



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

 etsinf

Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de una aplicación móvil multiplataforma para la creación y resolución de nonogramas

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Ignacio Ferrer Sanz

Tutor: Germán Francisco Vidal Oriola

Curso 2020-2021

Resumen

WIP

Palabras clave: WIP

Resum

WIP

Paraules clau: WIP

Abstract

WIP

Key words: WIP

Índice general

Índice general	v
Índice de figuras	vii
Índice de tablas	vii
<hr/>	
1 Introducción	1
1.1 Contexto y motivación	1
1.2 Objetivos	2
1.3 Metodología	2
1.3.1 Ciclo de vida de desarrollo	2
1.3.2 Metodología Personal Extreme Programming (PXP)	3
1.3.3 Enfoque personal	4
1.4 Estructura de la memoria	4
2 Estudio estratégico	7
2.1 Nonogramas en la era Digital	7
2.1.1 Nonograms Katana	7
2.1.2 Nonogram.com - Picture cross number puzzle	9
2.1.3 Nono Infinite	10
2.1.4 Family Crest Nonogram	11
2.2 Análisis de las aplicaciones	12
2.3 Propuesta	13
3 Análisis del problema	15
3.1 Especificación de requisitos	15
3.1.1 Propósito	15
3.1.2 Ámbito	15
3.1.3 Terminología	15
3.1.4 Modelo de Dominio	16
3.1.5 Límites del Sistema	18
3.1.6 Restricciones del Sistema	18
3.1.7 Características del Sistema	19
3.1.8 Requisitos funcionales	20
4 Diseño de la solución	29
4.1 Tecnología empleada	29
4.1.1 Flutter y Dart	29
4.1.2 Firebase	30
4.1.3 Visual Studio Code	32
4.1.4 Git	32
4.2 Arquitectura del Sistema	32
4.2.1 Clean Architecture	33
4.2.2 Manejador de estados	34
5 Desarrollo de la solución	35
5.1 Prototipado de pantallas	35
5.2 Estructura del proyecto	38

5.3	Codificación del aplicativo	39
5.3.1	Desarrollo de la capa de presentación	39
5.3.2	Desarrollo de la capa de lógica	40
5.3.3	Desarrollo de la capa de dominio	41
5.3.4	Desarrollo de la capa de datos	41
5.3.5	Inyección de dependencias	41
6	Pruebas	43
6.1	Metodología TDD	43
6.1.1	Pruebas unitarias	44
6.1.2	Pruebas de integración	45
6.1.3	Pruebas end-to-end	45
7	Implantación y mantenimiento	47
7.1	Integración Continua	47
7.2	Actualización del proyecto	47
7.3	Publicación	48
8	Manual de uso	49
9	Conclusiones y trabajo futuro	51
9.1	Relación del trabajo relacionado con los estudios cursados	51
Bibliografía		53

Apéndices

A	Diagrama de Gantt del proyecto	55
B	Fragments de código	57
B.1	Código de Pantalla de Resolución de Nivel	57
B.2	Código de Capa de Lógica de Niveles En Línea	58
B.3	Código de Capa de Dominio de Niveles En Línea	59
B.4	Código de Capa de Datos de Niveles En Línea	60
B.5	Código Inyector de dependencias	62

Índice de figuras

1.1	Diagrama de la metodología PXP	3
2.1	Pantalla principal de Nonograms Katana	7
2.2	Pantallas de Nonograms Katana con modalidad a color.	8
2.3	Modal de restricción en publicación en Nonograms Katana	8
2.4	Pantallas de Nonogram.com	9
2.5	Pantalla de Evento Primavera de Nonogram.com	9
2.6	Pantallas de selección de nivel y juego de Nono Infinite	10
2.7	Pantallas de tutorial en Nono Infinite	11
2.8	Pantallas de Family Crest	11
3.1	Diagrama del modelo de dominio	16
3.2	Diagrama de contexto del aplicativo	18
3.3	Diagrama de casos de usos principales	19
3.4	Diagrama de casos de usos en pantalla de resolución de nivel	19
3.5	Diagrama de casos de ajustes	20
4.1	Diagrama de renderización de <i>Flutter</i> [17]	29
4.2	Diagrama de flujo con <i>Firebase</i>	31
4.3	Diagrama de estructura de datos de <i>Firestore</i>	31
4.4	Diagrama del funcionamiento de <i>GitFlow</i>	32
4.5	Diagrama de <i>Diagrama Clean Architecture</i>	33
5.1	MockUps Pantalla de Inicio y Selector de niveles clásicos	35
5.2	MockUps Pantalla de niveles online y publicación de nonogramas	36
5.3	MockUp Pantalla de resolución de nivel	36
5.4	MockUp Pantalla de ajustes y cuenta	37
5.5	Estructura de directorios de <i>nonogramchallenge</i>	38
5.6	Árbol de widgets Pantalla de Resolución de nivel	39
5.7	Diagrama de estado de la funcionalidad Lista de nonogramas en línea	40
6.1	Diagrama de flujo de la metodología TDD	43
6.2	Diagrama de los tipos de pruebas en el aplicativo	46

Índice de tablas

2.1	Comparativa de características entre aplicaciones de interés y su inclusión como requisito	12
-----	--	----

3.1	Glosario de términos ontológicos del aplicativo	16
3.2	Glosario de términos técnicos del aplicativo	16
3.3	Atributos de la clase Usuario	17
3.4	Atributos de la clase Nivel	17
3.5	Atributos de la clase Progreso	17
3.6	Atributos de la clase Nonograma	18
3.7	Restricciones del sistema	18
3.8	Tabla de requisito funcional <i>Jugar nivel</i>	20
3.9	Tabla de requisito funcional <i>Visualizar Tutorial</i>	21
3.10	Tabla de requisito funcional <i>Acceder sección En Línea</i>	21
3.11	Tabla de requisito funcional <i>Acceder sección Ajustes</i>	21
3.12	Tabla de requisito funcional <i>Seleccionar nivel clásico</i>	21
3.13	Tabla de requisito funcional <i>Publicar nonograma</i>	22
3.14	Tabla de requisito funcional <i>Resolver nivel online</i>	22
3.15	Tabla de requisito funcional <i>Jugar nonograma</i>	22
3.16	Tabla de requisito funcional <i>Superar nivel</i>	23
3.17	Tabla de requisito funcional <i>Perder nivel</i>	23
3.18	Tabla de requisito funcional <i>Cargar progreso</i>	23
3.19	Tabla de requisito funcional <i>Salir del nivel</i>	24
3.20	Tabla de requisito funcional <i>Guardar progreso</i>	24
3.21	Tabla de requisito funcional <i>Cambiar tema</i>	24
3.22	Tabla de requisito funcional <i>Seleccionar vidas</i>	25
3.23	Tabla de requisito funcional <i>Iniciar sesión</i>	25
3.24	Tabla de requisito funcional <i>Seleccionar autoguardado</i>	25
3.25	Tabla de requisito funcional <i>Borrar datos</i>	26
3.26	Tabla de requisito funcional <i>Seleccionar idioma</i>	26
3.27	Tabla de requisito funcional <i>Cerrar sesión</i>	26
3.28	Tabla de requisito funcional <i>Sincronizar datos</i>	27
4.1	Funcionalidades <i>en nube</i> cubiertas por <i>Firebase</i>	30
4.2	Función de las capas de la arquitectura del sistema	34

CAPÍTULO 1

Introducción

Descubrir imágenes hechas píxel de situaciones del día a día, naturaleza, edificios famosos, personas, y cuantas cosas más, esta es la verdadera esencia de los nonogramas, también conocidos como hanzies, picross o griddlers.

1.1 Contexto y motivación

“No importa lo complejo que sea resolver un nonograma, la clave de estos rompecabezas reside en que su resolución pueda efectuarse por simple lógica.” [6] Este era el principal propósito de James Dalgety y su equipo de diseñadores, responsables de dar a conocer a occidente este conocido pasatiempo nipón, impulsado por el arte de la diseñadora Non Ishida, más adelante, responsable de su principal denominación: “Non” Ishida y Dia “gram”.

No fue hasta mediados del año 1990, cuando finalmente se dio a conocer los *nonogramas* a escala mundial, a través de una publicación del periódico británico *The Sunday Telegraph*. Más adelante, el mismo noticiero adoptó el término bajo el seudónimo de *griddlers*, publicándolos semanalmente.

A partir de estas publicaciones, se fue difundiendo exponencialmente el famoso puzzle y se puede encontrar en revistas, otros periódicos y libros. Fue tan notable su crecimiento que, como otros rompecabezas, alcanzó con prontitud el formato digital, en forma de sitios web, videojuegos y aplicaciones.

La capacidad creativa que ofrece resulta ilimitada, ya que con tan solo sus celdas dispuestas en forma de matriz (*filas y columnas*), permite representar todo tipo de figuras, siluetas y formas, como si de un lienzo se tratara.

Sorprendentemente y a pesar de que nos encontramos en plena era digital, son pocos los medios que ofrecen una capacidad de creación, más allá del simple tradicional método del lápiz y papel y así esta herramienta ayuda y otorga al jugador no solo el rol de *resolutor*, sino de *creador*, e impulsa que la cuantía de *nonogramas* a resolver no disminuya.

Un medio digital es ideal para, no solo hacer que esta propiedad de creación sea posible, sino de facilitar su proceso y que sea lo más liviano y recreativo posible, de esta forma las aplicaciones móviles, cada vez más conocidas y presentes en nuestra sociedad, constituyen el entorno perfecto.

Siguiendo esta premisa, lo que sería adecuado es que cualquier usuario pudiera, dentro de lo posible, emplear estos medios independientemente de cual sean las características técnicas, sistemas operativos y prestaciones de sus dispositivos.

1.2 Objetivos

El presente trabajo explora el mundo de las aplicaciones móviles y propone una solución software para: i) permitir al usuario resolver *nonogramas* de forma interactiva ii) dar la oportunidad de crear sus propios puzzles como lo hizo *James Dalgety* y su equipo y compartirlos con todos los demás usuarios, promoviendo así una comunidad de entusiastas de este divertido rompecabezas.

Por consiguiente, para el correcto desarrollo y funcionamiento de la aplicación se deben de cumplir una serie de requerimientos a nivel técnico bien diferenciados:

- Estudiar y desarrollar una funcionalidad integrada en el aplicativo, con el fin de posibilitar al usuario crear y resolver *nonogramas* en un dispositivo móvil, bajo unas variables determinadas, siempre dando prioridad a la experiencia de juego.
- Implementar un *backend* con el que el aplicativo pueda complementar sus funcionalidades con *servicios en nube*, tales como la base de datos, sincronización o inicios de sesión.
- Seguir los principios de *Clean Arquitecture* durante el proceso de desarrollo, implementando patrones de diseño y aplicando *suites de test* con el objetivo de aminorar el proceso de mantenimiento.
- Definir y realizar un MVP (*Mininum Viable Product*), con el que usuario pueda, en una primera versión, hacer uso de sus funciones principales.

Finalmente, encontramos requisitos de índole personal, que progresivamente se considerarán como cumplidos durante todo el desarrollo del proyecto, tales como:

- Ahondar en el desarrollo de una aplicación móvil, desde su inicio hasta su finalización, bebiendo de buenas prácticas y recomendaciones propuestas por artículos, documentaciones y comunidades de desarrolladores.
- Comprender y profundizar en el extenso mundo de desarrollo de juegos de puzzles, haciendo frente y reduciendo su marcada complejidad.

1.3 Metodología

El aplicativo resultante, como cualquier otra solución software, debe satisfacer una serie de requerimientos frente un problema o problemas concretos. Esta solución podría residir o enfocarse en diferentes plataformas, bajo una perspectiva tanto de *software* como de *hardware*.

Así mismo, antes de que esté considerada preparada la aplicación para su uso, ha de atravesar por una serie de procesos, que difieren mucho de un simple desarrollo y en su conjunto reciben el nombre de *System Development Life Cycle (SDLC)*, que corresponden a su ciclo de vida.

1.3.1. Ciclo de vida de desarrollo

Comprender esta serie de procesos, junto a los requisitos recién comentados, es definitorio para elegir una metodología ideal, que sirva como guía para el correcto desarrollo total del producto.

- **Planificación:** identificar los requisitos necesarios para la aplicación, considerando y comparando otras soluciones disponibles, para así establecer un *target* o perfil de usuario ideal para nuestra aplicación, sin entrar en el apartado técnico.
- **Análisis:** establecer los requisitos funcionales de la aplicación, recopilando y anticipándose a aquellos que puedan suponer un problema para la evolución del ciclo de vida.
- **Diseño:** documentar las, ya definitivas, características, partes y componentes a integrar en el aplicativo.
- **Codificación:** seguir los requisitos ya debidamente documentados e implementarlos creando así el aplicativo.
- **Testing:** realizar *suites de tests* con el fin de encontrar o mostrar posibles errores y *bugs*. Además de verificar que los requisitos de la fase de diseño están presentes en la solución.
- **Implantación:** hacer la solución *software* disponible para el usuario, listo para su uso.
- **Mantenimiento:** monitorizar la experiencia de usuario, contemplando y dando solución a posibles errores, además de realizar cambios y mejoras en forma de nuevas versiones.

Ya que el desarrollo del producto se ha realizado por una única persona, se optó en una primera instancia, por simplicidad y falta de recursos y equipo por la *metodología tradicional: en cascada*, enfocado en el desarrollo de un único *mínimo producto viable* (MVP).

Sin embargo, esta metodología impediría el uso de prácticas y herramientas propias de las *metodologías ágiles*, como por ejemplo: *la automatización de pruebas* o *la integración continua*. Por ello, tras un estudio de posibles metodologías del mundo del *software*, finalmente, para el desarrollo del producto, se optaron por las pautas y directrices marcadas por el subrama de la metodología *Extreme Programming: Personal Extreme Programming* (PXP).

1.3.2. Metodología Personal Extreme Programming (PXP)

Esta metodología presenta todos los beneficios que engloba la metodología *Extreme Programming*, normalmente enfocada al *desarrollo ágil para equipos de pequeña envergadura*, con algunas singularidades que la hacen única.

La idea central de esta subvertiente es la sustitución del concepto *desarrollo por pares*, propia de la metodología *XP*. Este modelo sostiene que el desarrollo debe de realizarse por dos programadores bajo revisiones continuas por turnos, mientras que en *PXP* la totalidad de las fases la realiza un solo desarrollador [1].

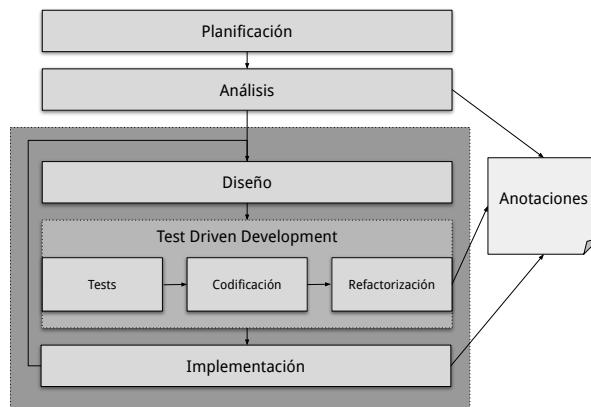


Figura 1.1: Diagrama de la metodología PXP

En cuanto al diagrama de flujo de las diferentes fases que compone la metodología *PXP* ([Figura 1.1](#)) no difiere mucho del clásico modelo Personal Software Process (*PSP*). Sin embargo, a diferencia de la corriente clásica, este permite realizar iteraciones si se requiere algún cambio de diseño en alguno de los requisitos.

Es importante remarcar que al igual que el modelo *PSP*, es recomendable realizar durante todo el proceso *anotaciones* de interés como: tiempos reales, sugerencias de mejora, detalles..., estas anotaciones serán de gran importancia de cara a cambio de requisitos o desarrollo de nuevas funcionalidades.

- Es necesario hacer hincapié en las etapas tempranas del modelo, ya que son las que van a definir de manera correcta los requisitos de la solución. De forma que, si no están bien establecidos, pueden aparecer problemas en el resto de fases [\[22\]](#).
- En la práctica, es recomendable marcar bien los tiempos de cada fase, marcando estimaciones de cada fase y contrastarlas con el tiempo real que ha supuesto realizarlas. Esta especie de monitorización se realizará periódicamente y se puede encontrar en el [Apéndice A](#)

1.3.3. Enfoque personal

Pese a que, a nivel personal, por experiencia en entornos profesionales, iba a resultar más cómodo el empleo de la metodología *ágil*, *SCRUM*, existían una serie de razones por las que descartar esta metodología:

- Para este proyecto, por tratarse de un trabajo individual resultaba insostenible adoptar esta metodología, ya que está enfocada a un equipo de desarrollo, mientras que la subvertiente *PXP* enfoca la totalidad del desarrollo en un único programador.
- La metodología *PXP*, junto a su rama padre *XP* promueven el uso de tests realizados por el propio programador, para verificar y validar los requisitos, empleando normalmente la metodología *TDD*. Sin embargo, para el uso de *SCRUM*, en muchas ocasiones, se requiere de un equipo de *QA* para efectuar la validación de requisitos.
- En *SCRUM*, los requisitos no pueden ser cambiantes, una vez se definen son definitivos, en *PXP* existe un margen de mejora ante un posible problema o estímulo, hecho algo probable en escenarios de esta índole.

