Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Implementación de un recurso docente sobre iOS para el autoaprendizaje de un lenguaje de programación

Autor: Javier López Aguirre

Tutor: Antonio J. Sierra Collado

Dep. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Implementación de un recurso docente sobre iOS para el autoaprendizaje de un lenguaje de programación

Autor:
Javier López Aguirre

Tutor:
Antonio J. Sierra Collado
Profesor titular

Dep. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado: Implementación de un recurso docente sobre iOS para el autoaprendizaje de un lenguaje de programación

Autor: Javier López Aguirre

Tutor: Antonio J. Sierra Collado

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

Resumen

Este proyecto consiste en el desarrollo de una aplicación que permita aprender, de forma básica, a programar. Está dirigido a usuarios que no dispongan de conocimientos de programación y quieran aprender las estructuras básicas presentes en la mayoría de los lenguajes de programación. El proyecto surge a partir de otro realizado el año pasado, en el que se desarrollaba para dispositivos Android, mientras que éste se desarrollará para dispositivos iOS.

Se ha optado desarrollarlo en esta plataforma debido a que entre los dispositivos Android e iOS suman una cuota de mercado del 96,4%, según los datos de la consultora IDC para el segundo trimestre de 2014. Para llegar a la mayor cantidad de usuarios posibles, es necesario desarrollar las aplicaciones para estas dos plataformas, como mínimo.

La aplicación se ha desarrollado con el objetivo de conseguir un videojuego en el que el usuario obtenga los conocimientos básicos de programación. Está divido en niveles y el usuario debe escribir programas para resolverlos.

El usuario deberá desarrollar sus conocimientos en cada nivel, ya que es necesario programar sentencias y estructuras para ir avanzando a través de ellos. Cada nivel irá aumentando la dificultad y se deberán usar expresiones más complejas. Para conseguir resolverlos, se proporcionarán ayudas como información de estructuras que se pueden utilizar, código ya escrito al inicio de un nivel o un teclado en el que se tratará de evitar la mayor parte de errores sintácticos.

El teclado se ha diseñado para que resulte sencillo escribir las sentencias, pulsando un botón se puede escribir una sentencia correcta sintácticamente. Además, se ha dividido agrupando los componentes del mismo estilo, por un lado los métodos, por otro los bucles, etc.

El juego tiene un personaje principal, cuyo nombre es Ada, el cual será programado por el usuario. Las sentencias que se escriban permitirán que Ada realice diversas acciones y cumpla los objetivos de cada misión.

Para resolver el nivel, se ejecutará línea por línea, el código escrito por el usuario. El juego dispone de un tablero en el que se podrá observar, de forma gráfica, los efectos de cada línea de código. Por ejemplo, se mostrarán animaciones al andar, golpear, etc.

Además, se permitirá seleccionar entre un lenguaje orientado a objetos, como Java, u otro estructurado, como C. El objetivo será que el usuario pueda comprobar las similitudes y diferencias entre distintos lenguajes y distintos modos de programación.

Los apuntes y niveles se han escrito teniendo en cuenta los aspectos que diferencian a estos lenguajes, de forma que sea posible resolverlo de cualquiera de las dos formas.

En esta memoria se realiza una introducción a las herramientas y tecnologías utilizadas, el funcionamiento del juego y como se ha implementado.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

1 OBJETIVOS

El objetivo de este proyecto es el desarrollo de un videojuego didáctico para dispositivos iOS. Su finalidad será que el usuario aprenda los conocimientos básicos de programación.

No se pretende estudiar las estructuras complejas que disponen los lenguajes de programación. Se estudiarán los elementos básicos presentes en la mayoría de ellos, para que el usuario posteriormente pueda indagar más en cada uno de ellos.

El videojuego presentará al usuario una interfaz que deberá ser intuitiva y fácil de usar. Incluirá indicaciones visuales o escritas para que el usuario sepa qué hacer en determinadas ocasiones. Algunas de las características principales son:

- El videojuego contendrá distintos niveles en los que se planteará un problema, y el usuario deberá escribir un código que lo resuelva. Cada nivel proporcionará pistas para ayudar a resolverlo.
- El jugador dispondrá de un teclado para escribir el código, este teclado contendrá los elementos necesarios para programar y se escribirán las líneas automáticamente con la sintaxis correcta.
- Cuando se haya terminado de escribir el código, se mostrará, de forma grafica, el resultado de cada línea de código.
- Habrá un escenario en el que los distintos personajes realizarán las acciones programadas.

Los conocimientos que debería tener una persona sobre programación tras acabar el videojuego deberían ser elementos como:

- Clases de Java.
- Objetos, métodos y atributos.
- Funciones y variables.
- Estructuras de control.

A continuación se va a decidir el entorno de desarrollo y el lenguaje para programar la aplicación.

Se utilizará el IDE Xcode para desarrollar la aplicación, el entorno de desarrollo proporcionado y recomendado por Apple para desarrollar aplicaciones nativas.

El lenguaje de programación que se utilizará será Swift, creado recientemente por Apple y que mejora en rendimiento a Objective-C.

2 INTRODUCCIÓN

2.1 Aplicaciones móviles

En los últimos años se ha producido una revolución con los teléfonos móviles desde que Apple presentó el iPhone en el año 2007. Aunque ya existían teléfonos móviles con pantalla táctil, el iPhone destacó por incluir una pantalla más grande de lo visto anteriormente.

