publicidad o el lanzamiento de dos versiones diferentes, una gratuita y otra de pago o con publicidad no invasiva. La versión gratuita, de características reducidas podría llamarse ***ThinkItApp Student***, y estaría enfocada a estudiantes. De esta versión estaría disponible el código fuente pero se prohibiría la reproducción de este para otro fin que no sea educativo. La versión con ingresos, llamada simplemente ThinkItApp, proporcionaría funcionalidades extra, como la modalidad de juego multijugador o funcionalidades online. No obstante, qué funcionalidades irían en una versión u otra es un punto a debatir por parte de los desarrolladores.

12. Conclusiones

ThinkItApp ha supuesto nuestro primer proyecto de desarrollo de una app a largo plazo. Desarrollar una aplicación de mediana o gran envergadura requiere mucho tiempo y dedicación, además de capacidad de trabajo en equipo. La mayoría de aplicaciones que hemos realizado durante el curso son pequeñas, y probablemente no son susceptibles de convertirse en un proyecto a largo plazo, añadiendo funcionalidades incluso después de haberlas entregado.

En cambio, desde que comenzó el desarrollo teníamos grandes planes con nuestra aplicación y estos meses han sido duros para alcanzar las fechas importantes a las que nos hemos sometido. Desde la primera versión para el Proyecto Switch, hasta la última versión para el proyecto de síntesis, hemos tenido que compaginar los 7 módulos con los que contaba el curso (e incluso la Formación en Centros de Trabajo), con el desarrollo de esta app, siempre con las fechas límite al acecho.

Desarrollar una aplicación de forma que no puedes enfocar tus esfuerzos solamente en ella, requiere sacrificar mucho tiempo libre, requiere que el código esté comentado, ordenado y no acumular fallos graves para evitar la reescritura de código. Pero por encima de todo, requiere mantener la cabeza fría en los momentos más complicados donde algo parece no tener solución o donde las obligaciones del resto de módulos impiden que el desarrollo de la app siga adelante.

Como es habitual en la programación, a medida que la aplicación iba creciendo, aparecían dificultades y errores que no estaban previstos. Estos obstáculos conllevan la realización de cambios relevantes en buena parte del código de la aplicación, aumentando el tiempo requerido para el desarrollo. Es por esto que, a pesar de que es útil la realización de una planificación previa al desarrollo o durante él, es extremadamente importante contar siempre con un margen de error para evitar entregar un producto defectuoso en la fecha límite.

A pesar de todo, el balance final no puede ser otra cosa que beneficioso. El proceso por el que hemos pasado, nos ha permitido conocer mejor cuáles son los retos en un proyecto a largo plazo, aprender a cómo afrontarlos y ganar una experiencia valiosa para el resto de nuestra carrera profesional, especialmente para los duros comienzos.

ThinkItApp ha servido de una motivación extra durante el curso para seguir programando y para que la programación, después de dos años, empiece a cobrar sentido. El entorno de Android es propicio a eso, ya que junta conceptos de front-end y back-end con los que crear un producto desde los pies a la cabeza. Y cuando uno observa, toca y prueba su propia creación, la sensación de orgullo personal es abismal. Máxime cuando los compañeros de clase y los profesores nos felicitan por ello, y cuando el proyecto nos ha abierto tantas puertas durante este año.

Finalmente, valió la pena.

Agradecimientos

A Sandra y a Teresa, por darnos la oportunidad de formar parte del Proyecto Switch y originarlo todo (gracias por confiar en nosotros, sois maravillosas). A Gerard, porque sin todo lo que nos ha enseñado no hubiera sido posible (la pasión que sientes nos motiva para seguir, eres increíble). A Dani, por creer en nuestra app para ganar el concurso y mover cielo y tierra para la votación popular (¡y nos llevamos 2!). A Fer y a Jordi por aguantar nuestros berrinches y ser flexibles con las fechas de entrega (gracias por entendernos). A todos los profesores que tuve en el primer curso (¡gracias Jose por tu paciencia conmigo este año!) y a los que tuve el grado medio en el Instituto Badalona VII. A todos los compañeros por vuestra ayuda, compañía y apoyo durante estos años. A los amigos que me llevé del grado medio y que siguen siendo grandes apoyos. Y a ti, Alejandro, por tus esfuerzos con todos los trabajos que hemos hecho, por reírte conmigo en los buenos momentos y por escucharme en los malos, pero por encima de todo por tu amistad, que es lo más importante que me llevo de aquí.

-Jordi

Agradezco el apoyo, la ayuda e interés de todas las personas, que no son pocas, que se han preocupado por el avance de nuestra aplicación y nuestro bienestar mientras la desarrollabamos. Agradezco a Teresa y Sandra el tiempo que han dedicado en el proyecto, por ofrecerme ésta gran oportunidad que me ha hecho aprender mucho, y ha mejorado mi CV, además de la posibilidad de viajar por primera vez en mi vida. A Jose Enrique, que envió mi CV, y gracias a él, tuve la oportunidad de ser entrevistado en mi futuro y primer empleo. A Daniel por ser el tutor del concurso y acompañarnos durante éste, además de cambiar el plazo de las entregas para proporcionarnos más tiempo. A Gerard por el entusiasmo, el feedback y todo lo que nos ha enseñado sobre Android, también su disposición y la ayuda con el código. A Fernando por sus ideas, apoyo y feedback, además, de ayudarnos a mejorar la aplicación y cambiar el plazo de las entregas. A Jordi Hernández por la ayuda para solucionar los distintos problemas encontrados durante la implementación del modo multijugador y por proporcionarnos más tiempo.

Por parte de mis compañeros, quiero agradecer a Cristina y Raúl por su opinión y feedback. A Aleix como tester del modo multijugador. Y especialmente a mi compañero Jordi S, sin su apoyo, su amistad, su paciencia, su dedicación y la gran habilidad que posee, éste projecto no hubiera sido posible. También, sin el apoyo de mi antiguo compañero Fran de Grado Medio, que dejó el ciclo superior, no estaría dónde estoy ahora.

Gracias por la oportunidad de dejarme formar parte de este instituto. Se me concedió el deseo de tener una nueva oportunidad en la vida, anteriormente no fuí buen estudiante, cuando llegué apenas sabía si sería capaz de obtener el título de Técnico. A pesar de que éstos dos últimos años no me he esforzado como debería haberlo hecho, he conseguido el título de Técnico Superior, que ha sido todo un desafío.

-Alejandro