1.4 Estructura de la memoria

El resto de capítulos que conforman la memoria, son los que se resumen a continuación:

- **Capítulo 2. Estudio estratégico:** corresponde al conocido apartado *estado del arte*, en el que se realiza una labor de estudio de aquellas soluciones relacionadas con *nonogramas* dentro del mundo digital y que puedan ayudar a la identificación y extracción de requisitos.
- **Capítulo 3. Análisis del problema:** en este capítulo se expone la parte de especificación de requisitos, identificando aquellos que puedan repercutir negativamente al transcurso del proyecto.
- **Capítulo 4. Diseño de la solución:** se muestran las decisiones que se han tomado a nivel de arquitectura, patrones de diseño, y tecnologías empleadas.

- **Capítulo 5. Desarrollo de la solución:** se comentan las distintas partes que han com-
puesto el aplicativo, cómo se comportan y el funcionamiento interno de cada una de
ellas, entrando en el apartado técnico.
- **Capítulo 6. Pruebas:** se muestran el conjunto de pruebas, que se han desarrollado para
la posible identificación de errores y posibles fallos en su ejecución, todas ellas divididas
en tipos.
- **Capítulo 7. Implementación y mantenimiento:** se explica el paso que se ha realizado para
hacer que el aplicativo sea accesible para los usuarios y las medidas para su mante-
nimiento.
- **Capítulo 8. Manual de uso:** presenta una pequeña demo con el fin de que el usuario se
familiarice con el uso de la solución, mostrando capturas del mismo.
- **Capítulo 9. Conclusión y trabajo Futuro:** contempla las conclusiones que se han ob-
tenido durante la realización del trabajo y las mejoras que se tomarán para siguientes
versiones del mismo.
- **Apéndices:** se muestran anexos relacionados con análisis de tiempos y apartados rela-
cionados con la codificación del la solución y análisis de tiempos.

CAPÍTULO 2

Estudio estratégico

Los nonogramas, junto otros gigantes rompecabezas de «papel y lápiz», tales como: sudoku, crucigramas, hundir la flota, ahorcado... se han adaptado a una era que está gobernada por las nuevas plataformas tecnológicas. Y en cualquier tiempo de ocio se puede propiciar que estos puzzles estén cada vez más presentes en nuestras vidas, fomentando la creatividad y un entretenimiento productivo y hasta didáctico.

2.1 Nonogramas en la era Digital

Pese a que la era Digital ofrece un amplio abanico de medios o plataformas que han incluido y popularizado estos rompecabezas, en este apartado nos enfocaremos en el de las aplicaciones móviles.

Para que el estudio sea exhaustivo, se explorarán aquellas aplicaciones disponibles en las principales tiendas de aplicaciones para las plataformas *Android* e *iOS*, *Google Play* y *App Store* respectivamente.

2.1.1. Nonograms Katana

Nonograms Katana es una aplicación con una remarcada temática nipona, que permite al usuario resolver una gran variedad de *nonogramas*, de una gran variedad de categorías y dimensiones, como se puede comprobar en la [Figura 2.2a](#).

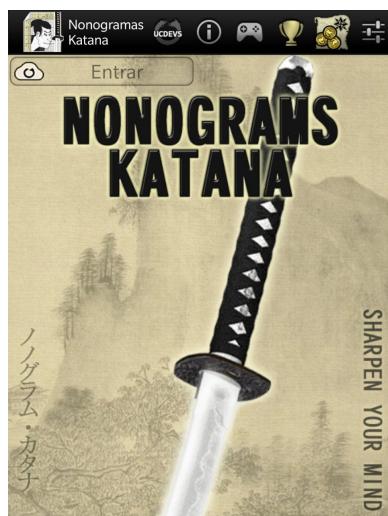


Figura 2.1: Pantalla principal de Nonograms Katana

Además, como se puede apreciar en la **Figura 2.2b**, se incluye la resolución de *nonogramas* a color, en el que mediante un selector de colores el usuario pinta cada una de las celdas, resolviendo de este modo el *nonograma*.

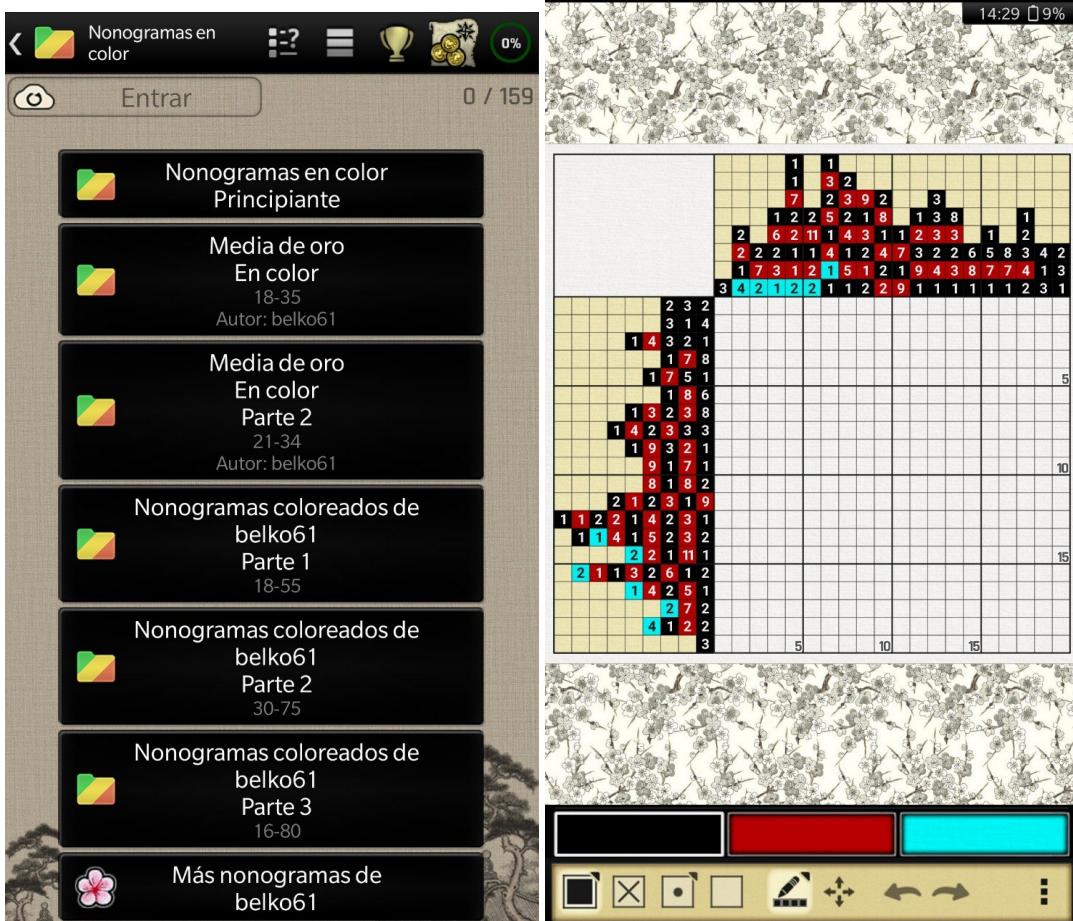


Figura 2.2: Pantallas de Nonograms Katana con modalidad a color.

El aplicativo sigue la corriente clásica y no permite al usuario resolver los *nonogramas* basado en un número determinado de *vidas* o intentos (*disminuyendo su valor al pulsar sobre celdas erróneas*), siendo algo tediosa la experiencia de juego, ya que puede acarrear fallos durante su resolución.

Una característica notable de la aplicación es la de permitir al usuario crear sus propios *nonogramas*, compartirlos con la comunidad, y resolver los de otros usuarios. Sin embargo, esta propiedad no es su principal función y aparece bloqueada si no estás registrado, además de estar limitada por restricciones como los de la **Figura 2.3**.

El aplicativo presenta la propiedad de multilenguaje, no obstante, este presenta errores en sus traducciones, como se puede comprobar en la figura anteriormente citada.



Figura 2.3: Modal de restricción en publicación en Nonograms Katana

2.1.2. Nonogram.com - Picture cross number puzzle

Una de las aplicaciones más descargadas dentro de la categoría puzzle con más de diez millones de descargas en la tienda *Google Play*, presenta una interfaz amigable y limpia.

Hace un buen uso de animaciones, destacando la experiencia de juego, algunas bastante remarcables como cuando la que se muestra en la [Figura 2.4b](#), en cuanto finalizas un nivel.

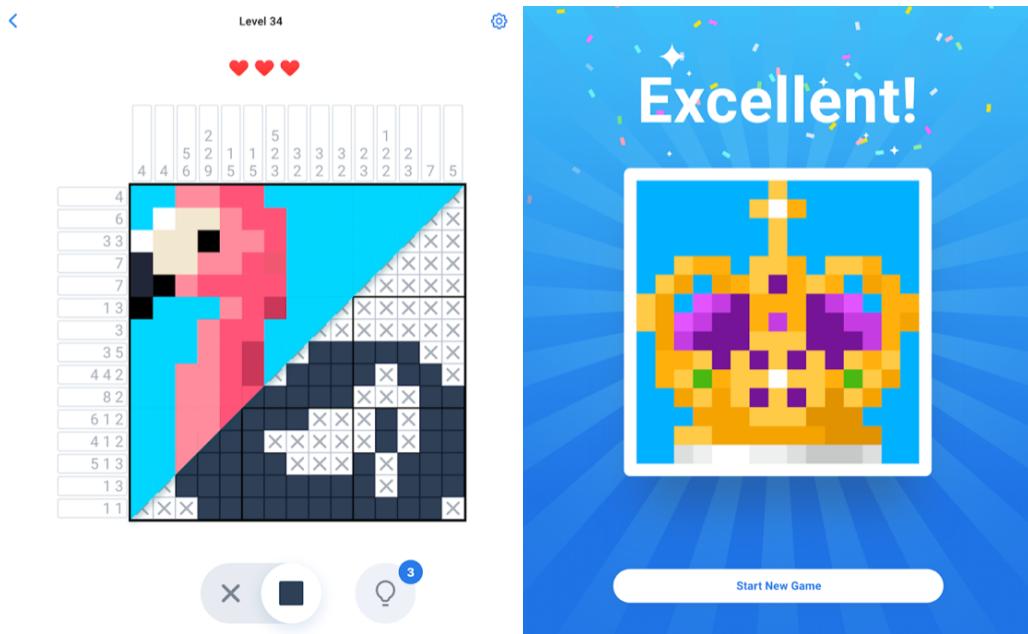


Figura 2.4: Pantallas de Nonogram.com

Como característica extra de juego, como se visualiza en la [Figura 2.4a](#), la barra inferior de juego incorpora un botón llamado *Pista*, con el que después de ser seleccionado, el usuario puede clicar sobre una celda determinada y descubrir si es correcta sin restar una vida, teniendo limitada esta opción a tres usos por nivel.

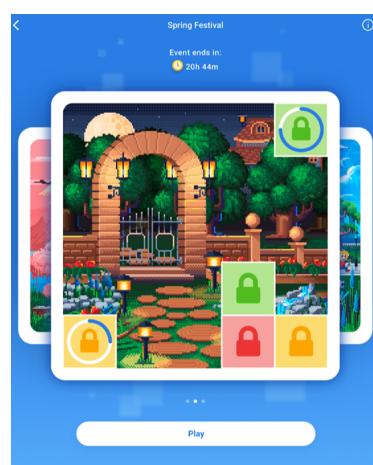


Figura 2.5: Pantalla de Evento Primavera de Nonogram.com

En el aplicativo, de forma recurrente, aparecen *Eventos* en los que el usuario puede resolver un conjunto de *nonogramas* especiales relacionados con una temática concreta, representada en la [Figura 2.5](#).

La solución, como se ha podido comprobar, es de las más completas de las disponibles, no obstante, incluye publicidad excesivamente intrusiva para el usuario, presente en casi todas sus funcionalidades, que entorpecen la experiencia de juego, incluso llegando a entorpecer al jugador.

2.1.3. Nono Infinite

Este producto destaca por su interfaz *arcade*, con una clara intención de ser dirigida para todos los públicos, constraintando colores muy vivos, con formas que recuerdan mucho a juegos clásicos para niños, se puede ver reflejado en [Figura 2.6](#).



Figura 2.6: Pantallas de selección de nivel y juego de Nono Infinite

Como peculiaridad, presenta *nonogramas* de dimensiones poco usuales que difieren mucho de los clásicos, como el 5x10 presente en la [Figura 2.6b](#). Así mismo, permite cambiar la dificultad de los niveles manteniendo las dimensiones del mismo, sin embargo, esta puede parecer un poco "artificial" ya que únicamente aumenta las distancias de las celdas correctas.

Resulta interesante cómo la solución aprovecha el apartado del tutorial, para habituar al usuario de forma interactiva con las reglas clásicas de los *nonogramas*, controles del mismo y consejos más avanzados, sobre todo para niveles más complejos ([Figura 2.7](#)).

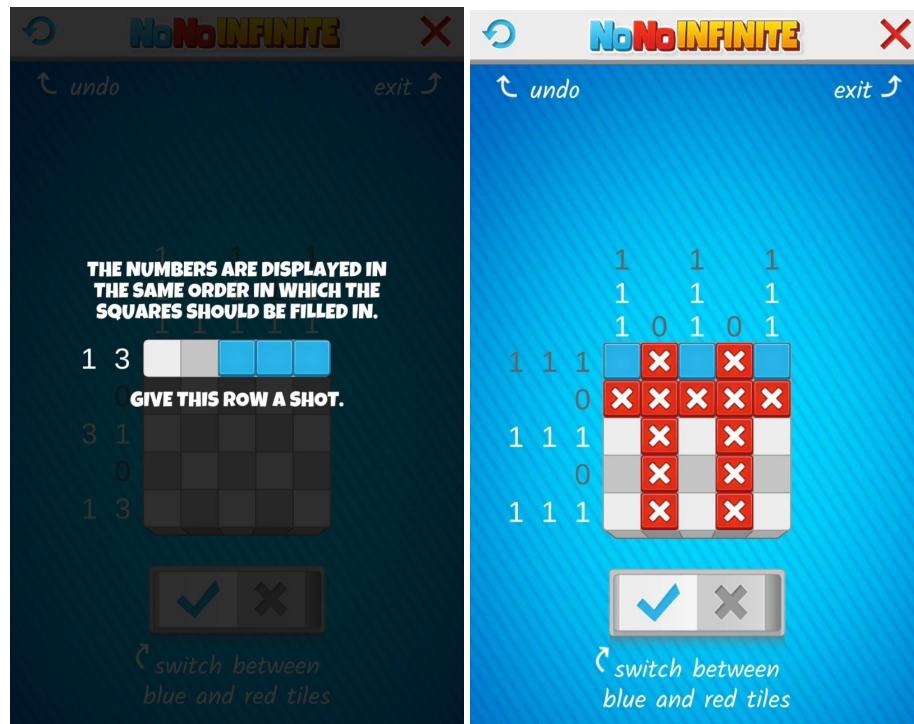


Figura 2.7: Pantallas de tutorial en Nono Infinite

2.1.4. Family Crest Nonogram

Family Crest Nonogram es un juego que, pese a que no destaque por su apartado gráfico e interfaz de usuario, presenta muchas características que lo diferencian del resto de soluciones del género.

Lo más remarcable de la aplicación es que tenga establecido su uso en modo *landscape* (apaisado), adaptándose al formato de la mayoría de géneros de juegos de móvil. Sin embargo, este puede ser un punto negativo para muchos, ya que priva al jugador la esencia de estar jugando a un pasatiempo, recordándolo más a la de un *videojuego*.



Figura 2.8: Pantallas de Family Crest

No obstante, su experiencia de juego es muy notable, ya que presenta una *cruceta* (para moverse por cada una de las celdas) y botones físicos de acción que facilitan un mayor control, Figura 2.8b, además de incorporar una música ambiente que acompaña al usuario cada vez que está jugando un nivel.

2.2 Análisis de las aplicaciones

Al analizar las aplicaciones enfocadas al submundo de los *nonogramas*, vemos que siguen un patrón de similitudes, que resultan sustanciales para el desarrollo de una solución del género rompecabezas, además de ciertas peculiaridades que parecen interesantes para su inclusión en la solución. Es importante también remarcar y apartar aquellas características que puedan ser perjudiciales para una primera versión de la solución. A continuación, se muestra un tabla representativa de cada una de las características estudiadas, donde se objetarán su final inclusión en el aplicativo:

Tabla 2.1: Comparativa de características entre aplicaciones de interés y su inclusión como requisito

Característica encontrada	Nonogram Katana	Nonograma.com Picture cross	Nono Infinite	Family Crest	Incluido como requisito
Aplicación multiplataforma	✓	✓	✗	✓	✓
Temática especial	✓	✗	✓	✓	✓
Creación nonogramas	✓*	✗	✗	✗	✓
Opción de autoguardado	✓	✓	✗	✗	✓*
Registro de usuario	✓	✗	✗	✗	✗
Sincronización de niveles en nube	✓	✗	✗	✗	✓
Juego con <i>vidas</i>	✗	✓	✗	✗	✓*
Música ambiente	✗	✗	✗	✓	✗
Botón de pistas	✗	✓	✗	✗	✗
Botones físicos de acción	✓	✓	✓	✓	✗
Variedad de visuales	✓	✗	✗	✗	✓
Multi-idioma	✓*	✓	✗	✗	✓
Eventos especiales	✗	✓	✗	✗	✗
Nonogramas a color	✓	✗	✗	✗	✗
Tutorial de juego	✓	✓	✓	✗	✓
Sección Leaderboard	✓	✓	✗	✗	✗
Selector de niveles	✓	✗	✗	✓	✓

La inclusión de las características de la [Tabla 2.1](#) se ven indicadas mediante la siguiente simbología:

1. ✓: Característica incluida en el aplicativo.
2. ✗: Característica no incluida en el aplicativo.
3. ✓*: Característica incluida en el aplicativo con ciertos matices.

2.3 Propuesta

Visualizando las características reflejadas en la [Tabla 2.1](#) sacamos las siguientes conclusiones, de cara a la creación de la primera versión de la aplicación como producto:

- El aplicativo funcionará para ambas plataformas, *Android* e *iOS*; ya que la mayoría de apps estudiadas son multiplataforma.
- Seguirá una temática especial con el fin de diferenciarlas del resto, además de ir alternando entre temas diversos, cambiando la interfaz de usuario.
- Tanto la resolución como la creación de los nonogramas serán funciones de la solución, sin ningún tipo de restricción.
- El usuario podrá usar *servicios in-cloud* como: sincronización en nube o acceso a la base de datos sin tener que registrarse.
- La aplicación tendrá posibilidad multi-idioma, teniendo disponible inglés y español para esta primera versión, sin ningún tipo de errata.
- Se podrán jugar niveles de *nonogramas* de diferentes dimensiones clásicas, recordando a los de los medios físicos tradicionales, previamente explicado su uso por un tutorial intuitivo.
- Durante la resolución del *nonograma*, el usuario podrá resolverlo con una interfaz minimalista, en ausencia de botones físicos, además de poder alternar entre resolución con y sin *vidas*.
- Muchas de las características estudiadas se han tomado como no esenciales y quedarán descartadas para la primera versión.

CAPÍTULO 3

Análisis del problema

“La Ingeniería de Requisitos (IR) es el área más importante de la Ingeniería de Software y posiblemente de todo el ciclo de vida de una solución software (SDLC)” [5]. Esta etapa es la responsable de que los requisitos recién detectados, aún incompletos e imprecisos, se transformen en especificaciones formales del aplicativo final.

3.1 Especificación de requisitos

Para llevar a cabo la especificación total de requisitos se seguirá la norma tradicional establecida por el estándar internacional IEEE Std 830-1998 [10], elegido por una gran mayoría de jefes de departamentos software por su gran agilidad en la fase de gestión de requisitos [8].

A continuación, de acuerdo a la normativa ISO elegida, se mostrarán los contenidos acompañados por diagramas y buenas prácticas.

3.1.1. Propósito

El propósito de esta sección es la de definir y formalizar los requerimientos que debe incorporar el *MVP* del aplicativo, facilitando y guiando el desarrollo del mismo.

3.1.2. Ámbito

El ámbito, como se ha comentado en capítulos anteriores, es el de las aplicaciones móviles disponibles en plataformas *iOS* y *Android*.

El aplicativo en su versión *MVP* adoptará el nombre provisional de *NonoChallenge*, compuesto por el juego de palabras: *Nonograma* junto con el término anglosajón *Challenge* (reto). En el cual, el usuario hará uso de sus servicios propios y *en nube* tales como inicios de sesión, interacción con base de datos y sincronización.

3.1.3. Terminología

Los términos relacionados con la ontología del sistema se ven enumerados y descritos por la [Tabla 3.1](#).

Tabla 3.1: Glosario de términos ontológicos del aplicativo

Término	Descripción
Usuario	Persona que hará el uso del conjunto de funcionalidades del aplicativo final
Nonograma	Rompecabezas de MxN dimensiones
Nivel	Nonograma a crear o resolver por el Usuario
Progreso	Datos relacionados con la persistencia de la resolución de un nivel

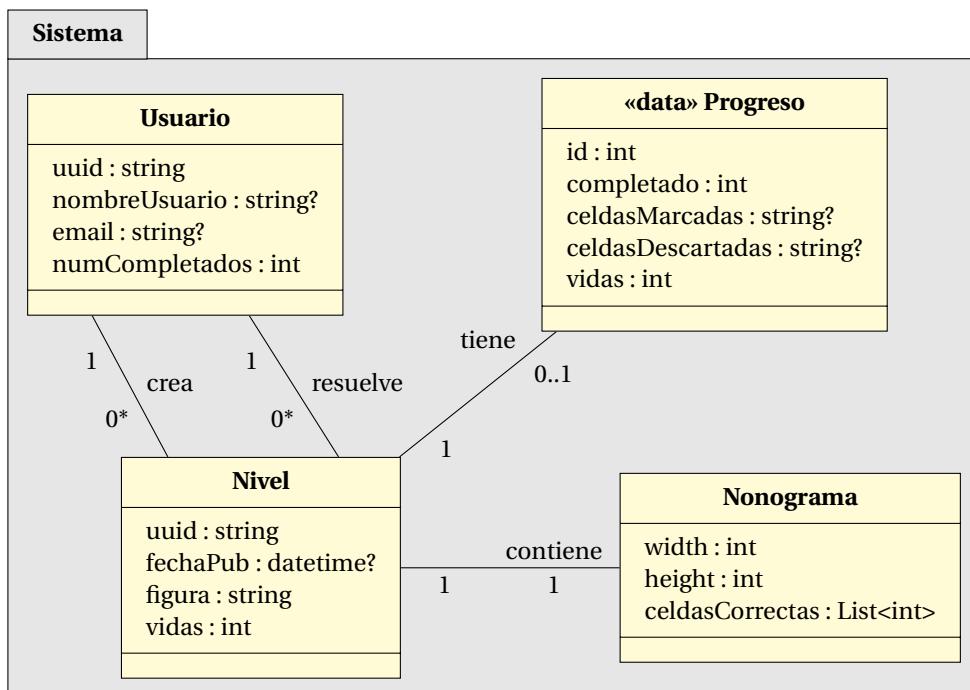
Por otra parte, los términos técnicos que componen el sistema son los que siguen:

Tabla 3.2: Glosario de términos técnicos del aplicativo

Término	Descripción
Nombre de usuario	Nombre que adoptará el Usuario de forma opcional para su identificación en el aplicativo.
Email	Cuenta de correo que hará uso el usuario para acceder a los servicios <i>en nube</i>
Fecha de publicación	Fecha en la que el usuario crea y publica un nivel
Figura	Nombre identificativo de un nonograma
Vidas	Número de intentos en la resolución de un nivel

3.1.4. Modelo de Dominio

Una vez introducida la terminología propia del sistema, para un mayor entendimiento del contexto del mismo, se representa un diagrama de clases de acuerdo a las reglas clásicas UML, como se puede visualizar en la [Figura 3.1](#).

**Figura 3.1:** Diagrama del modelo de dominio

A continuación, se comenta la función de cada una de las clases conceptuales y se enumeran en la **Tabla 3.3** y **Tabla 3.6** los atributos que las componen. (*La anotación ? al lado del tipo refleja su posibilidad de nulidad en el sistema.*)

- **Clase Usuario:** Conjunto de datos del usuario relacionados con el sistema.

Tabla 3.3: Atributos de la clase Usuario

Atributos de Usuario	Descripción
uuid <i>string</i>	Cadena de caracteres único que identifica al usuario a nivel interno
nombre de Usuario <i>string?</i>	Pseudónimo único a elegir por el usuario en el sistema
email <i>string?</i>	Cuenta de correo de registro única para el usuario
numCompletados <i>int</i>	Número de niveles completados por el usuario

- **Clase Nivel:** Muestra las características de un determinado nivel a resolver.

Tabla 3.4: Atributos de la clase Nivel

Atributos de Nivel	Descripción
uuid <i>string</i>	Cadena de caracteres único que identifica al nivel
fecha de Publicación <i>dateTime?</i>	Fecha de creación del nonograma en caso de llegar a publicarse
figura <i>string</i>	Nombre del nivel mostrado una vez resuelto
vidas <i>int</i>	Número de intentos que dispone un usuario para la resolución de un nivel

- **Clase Progreso:** Clase de persistencia que contiene los datos durante la resolución de un nivel.

Tabla 3.5: Atributos de la clase Progreso

Atributos de Progreso	Descripción
id <i>int</i>	Número de identificación del progreso de un determinado nivel
completado <i>int</i>	Indica con un 1 si el nivel está resuelto y 0 si no lo está (simulando un dato de tipo booleano)
celdasMarcadas <i>string</i>	Números de las celdas pintadas separados por delimitadores (simulando un dato de tipo lista de enteros)
celdasDescartadas <i>string</i>	Números de las celdas descartadas separados por delimitadores (simulando un dato de tipo lista de enteros)
vidas <i>int</i>	Números de vidas que dispone el usuario en ese momento

- **Clase Nonograma:** Contiene los datos del nonograma de un determinado nivel.

Tabla 3.6: Atributos de la clase Nonograma

Atributos de Nonograma	Descripción
width <i>int</i>	Número de filas de celdas que compone el Nonograma
height <i>int</i>	Número de columnas de celdas que compone el Nonograma
celdasCorrectas <i>List<int></i>	Lista de los números de celdas que componen la solución del Nonograma

3.1.5. Límites del Sistema

Para representar los límites de la aplicación, se emplea un diagrama de contexto en el que se muestra la jerarquía de los actores que van a interactúan con el aplicativo.

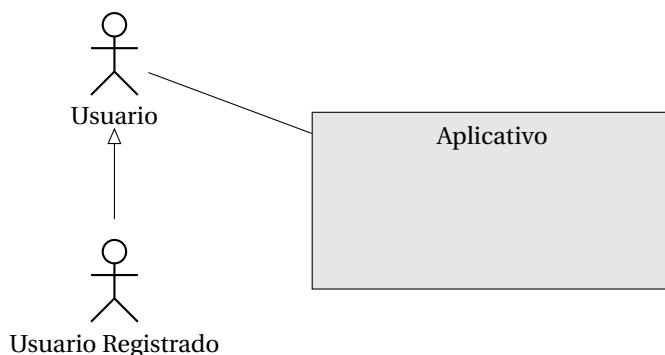


Figura 3.2: Diagrama de contexto del aplicativo

Como se puede apreciar en el diagrama de la [Figura 3.2](#), únicamente van a interactuar dos actores con el sistema: i) el usuario sin registro, que emplea únicamente las funciones locales del sistema y ii) el usuario registrado, que podrá emplear tanto las funciones locales como *en nube* del aplicativo.

3.1.6. Restricciones del Sistema

En este apartado se muestran algunas de las restricciones que pueden impedir algunas de las funcionalidades que ofrece el sistema al usuario, vistos en la [Tabla 3.7](#).

Tabla 3.7: Restricciones del sistema

Restricción	Descripción
Memoria del dispositivo	Posibilidad de que el dispositivo no tenga suficiente memoria disponible para guardar el progreso de la resolución de los niveles. <i>Probabilidad prácticamente despreciable</i>
Conexión a internet	Las funcionalidades <i>en nube</i> requieren de conexión a internet para poder llegar a usarse.
Cuenta de Google	El usuario puede carecer de una cuenta de correo de Google necesaria para hacer uso de las funcionalidades <i>en nube</i> .

3.1.7. Características del Sistema

Una vez vistas las limitaciones y restricciones del sistema, se pueden analizar las características finales a incorporar en la aplicación, mediante *casos de uso*.

3.1.7.1. Casos de uso principales

Estos, son los que se encontrarán en la *Pantalla Principal* y *Pantalla de Selección de Nonogramas* del aplicativo.

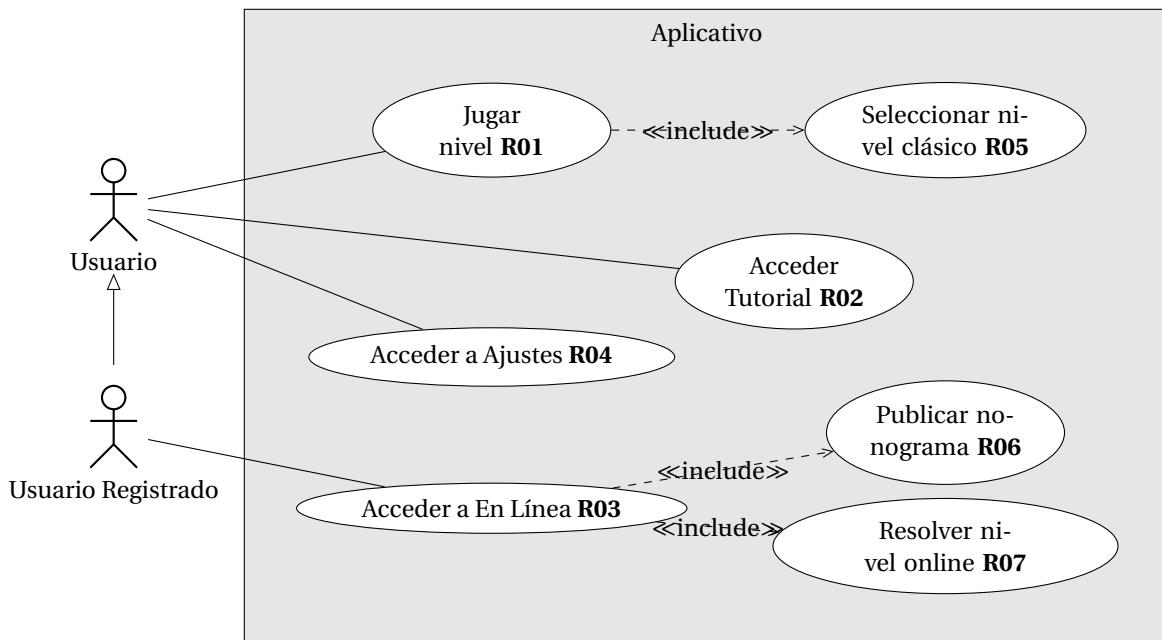


Figura 3.3: Diagrama de casos de usos principales

3.1.7.2. Casos de uso de resolución de nivel

Estarán disponibles en la pantalla de resolución de nivel.

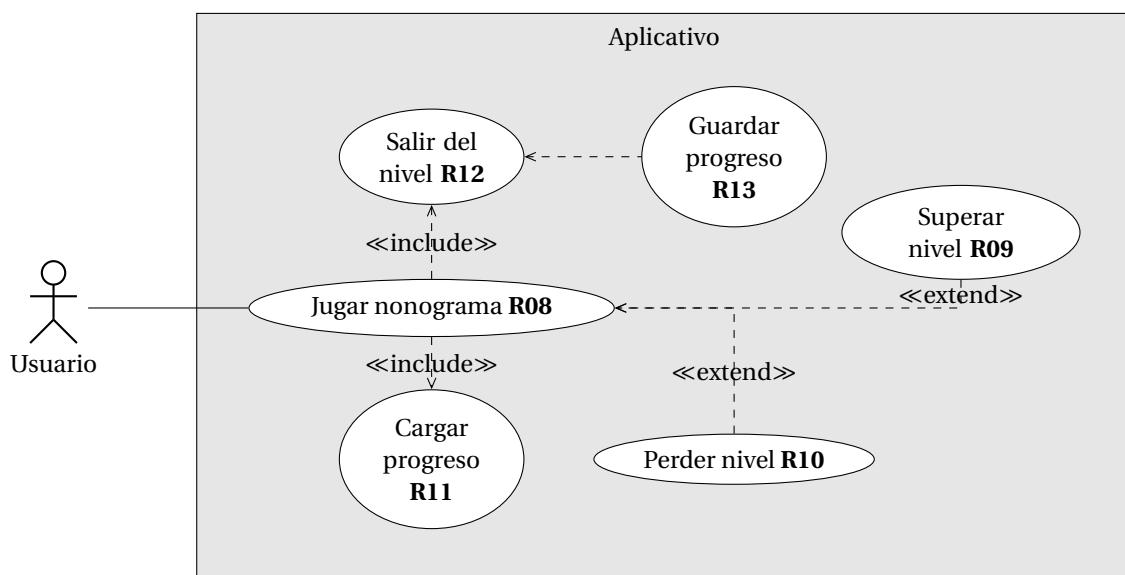


Figura 3.4: Diagrama de casos de usos en pantalla de resolución de nivel

3.1.7.3. Casos de uso de ajustes

Casos de uso visibles en la *Pantalla de Ajustes* del sistema.

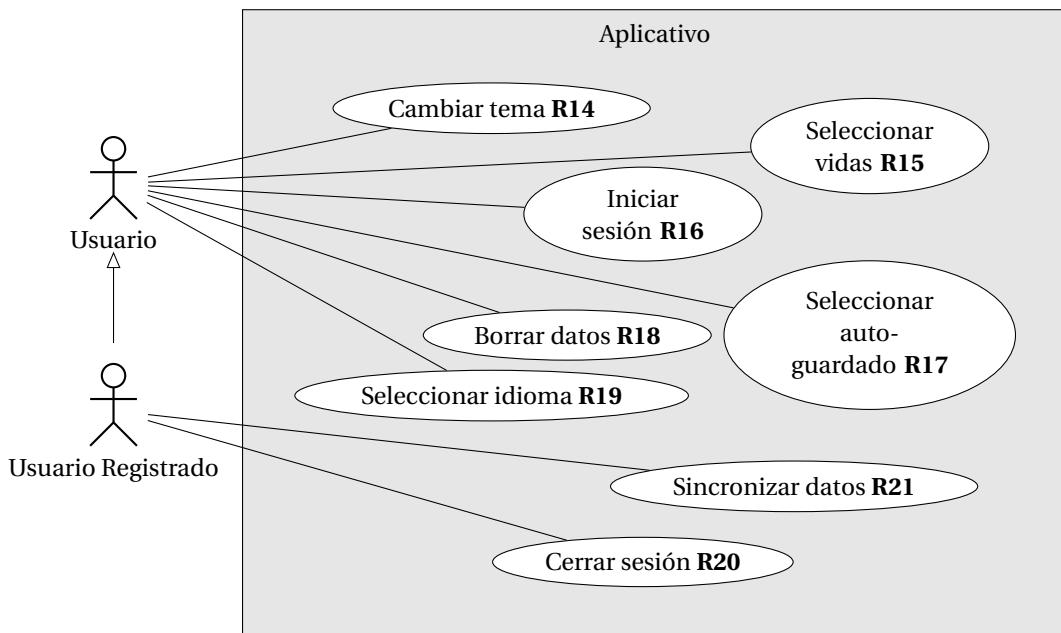


Figura 3.5: Diagrama de casos de ajustes

Cada uno de los requisitos funcionales (RF) han sido numerados con un identificador **RXX**, y se harán referencia a ellos en su especificación.

3.1.8. Requisitos funcionales

Para evitar posibles inconsistencias a nivel de producto, se especificarán cada uno de los requisitos funcionales previamente numerados, enumerando sus datos de entrada y salida, precondiciones, postcondiciones, flujos y criterios de aceptación.

Actor: Usuario	R1 - Jugar nivel
Descripción	Acceder al listado de niveles clásicos disponibles en el sistema
Precondiciones	Ninguna
Entradas	Ninguna
Salidas	Listado de sensores predeterminados clásicos
Postcondiciones	Se mostrará el listado de niveles
Flujo principal	1. Presionar la sección Jugar

Tabla 3.8: Tabla de requisito funcional *Jugar nivel*

Actor: Usuario	R2 - Visualizar Tutorial
Descripción	Muestra el tutorial para familiarizar al usuario de las reglas de la resolución de nonogramas
Precondiciones	Ninguna
Entradas	Ninguna
Salidas	Listado de reglas y consejos
Postcondiciones	Se mostrarán los consejos y reglas
Flujo principal	<ol style="list-style-type: none"> 1. Presionar la sección Tutorial 2. Deslizar el conjunto de consejos y reglas

Tabla 3.9: Tabla de requisito funcional *Visualizar Tutorial*

Actor: Usuario Registrado	R3 - Acceder sección En Línea
Descripción	Muestra el conjunto de funciones en red
Precondiciones	El usuario debe estar registrado
Entradas	Número identificativo del usuario registrado
Salidas	Listado de funcionalidades en línea
Postcondiciones	Se mostrará el listado de funcionalidades
Flujo principal	<ol style="list-style-type: none"> 1. Presionar la sección En Línea

Tabla 3.10: Tabla de requisito funcional *Acceder sección En Línea*

Actor: Usuario	R4 - Acceder sección Ajustes
Descripción	Muestra el conjunto de configuraciones propias del sistema
Precondiciones	Ninguna
Entradas	Ninguna
Salidas	Listado de configuraciones
Postcondiciones	Se mostrará el listado de configuraciones
Flujo principal	<ol style="list-style-type: none"> 1. Presionar la sección Ajustes

Tabla 3.11: Tabla de requisito funcional *Acceder sección Ajustes*

Actor: Usuario	R5 - Seleccionar nivel clásico
Descripción	Permite seleccionar un nivel predeterminado del aplicativo
Precondiciones	Ninguna
Entradas	Ninguna
Salidas	Listado de niveles predeterminados
Postcondiciones	Se mostrará el listado de niveles
Flujo principal	<ol style="list-style-type: none"> 1. Seleccionar un nivel del listado

Tabla 3.12: Tabla de requisito funcional *Seleccionar nivel clásico*

Actor: Usuario Registrado	R6 - Publicar nonograma
Descripción	Permite crear un nonograma con características específicas
Precondiciones	El usuario debe estar registrado
Entradas	Nombre del nonograma, dimensiones, celdas correctas y nombre del autor
Salidas	Nonograma creado y disponible en el listado de nonogramas en línea
Postcondiciones	Se mostrará el listado de niveles
Flujo principal	<ol style="list-style-type: none"> 1. Seleccionar la sección Crear nonograma 2. Establecer nombre y dimensiones del nonograma y autor 3. Seleccionar cada una de las celdas indicando que son correctas

Tabla 3.13: Tabla de requisito funcional *Publicar nonograma*

Actor: Usuario Registrado	R7 - Resolver nivel online
Descripción	Permite descargar y resolver un nivel creado por el mismo usuario o por la comunidad
Precondiciones	El usuario debe estar registrado
Entradas	Nonograma seleccionado
Salidas	Nonograma descargado
Postcondiciones	Se mostrará el nivel online a resolver
Flujo principal	<ol style="list-style-type: none"> 1. Seleccionar un nivel del listado de niveles online

Tabla 3.14: Tabla de requisito funcional *Resolver nivel online*

Actor: Usuario	R8 - Jugar nonograma
Descripción	Muestra la pantalla y permite la resolución del nonograma de forma interactiva
Precondiciones	Se debe haber seleccionado un nivel
Entradas	Nivel seleccionado
Salidas	Nonograma a resolver
Postcondiciones	Se mostrará el nonograma a resolver
Flujo principal	<ol style="list-style-type: none"> 1. Pulsar en las celdas para descubrir las celdas correctas y resolver el nonograma 2. Pulsar dos veces o mantener pulsado en las celdas para indicar la celda como no correcta

Tabla 3.15: Tabla de requisito funcional *Jugar nonograma*

Actor: Usuario	R9 - Superar nivel
Descripción	Muestra una pantalla de felicitación una vez resuelto el nonograma
Precondiciones	Todas las celdas correctas del nonograma se deben haber seleccionado y se debe disponer de vidas
Entradas	Nivel seleccionado, celdas correctas y vidas
Salidas	Nonograma descubierta
Postcondiciones	Se mostrará una pantalla de felicitación y la información del nonograma resuelto
Flujo principal	<ol style="list-style-type: none"> 1. Pulsar sobre cada una de las celdas correctas del nonograma superando el nivel

Tabla 3.16: Tabla de requisito funcional *Superar nivel*

Actor: Usuario	R10 - Perder nivel
Descripción	Muestra una pantalla indicando que has fallado en la resolución del nivel y permite intentarlo de nuevo
Precondiciones	El número de vidas especificado en ajustes debe ser finito y durante el nivel el usuario debe haberse quedado sin vidas
Entradas	Número de vidas
Salidas	Opción intentar de nuevo
Postcondiciones	Se mostrará una pantalla de repetir nivel
Flujo principal	<ol style="list-style-type: none"> 1. Pulsar sobre celdas incorrectas hasta quedarse sin vidas

Tabla 3.17: Tabla de requisito funcional *Perder nivel*

Actor: Usuario	R11 - Cargar progreso
Descripción	Muestra un pop-up permitiendo al usuario poder cargar el progreso de la resolución del nivel
Precondiciones	El nivel seleccionado para su resolución debe de haberse jugado anteriormente y tener un progreso
Entradas	Nivel específico y progreso del mismo
Salidas	Último progreso de la resolución del nivel seleccionado
Postcondiciones	Se mostrará la pantalla de resolución del nivel con el último progreso guardado
Flujo principal	<ol style="list-style-type: none"> 1. Pulsar sobre la opción cargar progreso en un nivel concreto

Tabla 3.18: Tabla de requisito funcional *Cargar progreso*

Actor: Usuario	R12 - Salir del nivel
Descripción	Permite al usuario salir del nivel
Precondiciones	El usuario debe de encontrarse en la pantalla de resolución de un nivel
Entradas	Ninguna
Salidas	Ninguna
Postcondiciones	Se mostrará el listado de niveles
Flujo principal	<ol style="list-style-type: none"> 1. Pulsar sobre la opción salir del nivel

Tabla 3.19: Tabla de requisito funcional *Salir del nivel*

Actor: Usuario	R13 - Guardar progreso
Descripción	Permite al usuario guardar su progreso durante la resolución de un nivel
Precondiciones	El usuario debe de encontrarse en la pantalla de resolución de un nivel y debe de haberse habilitado la opción Autoguardado en ajustes
Entradas	Ninguna
Salidas	Ninguna
Postcondiciones	Se guardará el progreso del nivel
Flujo principal	<ol style="list-style-type: none"> 1. Salir del nivel teniendo habilitada la opción de Autoguardado habilitada

Tabla 3.20: Tabla de requisito funcional *Guardar progreso*

Actor: Usuario	R14 - Cambiar tema
Descripción	Concede al usuario la opción de elegir un tema visual dentro de los disponibles del sistema
Precondiciones	Ninguna
Entradas	Tema a elegir
Salidas	Ninguna
Postcondiciones	Se actualizará el tema global del sistema
Flujo principal	<ol style="list-style-type: none"> 1. Seleccionar el tema a elegir mediante el botón Cambiar, eligiendo de forma dinámica la combinación de colores

Tabla 3.21: Tabla de requisito funcional *Cambiar tema*

Actor: Usuario	R15 - Seleccionar vidas
Descripción	Concede al usuario la opción de elegir el número de vidas disponibles durante la resolución de un nivel
Precondiciones	Ninguna
Entradas	Número de vidas a elegir
Salidas	Ninguna
Postcondiciones	Se actualizará el número de vidas por defecto en la resolución de un nivel
Flujo principal	<ol style="list-style-type: none"> 1. Seleccionar el número de vidas mediante el botón cambiar (de 1 a 5 vidas) o infinitas.

Tabla 3.22: Tabla de requisito funcional *Seleccionar vidas*

Actor: Usuario	R16 - Iniciar sesión
Descripción	Permite al usuario registrarse en el caso de no estar registrado mediante una cuenta de Google e identificarse dentro del aplicativo para acceder a las funciones online
Precondiciones	Ninguna
Entradas	Credenciales de Google
Salidas	Ninguna
Postcondiciones	Se actualizará la cuenta que ha iniciado sesión el usuario a nivel de sistema
Flujo principal	<ol style="list-style-type: none"> 1. Seleccionar la opción de Ingresar en Cuenta de usuario 2. Presionar Iniciar sesión con Google

Tabla 3.23: Tabla de requisito funcional *Iniciar sesión*

Actor: Usuario	R17 - Seleccionar autoguardado
Descripción	Permite al usuario elegir habilitar o deshabilitar la opción de autoguardado en la resolución de niveles
Precondiciones	Ninguna
Entradas	Ninguna
Salidas	Ninguna
Postcondiciones	Se actualizará la opción de autoguardado en las preferencias del sistema
Flujo principal	<ol style="list-style-type: none"> 1. Habilitar o deshabilitar la opción de autoguardado

Tabla 3.24: Tabla de requisito funcional *Seleccionar autoguardado*

Actor: Usuario	R18 - Borrar datos
Descripción	Borra el progreso de todos los niveles jugados
Precondiciones	Ninguna
Entradas	Ninguna
Salidas	Ninguna
Postcondiciones	Se debe de perder el progreso de todos los niveles jugados
Flujo principal	<ul style="list-style-type: none"> 1. Presionar la opción borrar en la sección borrar progreso

Tabla 3.25: Tabla de requisito funcional *Borrar datos*

Actor: Usuario	R19 - Seleccionar idioma
Descripción	Cambia el idioma global del aplicativo
Precondiciones	Ninguna
Entradas	Ninguna
Salidas	Ninguna
Postcondiciones	Se debe de perder el progreso de todos los niveles jugados
Flujo principal	<ul style="list-style-type: none"> 1. Presionar la opción cambiar, seleccionando el idioma deseado

Tabla 3.26: Tabla de requisito funcional *Seleccionar idioma*

Actor: Usuario Registrado	R20 - Cerrar sesión
Descripción	Cierra sesión del usuario, cambiando los datos del jugador a nivel global del aplicativo
Precondiciones	El usuario debe estar registrado
Entradas	Ninguna
Salidas	Ninguna
Postcondiciones	Mostrar al usuario que su sesión ha sido finalizada
Flujo principal	<ul style="list-style-type: none"> 1. Presionar en Ingresar en el apartado Cuenta de usuario 2. Presionar en Finalizar sesión con Google

Tabla 3.27: Tabla de requisito funcional *Cerrar sesión*

Actor: Usuario	R21 - Sincronizar datos
Descripción	Guarda el progreso de los niveles clásicos mediante el servicio en nube
Precondiciones	El usuario debe estar registrado
Entradas	Ninguna
Salidas	Ninguna
Postcondiciones	Mostrar un popup indicando que los datos han sido sincronizados
Flujo principal	<ol style="list-style-type: none">1. Presionar la Sincronizar

Tabla 3.28: Tabla de requisito funcional *Sincronizar datos*

CAPÍTULO 4

Diseño de la solución

“El desarrollo de una aplicación multiplataforma es una opción cada vez más a tener en cuenta dentro del ámbito del desarrollo móvil” [3]. La mayor parte de desarrollos se van a querer destinar a las principales plataformas móviles, con el fin de llegar al máximo número de usuarios posibles.

4.1 Tecnología empleada

Como se había decidido en el segundo capítulo, la realización de este *MVP* irá dirigido para las principales plataformas móviles: *Android* e *iOS*. El desarrollo de una aplicación *nativa* para cada sistema, evidentemente, no sería factible, ya que implicaría el doble de esfuerzo en la codificación, testing y mantenimiento del producto [9].

Por ello, es necesario el uso de un *framework* de ámbito multiplataforma que nos proporcione un soporte para ambas plataformas. En este caso, se ha optado por la prometedora herramienta de desarrollo impulsada por *Google*: *Flutter*.

4.1.1. Flutter y Dart

Flutter junto a su lenguaje de programación principal, *Dart*, buscan aproximar al usuario la apariencia y rendimiento propio al de un desarrollo *nativo*, aprovechando todas las ventajas que proporciona un *framework multiplataforma* [16].

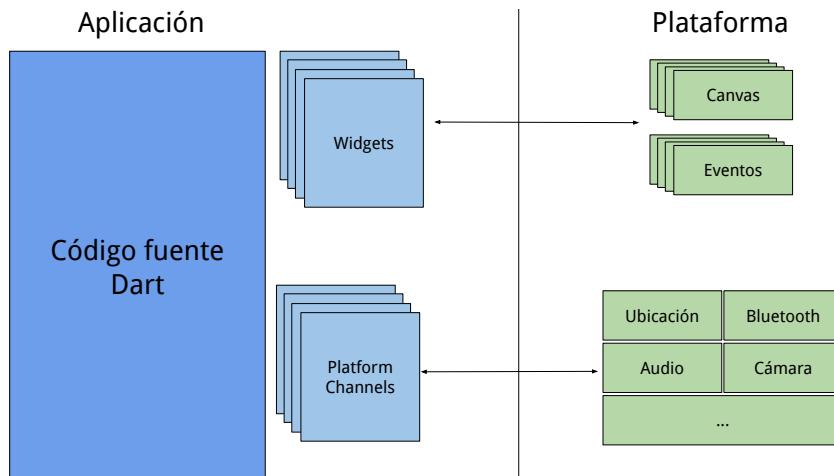


Figura 4.1: Diagrama de renderización de *Flutter*[17]

Esta relación *apariencia/rendimiento*, la consigue *Flutter* de forma exitosa *renderizando* directamente sus componentes, *widgets*, en un *canvas*, permitiendo visualizar el mismo componente en las diferentes plataformas.

Del mismo modo, realiza la conversión directa de servicios propios, *Platform Channels*, en librerías nativas que interactúan con las funcionalidades propias del dispositivo [17], tal y como se puede visualizar en la [Figura 4.1](#).

Esta es una de las razones principales por las que se ha elegido el uso de esta herramienta, frente a otros *frameworks* de desarrollo móvil como:

- *Frameworks de desarrollo móvil basados en Javascript* como: *React Native* y *Vue Native*, que necesitan de una capa adicional para realizar la conversión a *componentes nativos*, pudiendo afectar al rendimiento.
- Los *frameworks embebidos* como *Ionic* que emplea un *WebView* para mostrar los componentes, dando la sensación de que no estás ejecutando una aplicación móvil, sino un sitio web embebido.

4.1.2. Firebase

Para todas las funcionalidades relacionadas con servicios *en nube*, vistos en la fase de Análisis, se empleará la plataforma de desarrollo backend impulsada y recomendada por *Google*: *Firebase*.

Gracias a que *Firebase* presenta una base de datos del tipo *real-time*, se permitirá la sincronización de los datos durante la ejecución del aplicativo [12]. Esta característica resulta ideal, ya que *Flutter*, al tratarse de un *framework "declarativo"* no requiere de eventos para sincronizar los datos, permitiendo al usuario visualizar los datos, en todo momento, sincronizados en pantalla, sin necesidad de que el usuario realice un *input de refrescar*.

Resulta muy potente esta propiedad, ya que la han querido adoptar los frameworks de desarrollo móvil nativo principales, con aproximaciones como: *SwiftUI* para aplicaciones *iOS* y *Jetpack Compose* para soluciones *Android*.

Además, esta, al pertenecer al tipo de las *no relacionales*, presenta ciertos beneficios frente a las *relacionales*, a tener en cuenta en la definición de los modelos, tales como flexibilidad, seguridad y rendimiento.

La funcionalidades *in-cloud* del *back-end* de la aplicación final quedarán cubiertas por los apartados mostrados en la [Tabla 4.1](#)

Tabla 4.1: Funcionalidades *en nube* cubiertas por *Firebase*

Funcionalidad	Tecnología
Autentificación	Firebase Authentication , permite al usuario el acceso a las funciones de la plataforma mediante inicio sesión tradicional (cuenta de correo y contraseña) o por RRSS.
Base de datos	Firebase Firestore , permite al usuario interactuar mediante funciones CRUD la base de datos del sistema.
Analíticas	Firebase Analytics , monitoriza la experiencia de usuario y extrae estadísticas de cara al lanzamiento de futuras versiones tales como: porcentaje de usuarios libre de errores y número de usuarios activos.

El diagrama de flujo del aplicativo con los servicios que ofrece la plataforma *Firebase* puede verse representado por el diagrama de la [Figura 4.2](#)

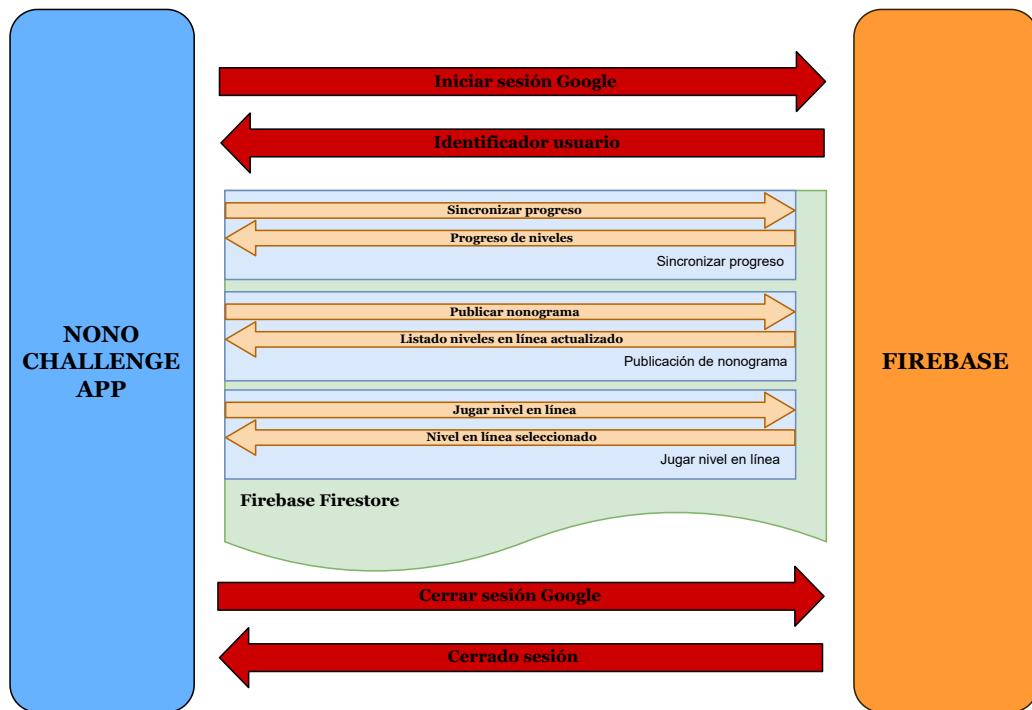


Figura 4.2: Diagrama de flujo con *Firebase*

La gestión del progreso y publicación/resolución de niveles en línea se realizará mediante la función de *Cloud Firestore*, para ello todas las transacciones se realizarán a través del identificador único de usuario.

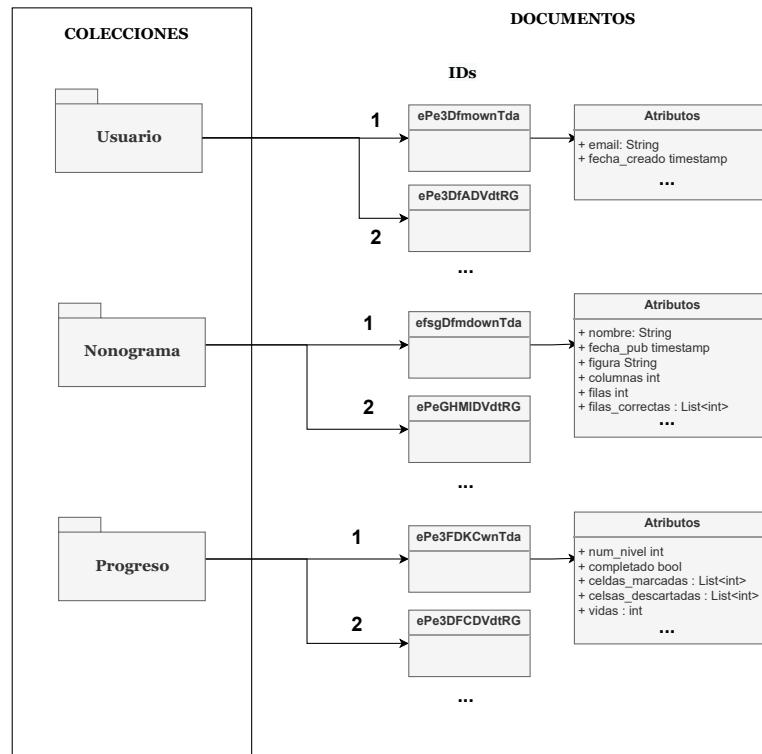


Figura 4.3: Diagrama de estructura de datos de *Firestore*

En cuanto a la estructuración de los datos, al tratarse de una base de datos no relacional, de tipo *NoSql*, se tratarán los datos en forma de *documentos*, cada uno con una clave identificativa única y una serie de atributos. Estos documentos se almacenarán según su tipo en contenedores llamados *colecciones*: *usuarios*, *nonogramas* y *progresos*, en el caso de este aplicativo.

Este modelo de datos, a diferencia de los relacionales, puede ser totalmente flexible y definido por el aplicativo, no obstante, para asegurar una mayor consistencia se ha definido desde el portal web de *Firestore*, [Figura 4.3](#).

4.1.3. Visual Studio Code

Como *IDE* de desarrollo se empleará la herramienta *Visual Studio Code*, desarrollado por *Microsoft*, frente a otros entornos como: *XCode* y *Android Studio*, ya que este dispone de una gran cantidad de *extensiones o plugins*, que facilitan notablemente el proceso de *codificación*, además de estar mejor optimizado para labores de *compilación y depuración*.

4.1.4. Git

Se empleará *Git* para el *control de versiones*, alojando el proyecto en un repositorio en *GitLab*. Además, siguiendo la metodología propias de *GitFlow*, se presentará, como se puede visualizar en la [Figura 4.4](#): i) una rama *master* para versiones finales del aplicativo, ii) una rama *develop* en la que aunar las funcionalidades desarrolladas y iii) las ramas *feature* que se crearán por cada funcionalidad a desarrollar, dividida por todas las capas propuestas por la filosofía *Clean Architecture*, junto a sus *suites de tests*.

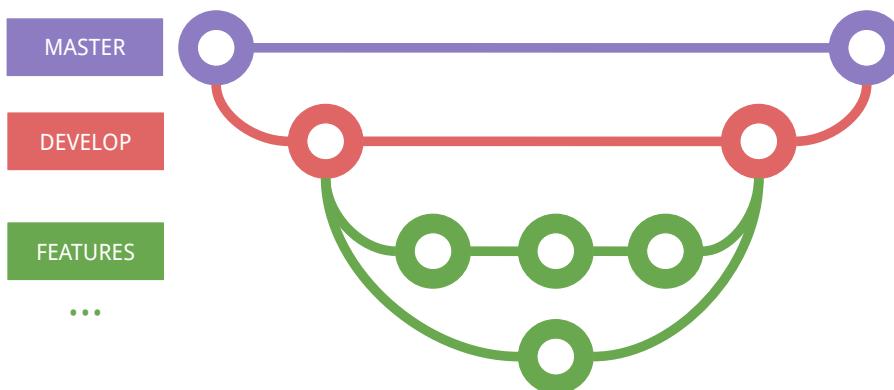


Figura 4.4: Diagrama del funcionamiento de *GitFlow*

Además, nos nutriremos de las buenas prácticas, *librerías y repositorios* disponibles en la plataforma, aprovechando la subida exponencial de la comunidad *Flutter* en este último año, posicionándose dentro de las tecnologías con más *stars* en *sitios web* como *GitHub* [19].

4.2 Arquitectura del Sistema

Una vez definidas las tecnologías que se van a utilizar para el desarrollo de la solución, es importante definir una arquitectura que encaje dentro del abanico de requisitos que se desean incluir, además de las funcionalidades que aportan las tecnologías estudiadas.

Se requiere de una arquitectura de *software*, que no tenga tanta complejidad, ya que no se trata de un proceso industrial y el desarrollo va a ser realizado por una persona, además de

estar bien establecida, ya que no se quiere entorpecer la escalabilidad del sistema, facilitando, de esta forma, las fases de testeo y mantenimiento.

Para un desarrollo con *Flutter* como *framework* de desarrollo *front-end* y *Firebase* como infraestructura de servicios *back-end*, se empleará, *una arquitectura hexagonal*, o como se conoce en *frameworks* como *Flutter*, *Clean Architecture*, una de las más arquitecturas más conocidas y elegidas en entornos de desarrollo móvil [14].

4.2.1. Clean Architecture

El objetivo de esta arquitectura es la de abstraer lo máximo posible los diferentes bloques o componentes que van a conformar el sistema.

Empleando la siguiente arquitectura se pretenderá obtener, a corto y largo plazo, los siguientes beneficios:

- Componentes que desempeñan una única funcionalidad bien definida.
- Mayor escalabilidad, favoreciendo la inclusión de nuevas funcionalidades.
- Facilidad en la identificación de posibles *bugs* o errores relativos a la ejecución del aplicativo.
- Permite la inclusión de *patrones de diseño* en el sistema.
- Código más legible y fácil de entender.

A partir de esta arquitectura es posible dividir el sistema mediante N bloques, para el aplicativo final se ha optado por una arquitectura dividida en cuatro capas, como se puede visualizar en la Figura 4.5.

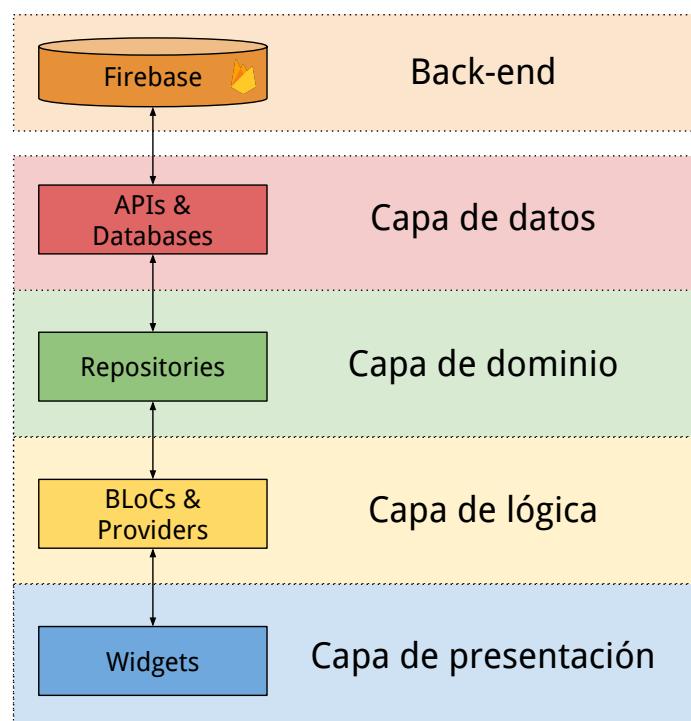


Figura 4.5: Diagrama de *Diagrama Clean Architecture*

A continuación, se comentará la funcionalidad de cada una de las capas, en la [Tabla 4.2](#):

Tabla 4.2: Función de las capas de la arquitectura del sistema

Capa	Funcionalidad
Capa de presentación	Mediante <i>widgets</i> se presentarán todas las vistas y componentes que contiene el aplicativo.
Capa de lógica	A partir de <i>manejadores de estados</i> se controlarán todos los cambios de estado presentes en la capa de <i>presentación</i> .
Capa de dominio	Las interfaces <i>repositorio</i> servirán de esqueleto para las clases relacionadas con la capa de <i>datos</i> .
Capa de datos	Implementando las interfaces de la capa de <i>dominio</i> se obtendrán los datos de fuentes externas como nuestro <i>back-end</i> de <i>Firebase</i>

Adicionalmente, justo a este modelo de capas se adoptará la metodología de desarrollo guiado por pruebas *TDD*, práctica cada vez más vista en proyectos extensos y de gran escalabilidad [4]. La combinación de estas dos praxis permitirán modularizar al máximo las funcionalidades del sistema, haciéndolos más reutilizables y fáciles de monitorizar en las tareas de identificación de errores y rendimiento.

4.2.2. Manejador de estados

Flutter permite *manejar* todos los diferentes *estados* que van surgiendo en el aplicativo, como por ejemplo: *refrescar de forma dinámica una lista de objetos*; directamente en la *capa de presentación*, mediante la función *setState()*[21].

Sin embargo, para nuestra aplicación u otras soluciones de gran envergadura esta tarea puede resultar muy compleja, además de ser totalmente opuesta a la modularidad y reusabilidad que ofrece la arquitectura *Clean Architecture*. Por ello, se hacen necesarios patrones de diseño que actúen como *manejadores de estados*, que estén presentes en la capa de *lógica* y que se encarguen de:

- Controlar todos los eventos presentes en el aplicativo para transformarlos en estados y transmitirlos a la capa de *presentación*.
- Obtener los datos sincronizados transmitidos por la capa de *datos*.

Dentro del gran abanico de librerías e implementaciones que ofrece *Flutter* se ha optado por, como se puede visualizar en la [Figura 4.5](#), la combinación de los siguientes patrones:

- **Patrón BLoC:** empleado para centralizar los cambios de estado, recibiendo *eventos* de la capa de *presentación* y transmitir *estados* que cambien de forma dinámica los componentes de la misma.
- **Provider:** nos permitirá, como su nombre indica, proveer de funcionalidades propias de la capa de lógica, como la lógica de los BLoCs a la capa de *presentación*.

Esta combinación de patrones se incluirá en el aplicativo mediante el paquete *flutter_bloc*. En el capítulo de *Desarrollo de la solución*, se podrá visualizar algunos ejemplos de dicha metodología en el sistema.

CAPÍTULO 5

Desarrollo de la solución

“El desarrollo de aplicaciones móviles resulta desafiante por la gran variedad de dispositivos móviles con diferentes sistemas operativos, características y tamaños” [7]. Esta tarea la suplementa Flutter con sus widgets, permitiendo desarrollar todo tipo de vistas con un diseño adaptativo para la mayoría de dispositivos.

5.1 Prototipado de pantallas

El diseño de la totalidad de pantallas se ha realizado siguiendo las pautas y directrices marcadas por el estándar *Material Design*. Su gran contenido de iconos, estilos y componentes se ha plasmado, gracias al paquete *material*, directamente en el aplicativo.

Antes del desarrollo completo de cada una de las pantallas del aplicativo, se ha optado por el diseño de *MockUps*, indicando cada uno de los casos de uso y requisitos funcionales vistos en el Capítulo 3, indicados por los identificadores del 1 al 21.



Figura 5.1: MockUps Pantalla de Inicio y Selector de niveles clásicos

La Pantalla de Inicio será la encargada de proveer al usuario el acceso de todos los casos de uso y requisitos funcionales disponibles.

La pantalla de Selector de niveles clásicos mostrará todos los niveles disponibles pre-determinados del sistema, muchos de ellos extraídos de los libros *Griddlers*, publicados por el noticiero británico *Sunday Telegraph*. A medida los niveles han sido superados se descubrirá el nombre de la figura que representa, **Figura 5.1**.

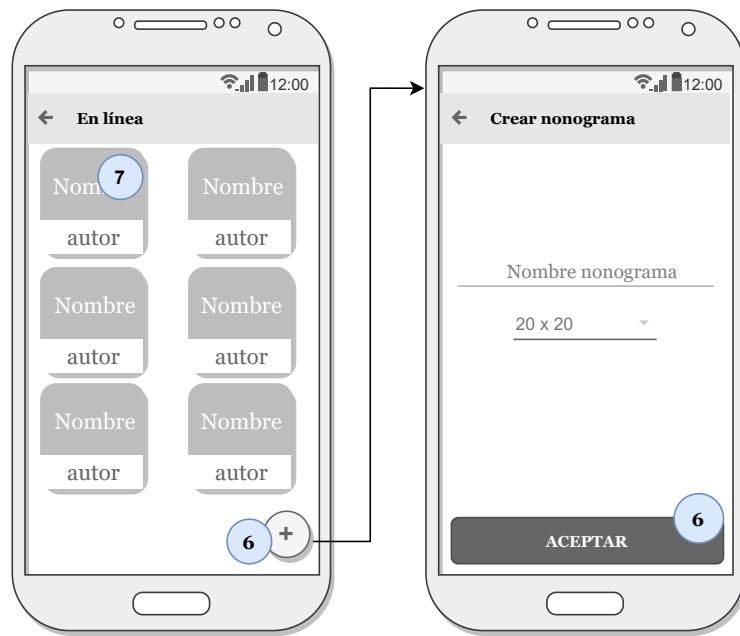


Figura 5.2: MockUps Pantalla de niveles online y publicación de nonogramas

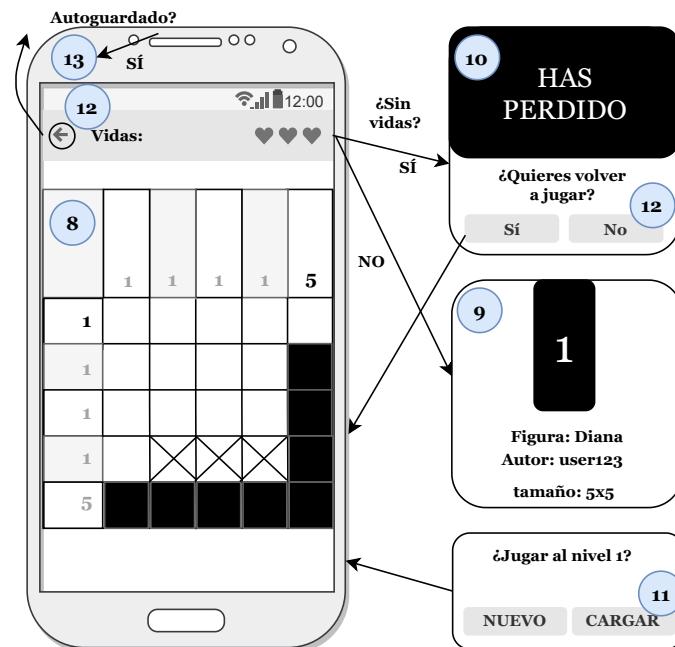


Figura 5.3: MockUp Pantalla de resolución de nivel

En la [Figura 5.3](#) se puede observar el diagrama de flujo de resolución de nivel. Como se puede apreciar, la apariencia y esencia del rompecabezas es fiel al de los medios tradicionales:

- Las celdas pintadas representan las celdas que se han seleccionado (con un clic) y resultan correctas
- Aquellas celdas con *equis* son las marcadas por el usuario (con doble clic sobre ella), asegurando que no son correctas.
- Los bloques de los extremos contienen el número de celdas correctas de esa fila o columna, los cuales se van difuminando a medida que se van resolviendo por el usuario.

El jugador puede abandonar el nivel en cualquier momento, guardando su progreso si la opción de autoguardado está activada. En el instante de resolverlo satisfactoriamente (con al menos una vida) se mostrarán las características del mismo, y en el caso contrario, la opción de volver a intentarlo.

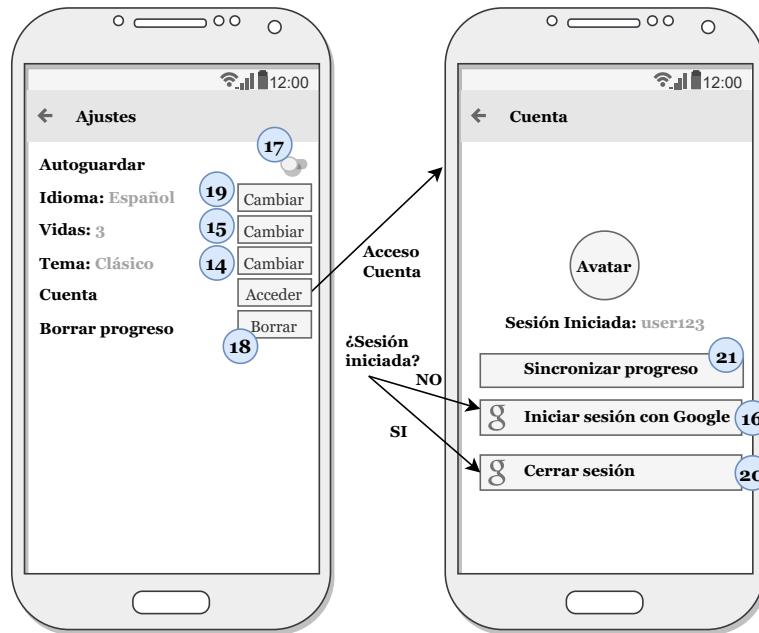


Figura 5.4: MockUp Pantalla de ajustes y cuenta

En la pantalla se pueden configurar propiedades relativas a:

- La experiencia de juego como: la posibilidad de autoguardado y el número de vidas.
- El idioma de la totalidad del sistema, por ahora para este *MVP*, entre español e inglés.
- La combinación de colores que componen el tema del aplicativo.

El usuario puede borrar el progreso de todos sus niveles jugados mediante la opción de borrar progreso, además de sincronizarlo con el *servicio en nube* desde el apartado cuenta, iniciando sesión previamente ([Figura 5.4](#))

5.2 Estructura del proyecto

Es importante que en un desarrollo de software se establezca una estructura propia de la arquitectura elegida, para mantener una modularidad y escalabilidad a nivel de sistema. Para ello, siguiendo las directrices del modelo *Clean Architecture*, se ha definido el siguiente árbol de directorios:

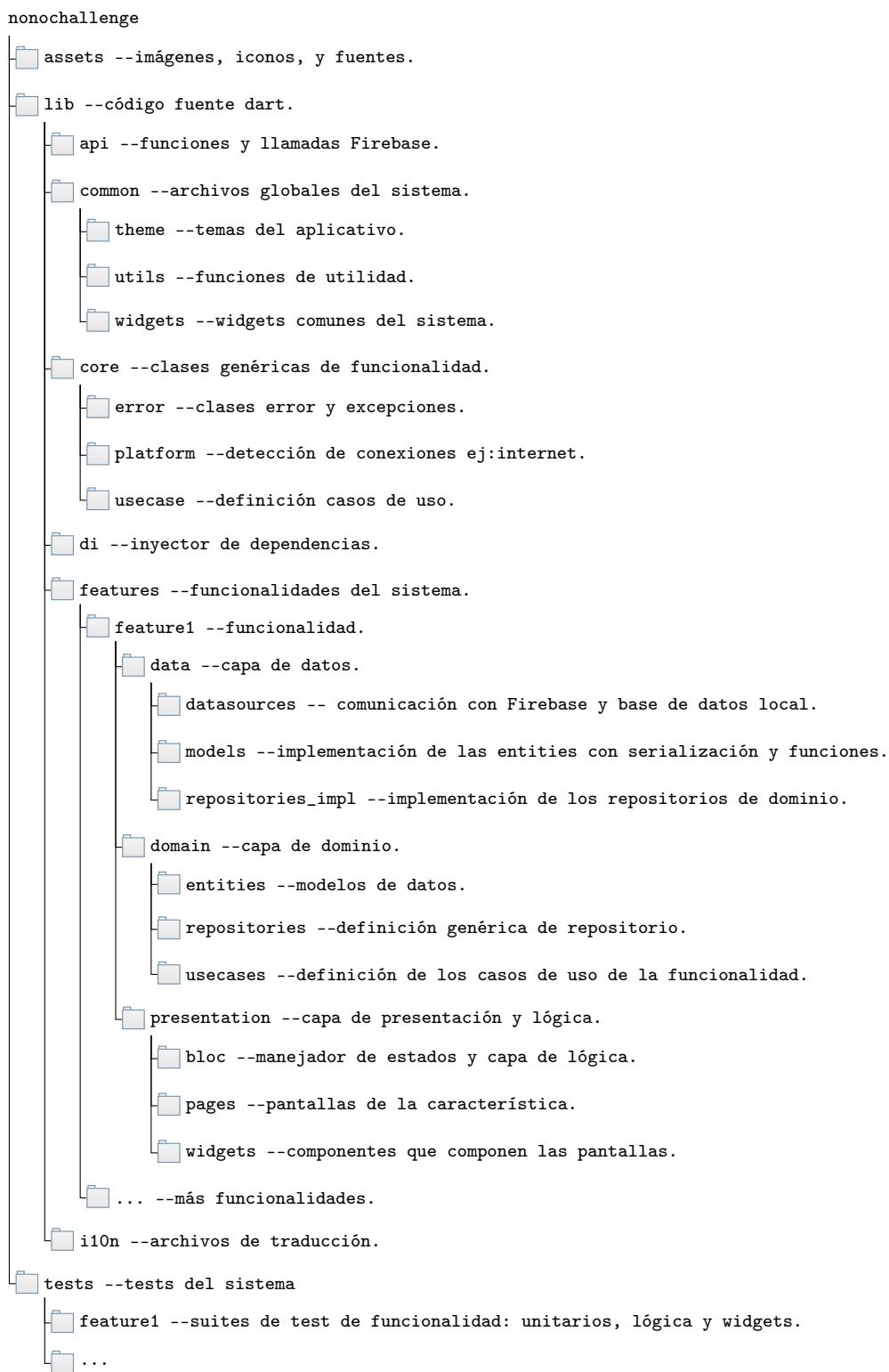


Figura 5.5: Estructura de directorios de *nonogramchallenge*

5.3 Codificación del aplicativo

El proceso de codificación se ha realizado posteriormente a la creación de una *suite de tests* de funcionalidades, siguiendo las pautas y reglas establecidas por la metodología TDD, proceso documentado en el sexto capítulo.

En el siguiente apartado se explicará el *modus operandi* de la codificación de funcionalidades, acompañado de ejemplos definidos en el [Apéndice B](#), comenzando por la capa de presentación.

5.3.1. Desarrollo de la capa de presentación

La creación de cada una de las pantallas y sus componentes se ha realizado, como no podría ser de otra forma, a partir de un *árbol de widgets*. Para el desarrollo de pantallas se ha optado por usar widgets inmutables, llamados, *stateless*, en contraposición a los de tipo *stateful*, con estado mutable. [15]

Esta práctica permitirá que la *capa de presentación* contenga la mínima lógica posible, pudiendo en un futuro reutilizarla en otras funcionalidades del sistema o incluso en otras soluciones.

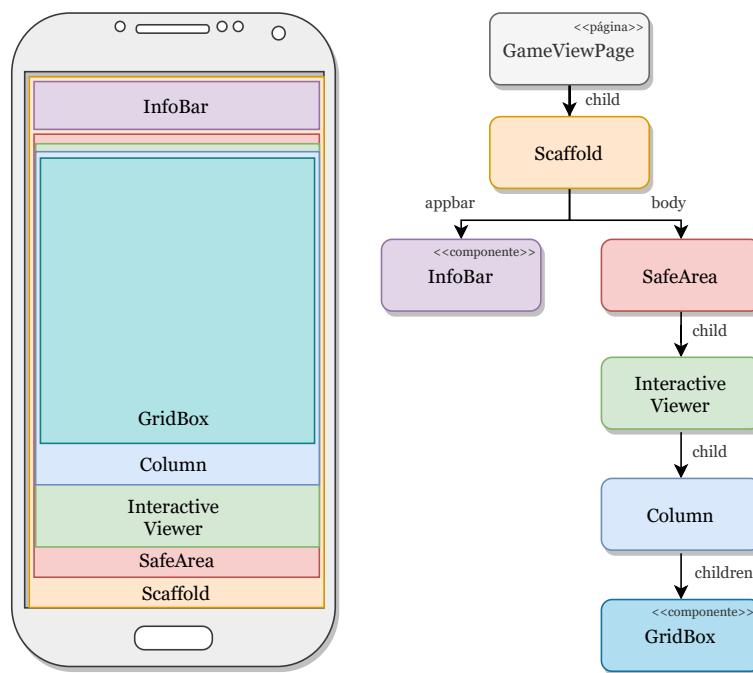


Figura 5.6: Árbol de widgets Pantalla de Resolución de nivel

En la [Figura 5.6](#) y [Sección B.1](#) se puede visualizar el *árbol de widgets* de la pantalla de resolución de nivel, los cuales se “construyen” mediante el método *build()*.

Esta vista contiene algunos widgets de interés que se han usado para mejorar la experiencia de usuario como:

- *SafeArea*: contenedor que acopla la pantalla en el caso que el dispositivo contenga elementos físicos como *notch*, *statusbar*, marcos de pantalla redondeados...
- *InteractiveViewer*: permite al usuario realizar gestos como acercar, alejar y mover el rompecabezas para casos en los que el *nonograma* es de grandes dimensiones.

5.3.2. Desarrollo de la capa de lógica

Para implementar la capa de lógica, se usará el paquete *flutter_bloc*, para ello se definirán todos los posibles estados y eventos presentes en dicha funcionalidad y una clase *BLoC* que se encargue de “manejar” el flujo de cambios de estado.

A continuación, se mostrará un diagrama de flujo de estados de la funcionalidad *lista de nonogramas en línea*.

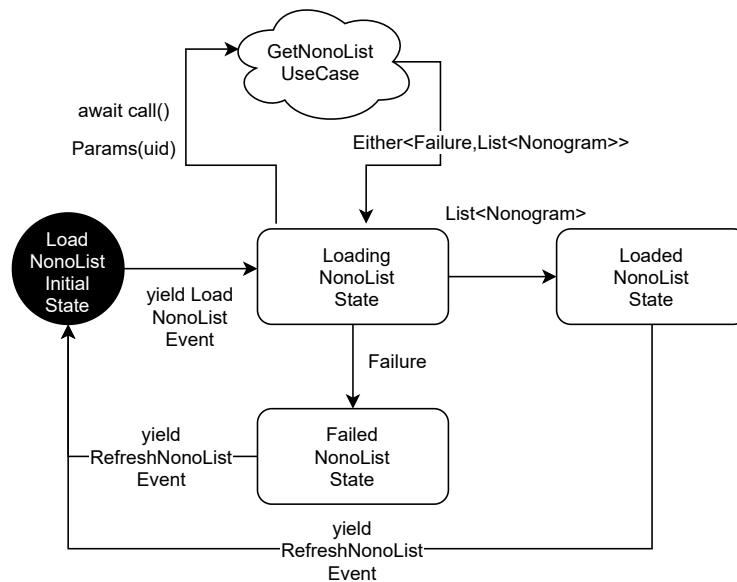


Figura 5.7: Diagrama de estado de la funcionalidad Lista de nonogramas en línea

Cuando el usuario accede a la sección de nonogramas en línea se manejará, por defecto, el estado inicial *LoadNonoListInitialState*, que a su vez, enviará el evento *LoadingNonoListEvent*.

En cuanto se envíe tal evento, se tramitará el estado *LoadingNonoListState*, mostrando un widget de carga.

Durante este estado de “carga” pedirá al caso de uso *GetNonoListUseCase* la lista de *nonogramas* disponible en *Firebase*. Tras esta llamada del caso de uso, nos puede devolver los siguientes tipos de objeto:

- La totalidad de *nonogramas* publicados por la comunidad disponible en *Firebase*.
- Un objeto de tipo *Failure*, indicando que no se ha podido tramitar la operación debido a por ejemplo un error de conexión o de base de datos.

En el caso de devolver la lista de *nonogramas* actualizado con éxito, se manejará el estado *LoadedNonoListState*, mostrando la lista de *rompecabezas* al usuario y en su defecto el estado de *FailedNonoListState*, cargando una vista de error.

El código correspondiente se puede encontrar en [Sección B.2](#).

5.3.3. Desarrollo de la capa de dominio

El desarrollo de la *capa de dominio* comienza con la definición de casos de uso a usar por el *BLoC*. Esta implementación se realiza a partir de la clase genérica *usecase* disponible en el paquete *core*.

Como se ha podido comprobar en la *capa de dominio*, este caso de uso devuelve un objeto de tipo *Either<Failure,Type>*, indicando que puede devolver el tipo de objeto deseado, *Type*, o en caso contrario, un objeto de tipo *Failure* debido a un error ocurrido en la llamada del repositorio. Adicionalmente, este caso de uso puede recibir o no una lista de parámetros a usar por el repositorio, como por ejemplo, el *uid* del usuario.

Este aspecto, puede realizarse gracias al paquete externo *dartz*, dotando a *Dart* de aspectos y beneficios propios de los lenguajes funcionales.

Una vez definido el caso de uso se definirán las clases genéricas de *repositorios* y *entities* a usar en la *capa de datos*.

Siguiendo la funcionalidad de *Lista de nonogramas en línea* se puede mostrar la implementación de su *capa de dominio* en el [Sección B.3](#).

5.3.4. Desarrollo de la capa de datos

El último paso consiste en desarrollar la *capa de datos* de la funcionalidad, comenzará con la definición de los *modelos* y la *implementación de los repositorios* a partir de las clases genéricas de *entities* y *repositories* respectivamente, definidas previamente en la *capa de dominio*.

La implementación de estos *repositories* emplearán clases de tipo *datasource* que establecerán una conexión con fuentes de datos externas propias del aplicativo como: i) la base de datos local de la solución y ii) el servicio *FireStore* propio de *Firebase*.

Esta *implementación del repositorio* se encargará de devolver un objeto *Left* o *Right* propio del paquete *dartz*, dependiendo del comportamiento de los *datasources* empleados:

En la funcionalidad citada como ejemplo dichos ejemplos contendrán los siguientes tipos de objeto, tal y como se puede visualizar en [Sección B.4](#):

- El objeto *Right* contendrá la lista de nonogramas actualizado obtenido por la llamada a *Firebase* realizada por el *datasource*
- El objeto *Left* contendrá un *Failure* indicando que no se ha podido tramitar la operación debido a por ejemplo un error de conexión o de base de datos.

5.3.5. Inyección de dependencias

Por último, con el fin de “localizar” cada una de las capas mencionadas y hacerlas accesibles en cada punto del aplicativo, se empleará el uso de un *Inyector de dependencias*.

Esta práctica es altamente recomendable, sobre todo, en soluciones de una extensión importante. Para ello, se definirá en el directorio *di* del sistema raíz *lib* la clase *injection_container.dart*, tal y como se puede visualizar en [Sección B.5](#).

La definición de esta clase se realizará mediante la librería externa *Get_It*, uno de los *localizadores de servicios* más popular. En el se crearán *factories* para “inyectar” los *BLoCs* y *singletons* para el resto de servicios (*repositories, use cases, datasources* y servicios externos como *Firebase* y *Sqflite*).

CAPÍTULO 6

Pruebas

“Los tests en el desarrollo guiado por pruebas (TDD) son los dientes de un engranaje. Una vez que se desarrolla un test completamente funcional, es seguro que este va a funcionar por y para siempre.” [2]

6.1 Metodología TDD

Tal y como se había comentado en el capítulo anterior, el periodo de codificación de cada una de las funcionalidades ha sido posterior a una fase de creación de *suite de tests automatizados*. Este *modus operandi* es propio de la metodología *Test Driven Development* (TDD), práctica que se ha usado por desarrolladores durante decadas y que ha ido ganando popularidad como una de las prácticas principales de la metodología *Extreme Programming* [11].

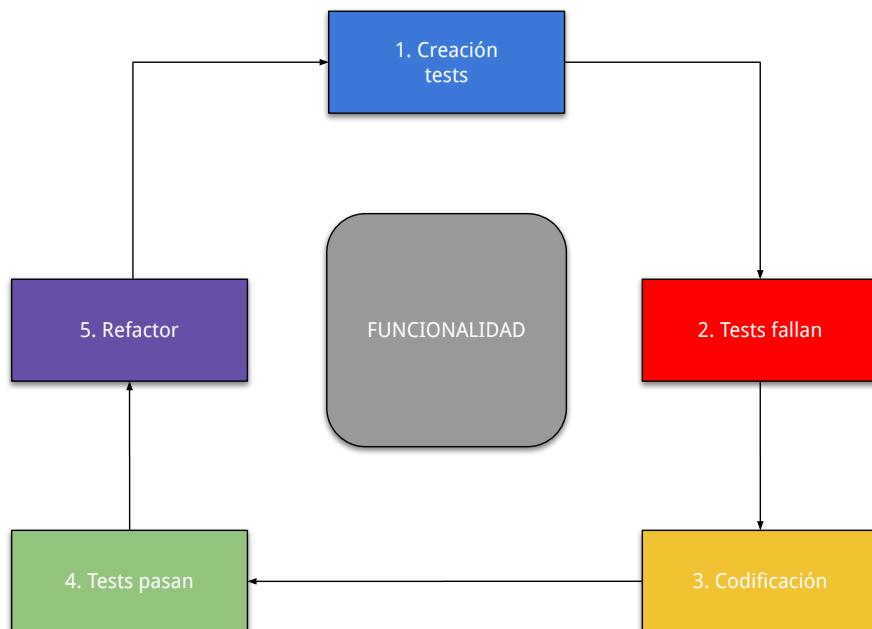


Figura 6.1: Diagrama de flujo de la metodología TDD

La elección de la arquitectura hexagonal *Clean Architecture* en el proyecto es un aspecto que facilita notablemente la implementación de la metodología en el proyecto, permitiendo abstraer completamente los test para poder reutilizarlos incluso en otros casos de uso.

Las pautas de esta metodología sugieren un diagrama de flujo similar al de la [Figura 6.1](#), este flujo de estados se ha aplicado a todas las funcionalidades (*features*) del proyecto:

1. De cara a integrar una nueva funcionalidad en el proyecto, se desarrolla una *suite de tests*, obteniendo una idea central y unas pautas más claras de la funcionalidad a desarrollar.
2. El siguiente paso consiste en ejecutar los tests. De forma evidente se obtiene una serie de resultados fallidos, ya que aún no se ha desarrollado la funcionalidad. Las pruebas deben de testar funciones lo más concretas y triviales posibles, además de ser completamente independientes unas de otras.
3. En este momento se desarrolla la funcionalidad, dividida por las capas ya comentadas en el anterior capítulo.
4. Conforme se va desarrollando la funcionalidad la ejecución de los tests van adquiriendo valores exitosos. De este modo, se asegura que las porciones de código desarrolladas puedan contener el mínimo número de *bugs*.
5. Una vez la ejecución de los tests de la funcionalidad tiene ausencia de errores, se procede a la *refactorización* del código (desde cambios de nombres de variables hasta desechar posibles elementos innecesarios). Estas sucesivas *refactorizaciones* favorecen que el producto esté mantenido de forma constante.

Ya enumerados estos pasos, se llega a la conclusión de que la idea central, que alberga esta metodología *TDD*, no se resume como el simple hecho de *testear antes de escribir código*, sino un proceso capaz de:

- Explorar y comprender mejor la problemática central de la solución antes de comenzar con su implementación.
- Detectar y anticiparse frente a posibles excepciones o problemas que puedan surgir durante el desarrollo.
- Detectar posibles actualizaciones de requisitos en base a la complejidad de una determinada prueba (*retrospectiva*).

En la práctica, resulta realmente extensa la cantidad de tipos de pruebas que se pueden emplear para testar una solución *software*. *Flutter*, como cualquier otro *framework* de desarrollo, presenta numerosas herramientas para favorecer y aligerar esta tarea. Los tipos de pruebas más recurrentes que presenta este entorno son: tests de *widgets*, *golden*, *end-to-end*, *integración* y *unitarios*.

Con el fin de obtener la mayor *cobertura de código* posible en el aplicativo, se ha optado por el uso de estos tres últimos, además de ser plenamente complementarios. En los siguientes apartados se introducirán y explicarán su uso mediante algunos ejemplos.

6.1.1. Pruebas unitarias

Los tests unitarios son una parte fundamental de cualquier desarrollo de *software*, pues son tests automatizados que aseguran el correcto funcionamiento de determinados componentes, llamados individualmente como *unidad*.

Según el volumen “*Unit Testing: Principles, Practices, and Patterns*” de *Vladimir Khorikov* (pp. 21) [13], un test se considera unitario cuando reúne estas tres características:

- Verifica una única funcionalidad.
- Se crea de forma trivial y rápida.
- Se desarrolla de forma independiente.

En este proyecto se han realizado tests de la totalidad de las clases *modelo* y *entidad*, ya que son las que va a emplear las funcionalidades del aplicativo final. Entre los más importantes se destaca, como no podía ser de otra forma, el *modelo* de la entidad *nonograma*, ya que es la que está presente en la mayoría de casos de uso de la solución.

Como se puede ver en el anexo «X», mediante el paquete *flutter_test* se testeá un objeto del modelo *Nonograma*. Este objeto se instancia en un método *setUp()* con ciertos valores arbitrarios y se testeán ciertas casuísticas en forma de *tests*, todos ellos comprendidos en un método *main()* que se encarga de ejecutarlos. Todos estos *tests* pueden “agruparse” en forma *groups*, en el caso de que exista una correlación entre cada uno de ellos.

6.1.2. Pruebas de integración

Siguiendo el mismo ejemplar anteriormente citado, un test de *integración* es aquel que no reúne ninguna de las tres características, previamente enumeradas, propias de los *tests unitarios*. Estos tipos de test cubren las porciones de código relacionadas con los controladores y lógicas de negocio.

En el caso de este aplicativo, las pruebas de integración se encargarán de cubrir y testar todos los posibles comportamientos que comprenden los manejadores de estados, las clases *BLoC*.

En estos tipos de pruebas se inicializa el *BLoC* a probar mediante el método *setUp()* y se testeá la sucesión de *estados* que emite este mismo objeto frente a un determinado *evento*. Toda esta sucesión de *estados* se definen en un *array* y se comparan con la cadena de *estados* esperada mediante el método *expect()*.

Además, los *BLoCs* pueden interactuar con servicios externos a usar por el aplicativo como *bases de datos locales y remotas, pasarelas de pago, librerías de terceros...* Y aquí entra en juego un término ampliamente conocido en el mundo del *testing*, los *Mocks*.

Los *mocks* son objetos de “prueba” que simulan o imitan ciertos comportamientos de objetos reales en un entorno controlado. Con estos tipos de objetos se pueden simular todos las casuísticas de una determinado caso de uso, desde excepciones y fallos hasta situaciones ideales. Todas estos casos en el patrón *BLoC* se pueden traducir en forma de estados.

El uso de estos *mocks* permitirá el uso de los servicios a emplear por *BLoC* sin su concreta implementación en un entorno de *producción*.

Mediante la librería externa *mockito* se permitirá incorporar este tipo de objetos en el aplicativo. Como se puede comprobar en el anexo «X», cada servicio a simular debe de extender de la clase *Mock*, proporcionado por el propio paquete *mockito*. En él se inicializan, mediante el método *setUp()*, el *BLoC* a probar junto con los servicios a simular mediante *Mocks*.

6.1.3. Pruebas end-to-end

Las *pruebas end-to-end*, son las pruebas finales y verifican que toda la experiencia de usuario es satisfactoria en un entorno con los servicios reales y en conexión y disponibles para el aplicativo. En este momento se definirá si una determinada funcionalidad estará preparada para su inclusión en el sistema, o en su defecto, requiere un ajuste en el diseño.

La representación de estos tres tipos de pruebas en el aplicativo se refleja en la [Figura 6.2](#), en esta se puede observar su alcance en este proyecto.

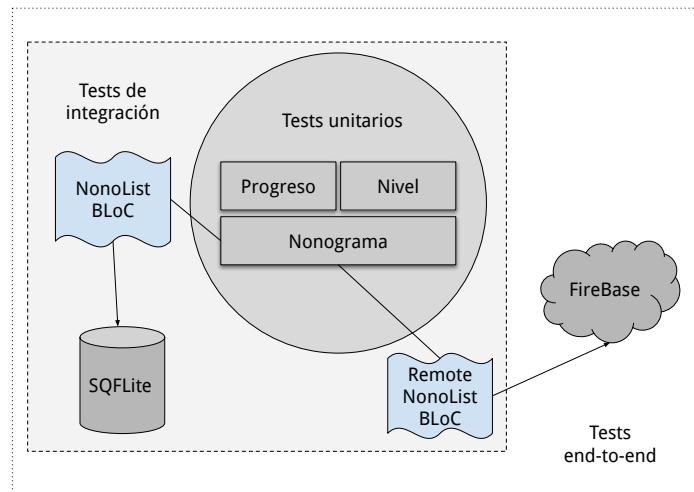


Figura 6.2: Diagrama de los tipos de pruebas en el aplicativo

CAPÍTULO 7

Implantación y mantenimiento

En un ciclo de vida de un entorno de Integración Continua (CI), tras una inserción de código en un repositorio, posteriormente revisado por un tercero, un sistema funcional coge el cambio introducido, comprueba el código y efectúa una serie de comandos que el cambio es positivo y no resulta perjudicial para la solución. [18]

7.1 Integración Continua

Ya que se ha empleado tanto metodología *PXP* como *TDD* para este proyecto, resultó idóneo el incorporar un aspecto tan positivo como es la *Integración Continua*.

Como se había comentado anteriormente en la [Subsección 4.1.4](#), la solución se aloja en un repositorio de la plataforma *GitLab*. Como la mayoría de plataformas de servicios web de control de versiones, presenta una serie de herramientas propias de prácticas *DevOps* que promueven el continuo mantenimiento y seguridad del código, esta propiedad es posible mediante el servicio *Auto DevOps*.

A través de la opción de *Quality Gate* de *GitLab*, se establecieron ciertas condiciones que debe cumplir cualquier cambio que se desee introducir a la rama *develop*. Estas comprobaciones están definidas en un archivo *YAML* en el archivo raíz del proyecto. De tal forma que cada vez que se realiza un *merge* de una determinada *feature* o directamente un *commit* hacia la rama *develop* se ejecutan la totalidad de tests desarrollados. En este momento, se desarrollan dos vertientes:

- Si las condiciones fallan no se admite la inserción de los cambios a la rama protegida a *develop*, devolviendo un mensaje de error con el test o tests fallidos.
- Si las condiciones se cumplen, se inserta el cambio o los cambios deseados en la rama *develop* con éxito.

7.2 Actualización del proyecto

Por último, se quería destacar la adaptación del proyecto de cara a las últimas versiones de *stable* en *Flutter*, más concretamente la versión *Flutter 2.0.0*.

Esta actualización se estableció el 3 de marzo de 2021 e incorporaba muchas novedades y cambios, como: soporte para entorno *web*, soporte enfocadas a plataformas: *macOS*, *Windows* y *Linux*, manejo de anuncios *Add-To-App* y la incorporación de *null-safety* en *Dart*.

Es esta última característica de la versión la que se quiso incorporar en el proyecto, esta característica permitiría evitar *aserciones* continuas en el código, definiendo clases con atributos

que pueden llegar a adquirir valores nulos, asegurando que no ocurren excepciones durante su ejecución.

Para la inclusión de esta última actualización en el proyecto se realizó mediante una herramienta de migración desarrollada por *Flutter*. El equipo de Dart sugirió una serie de pasos para la inclusión de la característica *null-safety* en el proyecto [20].

1. Esperar a que los paquetes incluidos en el proyecto adopten la última actualización con la etiqueta *null-safety*.
2. Ejecutar el cliente de migración
3. De forma estática analiza el código de tu proyecto.
4. Prueba que los cambios realizados funcionan.
5. Si el proyecto está publicado en *pub.dev*, publica el proyecto con la etiqueta *null-safety* como una versión *prerelease*.

El resultado de esta migración promovió que todos los paquetes incluidos en el proyecto estén actualizados con la última actualización *null-safety*, un código más limpio y libre de código innecesario, menos excepciones capturadas relacionadas con valores nulos.

7.3 Publicación

El último paso sería incluir el aplicativo en las principales tiendas de aplicaciones: *Google Play Store* y *App Store*, para ello se realizó las siguientes pasos para ambas plataformas.

1. Establecer a nivel de proyecto un id de aplicativo para ambas plataformas, junto a sus versiones mínimas y máximas de los sistemas operativos que soporta.
2. Creación de una cuenta con rol de desarrollador,
3. Generar una versión *release* (un *App Bundle* para Android y un *IPA* para iOS) firmadas con la cuenta de desarrollador deseada.
4. Configurar el entorno, opciones y características propias de las plataformas.

CAPÍTULO 8

Manual de uso

CAPÍTULO 9

Conclusiones y trabajo futuro

La Ingeniería de Software

9.1 Relación del trabajo relacionado con los estudios cursados

Bibliografía

- [1] R. Agarwal and D. Umphress, “Extreme programming for a single person team,” 01 2008, pp. 82–87.
- [2] K. Beck, *Test-driven Development: By Example*, ser. Addison-Wesley signature series. Addison-Wesley, 2003. [Online]. Available: https://books.google.es/books?id=gFgnde_vwMAC
- [3] A. Biørn-Hansen, T.-M. Grønli, and G. Ghinea, “A survey and taxonomy of core concepts and research challenges in cross-platform mobile development,” *ACM Comput. Surv.*, vol. 51, no. 5, Nov. 2018. [Online]. Available: <https://doi.org/10.1145/3241739>
- [4] S. Boukhary and E. Colmenares, “A clean approach to flutter development through the flutter clean architecture package,” in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 1115–1120.
- [5] A. Chakraborty, M. Baowaly, U. A Arefin, and A. N. Bahar, “The role of requirement engineering in software development life cycle,” *Journal of Emerging Trends in Computing and Information Sciences*, vol. 3, pp. 723–729, 2012.
- [6] J. Dalgety. (2017) Griddler puzzles and nonogram puzzles. [Online]. Available: <https://www.puzzlemuseum.com/griddler/gridhist.htm>
- [7] M. R. J. Estuar, M. De Leon, M. D. Santos, J. O. Ilagan, and B. A. May, “Validating ui through ux in the context of a mobile - web crowdsourcing disaster management application,” in *2014 International Conference on IT Convergence and Security (ICITCS)*, 2014, pp. 1–4.
- [8] E. D. Guzmán Chamorro, “Impacto de la implementación del software de gestión para la fase de análisis de requerimientos funcionales en la Cooperativa Financiera Atuntaqui,” Master’s thesis, 2018.
- [9] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, “Cross-platform model-driven development of mobile applications with md2,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 526–533. [Online]. Available: <https://doi.org/10.1145/2480362.2480464>
- [10] IEE, *IEEE Recommended Practice for Software Requirements Specifications*. IEEE Std 830-1998, 1998.
- [11] D. Janzen and H. Saiedian, “Test-driven development concepts, taxonomy, and future direction,” *Computer*, vol. 38, no. 9, pp. 43–50, 2005.
- [12] C. Khawas and P. Shah, “Application of firebase in android app development-a study,” *International Journal of Computer Applications*, vol. 179, no. 46, pp. 49–53, 2018.

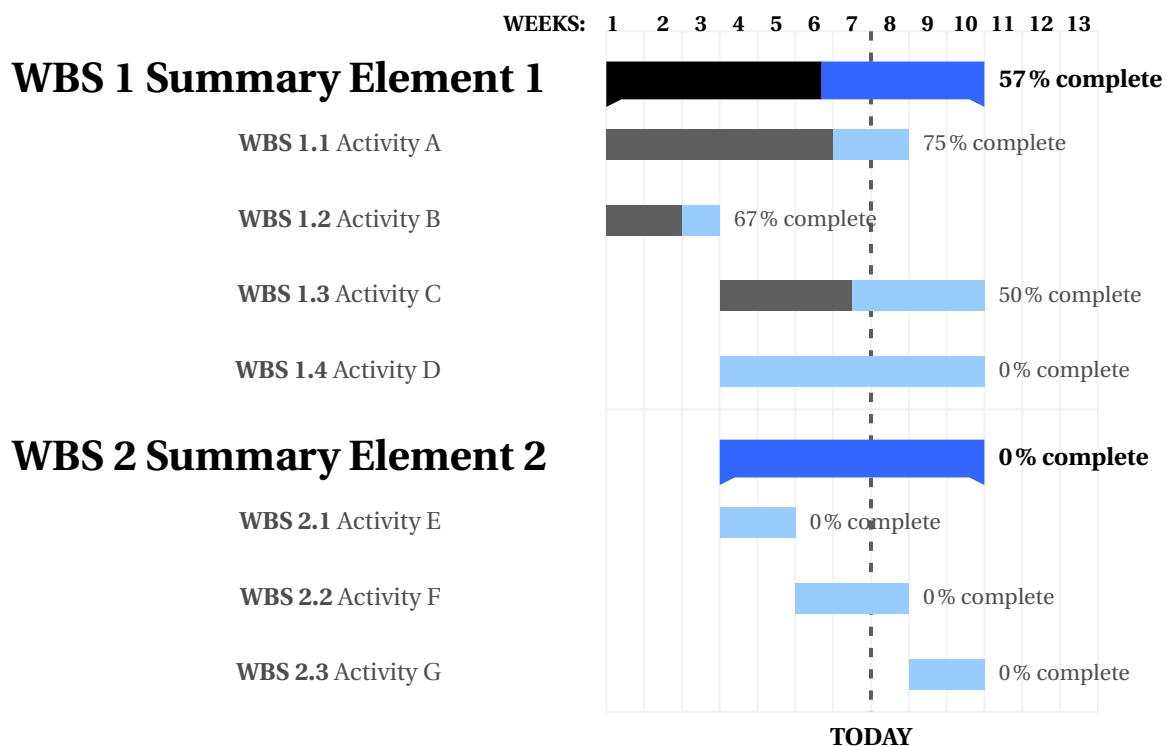
- [13] V. Khorikov, *Unit Testing: Principles, Practices and Patterns*. Manning Publications, 2020.
- [14] B. A. Kumar, “Layered architecture for mobile web based applications,” in *Asia-Pacific World Congress on Computer Science and Engineering*, 2014, pp. 1–6.
- [15] N. Kuzmin, K. Ignatiev, and D. Grafov, “Experience of developing a mobile application using flutter,” in *Information Science and Applications*, K. J. Kim and H.-Y. Kim, Eds. Singapore: Springer Singapore, 2020, pp. 571–575.
- [16] M. Latif, Y. Lakhrissi, E. H. Nfaoui, and N. Es-Sbai, “Review of mobile cross platform and research orientations,” in *2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS)*, 2017, pp. 1–4.
- [17] W. LELER, “What’s revolutionary about flutter,” *Hacker Noon*, 2019. [Online]. Available: <https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514>
- [18] M. Meyer, “Continuous integration and its tools,” *IEEE Software*, vol. 31, no. 3, pp. 14–16, 2014.
- [19] T. Sneath. (2020) Github repo tracker. [Online]. Available: <https://github.com/timsneath/github-tracker>
- [20] D. Team, “Migrating to null safety.” [Online]. Available: <https://dart.dev/null-safety/migration-guide>
- [21] F. D. Team. Adding interactivity to your flutter app. [Online]. Available: <https://flutter.dev/docs/development/ui/interactive>
- [22] K. Wiegers and J. Beatty, *Best Practises: Software requirements*. Microsoft Press, 2013.

APÉNDICE A

Diagrama de Gantt del proyecto

En el siguiente apéndice se puede observar una evolución de las distintas fases que conforman el proyecto divididas en semanas, cada una de las fases refleja su tiempo estimado y el tiempo original desde su inicio hasta su finalización.

(WORKING IN PROGRESS)



APÉNDICE B

Fragmentos de código

La intención de este anexo es la de mostrar algunos ejemplos de código extraídos del resultado de la fase de codificación, con el fin de ofrecer un mayor entendimiento de los pasos realizados para cada una de las funcionalidades desarrolladas.

B.1 Código de Pantalla de Resolución de Nivel

Este apartado muestra la codificación de la página de la funcionalidad Resolución de nivel, (*play_game_page.dart*).

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

import '../../../../../common/theme/theme.dart';
import '../widgets/grid_box_view.dart';
import '../widgets/info_bar_view.dart';
import '../bloc/play_game_bloc.dart';

class PlayGamePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    var gridBloc = BlocProvider.of<PlayGameBloc>(context);
    var colorSet = ThemeManager.of(context).colorSet;
    return Scaffold(
      appBar: InformationBar(
        mainColor: colorSet.primaryColor,
        sideColor: colorSet.secondaryColor),
      backgroundColor: colorSet.primaryColor,
      body: SafeArea(
        child: InteractiveViewer(
          maxScale: 3.6,
          child: Column(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: [
              GridBox(
                width: gridBloc.width,
                height: gridBloc.height,
              ),
            ],
          )));
  }
}
```

B.2 Código de Capa de Lógica de Niveles En Línea

El siguiente apartado contiene la codificación de las entidades, repositorios y casos de uso de la funcionalidad *Niveles en Línea*, (*nonolist_bloc.dart*, *nonolist_state.dart* y *nonolist_event.dart*).

```
part of 'nonolist_bloc.dart';

abstract class NonoListState {}

class LoadNonoListInitialState extends NonoListState {}

class LoadedNonoListState extends NonoListState {
    final List<Nonogram> nonograms;

    LoadedNonoListState({required this.nonograms});
}

class FailedNonoListState extends NonoListState {
    final String errorMessage;

    FailedNonoListState({required this.errorMessage});
}

class LoadingNonoListState extends NonoListState {}
```

```
part of 'nonolist_bloc.dart';

abstract class NonoListEvent {}

class LoadNonoListEvent extends NonoListEvent {
    final String uid;

    LoadNonoListEvent({required this.uid});
}

class RefreshNonoListEvent extends NonoListEvent {}
```

```
import 'dart:async';
import 'package:bloc/bloc.dart';
import 'package:meta/meta.dart';

import '../../../../../core/usecase/usecase.dart';
import '../../../../../domain/entities/nonogram.dart';
import '../../../../../domain/usecases/get_online_nonograms_use_case.dart';
part 'nonolist_event.dart';
part 'nonolist_state.dart';

class NonoListBloc extends Bloc<NonoListEvent, NonoListState> {
    NonoListBloc({required this.getNonoListUseCase}) : super(
        LoadNonoListEvent());
    final GetNonoListUseCase getNonoListUseCase;

    @override
    Stream<NonoListState> mapEventToState(
        NonoListEvent event,
    ) async* {
```

```

        if (event is LoadNonoListEvent) {
            final data = await getNonoListUseCase.call(event.uid);
            yield data.fold(
                (left) => FailedNonoListState(errorMessage: left),
                (right) => FailedNonoListState(nonograms: right));
        }
        if (event is RefreshNonoListEvent) {
            bloc.add(LoadNonoListEvent(event.uid));
        }
    }
}

```

B.3 Código de Capa de Dominio de Niveles En Línea

Este apartado contiene la codificación de las entidades, repositorios y casos de uso de la funcionalidad *Niveles en Línea*, (*get_nono_list_use_case.dart*, *nonogram.dart* y *nonolist_repository.dart*).

```

import 'package:dartz/dartz.dart';

import '../../../../../core/error/failures.dart';
import '../../../../../core/usecase/usecase.dart';
import '../entities/nonogram.dart';
import '../repositories/nonolist_repository.dart';

class GetNonoListUseCase extends UseCase<List<Nonogram>, GetNonoListUseCaseParams> {
    final NonoListRepository repository;

    GetNonoListUseCase({required this.repository});
    @override
    Future<Either<Failure, List<Nonogram>>> call(GetNonoListUseCaseParams params) async {
        return await repository.getNonoList();
    }
}

class GetNonoListUseCaseParams extends Equatable {
    final String uid;

    GetNonoListUseCaseParams({required this.uid});

    @override
    List<Object?> get props => [uid];
}

```

```

import 'package:dartz/dartz.dart';

import '../../../../../core/error/failures.dart';
import '../entities/nonogram.dart';

abstract class NonoListRepository {
    Future<Either<Failure, List<Nonogram>>> getNonoList();
}

```

```

class Nonogram extends Equatable {
    Nonogram({this.id,
        this.solved,
        this.name,
        this.publishDate,
        this.userId,
        this.figure,
        this.width,
        this.height,
        this.correctTiles});
    int? id;
    bool? solved;
    String? name;
    DateTime? publishDate;
    String? userId;
    String? figure;
    String? width;
    String? height;
    List<int>? correctTiles;

    @override
    List<Object?> get props => [id, solved, name, publishDate, userId,
        figure, width, height, correctTiles];
}

```

B.4 Código de Capa de Datos de Niveles En Línea

Este apartado contiene la codificación de las entidades, repositorios y casos de uso de la funcionalidad *Niveles en Línea*, (*nonolist_repo_impl.dart*, *nonogram_model.dart* y *nonolist_data_source.dart*).

```

import 'package:dartz/dartz.dart';
import '../../../../../core/error/exceptions.dart';
import '../../../../../core/error/failures.dart';
import '../../../../../domain/repositories/nonolist_repo_impl.dart';
import '../model/nonogram_model.dart';
import '../datasources/nonolist_data_source.dart';

class NonoListRepositoryImpl implements NonoListRepository {
    final NonoListDataSource nonoListDataSource;
    final AuthFirebaseDataSource authDataSource;

    NonoListRepositoryImpl({required this.nonoListDataSource,
        required this.authDataSource});
    @override
    Future<Either<Failure, List<Nonogram>>> getNonoList() async {
        var authFireBaseInstance;
        try {
            authFireBaseInstance = await authDataSource.getAuthInstace();
        } catch (e) {throw Left(AuthFailure);}
        try {
            final remoteNonoList =
                await authFireBaseInstance.getRemoteNonolist(
                    authFireBaseInstance);
            return Right(remoteNonoList);
        } on ServerException { throw Left(ServerFailure);}}}

```

```

class NonogramModel extends Nonogram {
    Nonogram({this.id,
        this.solved,
        this.name,
        this.userId,
        this.publishDate,
        this.figure,
        this.width,
        this.height,
        this.correctTiles}):super(
        id: id,
        solved: solved,
        publishDate: publishDate,
        userId: userId,
        figure: figure,
        width: width,
        height: height,
        correctTiles: correctTiles
    )}
    DocumentReference get ref =>
    FirebaseFirestore.instance.collection('nonograms').doc(id);

    static Nonogram fromDoc(DocumentSnapshot data) =>
    Nonogram.createFromData(data.data()!, data.id);

    static Future<QuerySnapshot> getNonoListSorted() {
        return FirebaseFirestore.instance
            .collection('nonograms')
            .orderBy('publish_date')
            .get(dataSource);}
```

```

import 'dart:convert';

import '../../../../../api/endpoints.dart';
import '../../../../../api/api_configuration.dart';
import '../../../../../core/error/exceptions.dart';
import '../models/nonogram_model.dart';

abstract class NonoListDataSource {
    Future<NonogramModel> getRemoteNonolist(FirebaseFirestore instance);
}

class NonoListDataSourceImpl implements NonoListDataSource {
    @override
    Future<bool> getRemoteNonolist() async {
        final nonolist =
        await instance
            .collection(NonoCollections.nonograms) //<- "nonograms"
            .get()
        if(nonolist != null) {
            return nonolist;
        } else {
            throw ServerException();
        }
    }
}
```

B.5 Código Inyector de dependencias

Este último apartado refleja el *inyector de dependencias* del proyecto, mediante la librería externa *get_it*, donde se “localizan” todas las capas (*injection_container.dart*).

```
final sl = GetIt.instance;

Future<void> init() async {
    final sharedPreferences = await SharedPreferences.getInstance();
    final auth = await FirebaseAuth.instance;
    final fireStore = await FirebaseFirestore.instance;

    sl
        // External
        ..registerLazySingleton(() => sharedPreferences)
        ..registerLazySingleton(() => auth)
        ..registerLazySingleton(() => fireStore)

        // BLoC
        ..registerFactory(() => NonoListBloc(getNonoListUseCase: sl()))

        // Use cases
        ..registerLazySingleton(() => GetNonoListUseCase(repository: sl()))

        // Repository
        ..registerLazySingleton<NonoListRepository>(() =>
            NonoListRepositoryImpl(
                nonoListDataSource: sl(),
                authDataSource: sl()))

        // Data sources
        ..registerLazySingleton<NonoListDataSource>(
            () => NonoListDataSourceImpl())
}
```