La gran pantalla ha permitido realizar tareas cada vez más similares a las que se puede hacer con un ordenador y de forma más accesible. A partir de ahí, diversos fabricantes comenzaron a lanzar teléfonos con características parecidas, diferenciándose, principalmente, en el Sistema operativo, como Android, iOS, Windows Phone o BlackBerry OS. Por otro lado, los desarrolladores empezaron a crear aplicaciones para estos dispositivos, que introducían limitaciones, como el consumo de batería, los distintos tamaños de pantalla u otras características hardware y software.

En el año 2008, Apple lanzó la App Store, una tienda con 500 aplicaciones que fue una revolución con 10 millones de descargas en su primera semana. Poco después, Google lanzó el Android Market, una tienda similar pero sólo con 50 aplicaciones. En este tipo de tiendas, existen aplicaciones gratuitas y de pago que introducen un nuevo modelo de negocio.

Estos teléfonos, llamados *Smartphone*, se han ido expandiendo hasta tal punto en el que existen casi tantos *Smartphone* como personas. Además, las tiendas de aplicaciones tienen un alcance global.. Como consecuencia, los desarrolladores se dedican cada vez más a crear aplicaciones para estos dispositivos que permiten llegar a un número de usuarios que no existía tiempo atrás. Existen diferentes estrategias comerciales para obtener beneficios, como poner una aplicación de pago, una aplicación gratuita con publicidad, una aplicación gratuita con compras dentro o una combinación de las anteriores.

2.2 Videojuegos

Los videojuegos son uno de los tipos de aplicaciones más usados, solo por debajo de la mensajería instantánea, redes sociales o aplicaciones de correo. Las tiendas de aplicaciones están repletas de videojuegos, y muchos tipos distintos de personas juegan con ellos. En los últimos años, cada vez hemos visto más juegos con fines didácticos que permiten aprender cosas mientras sigue divirtiendo o planteando un reto para seguir avanzando.

Dentro de los videojuegos didácticos, existen algunos destinados a aprender a programar. Estas aplicaciones suelen distinguirse según se orienten más a lo didáctico o a la jugabilidad. La mayoría de estas tienen mecanismo por el que el código se escribe sin un teclado normal, aunque la dificultad del código también depende de cada aplicación.

Este proyecto toma esta idea, un juego en el que el fin sea aprender las estructuras básicas para programar una aplicación, pero a través de una interfaz agradable para ello. Se darán ayudas sobre los distintos aspectos de programación y el usuario deberá escribir el código a través del teclado que proporciona la aplicación. Ese código, servirá para resolver el nivel posteriormente. La aplicación estará destinada a personas sin conocimiento o con conocimientos mínimos de programación.

2.3 Aplicaciones existentes

A continuación se van a mencionar algunas aplicaciones existentes para aprender a programar. Existe una amplia variedad, algunos para personas con conocimientos básicos y otros para expertos.

- **CargoBot**: es un juego gratuito que nos permite programar a un robot para que vaya ordenando cajas según un patrón. Dispone de distintos niveles de dificultad por lo que es apto para distintas edades

- **Hopscotch**: es una aplicación gratuita que permite crear videojuegos y animaciones. Podemos crear objetos y programarlos mediante bloques e instrucciones.
- **Kodable**: es una aplicación gratuita, si queremos desbloquear todo es necesario realizar un pago, para aprender los conceptos básicos de programación a la vez que se está jugando.
- **Lightbot**: es una aplicación que tiene una parte gratuita y la posibilidad de aumentar el número de niveles realizando un pago. Dispone de una interfaz intuitiva y se compone de un robot la que tendremos que ir guiando para llevarlo a su destino. Permite usar bucles, condicionales, llamadas a procedimientos, etc.
- **Tynker**: es una aplicación para enseñar conceptos básicos de programación a niños y jóvenes. Se basa en un conjunto de puzles que se deben de resolver arrastrando y uniendo piezas, dispone de una interfaz similar a la de Scratch.

3 HARDWARE UTILIZADO

La implementación de la aplicación se realizará en un ordenador portátil ***Macbook Pro Retina*** de mediados de 2015. Las características son:

- Pantalla retina de 15,4 pulgadas (resolución de 2880 por 1800).
- Procesador Intel Core i7 de cuatro núcleos a 2,2 GHz (Turbo Boost de hasta 3,4 GHz) con 6 MB de caché de nivel 3 compartida.
- 16 GB de memoria DDR3L integrada a 1.600 MHz.
- 256 GB de almacenamiento flash PCIe.
- Tarjeta gráfica integrada Intel Iris Pro Graphics.
- Sistema operativo OS X 10.11.3 El Capitan.

Ilustración 1. Portátil Macbook

Para probar la aplicación en un dispositivo externo, será necesario uno que disponga del sistema operativo iOS. En concreto, uno que disponga de una versión superior a la 8.0.

El dispositivo mínimo que soporta esta versión corresponde al ***iPhone 4S***, cuyas características se disponen a continuación:

- Pantalla retina de 3,5 pulgadas (resolución de 960 por 640).
- Procesador dual-core Apple A5 1GHz.
- GPU PowerVR SGX543MP2.
- 512 MB de memoria RAM.
- 16/32/64 GB de memoria interna.
- Sistema operativo al comienzo iOS 5.0, y actualmente iOS 9.2.

Ilustración 2. iPhone

4 TECNOLOGÍAS UTILIZADAS

En el siguiente apartado se van a comentar las tecnologías necesarias para desarrollar la aplicación.

4.1 iOS

Este sistema operativo se hace público en el evento Macworld Conference & Expo del 9 de enero de 2007. El lanzamiento oficial fue el 29 de junio del 2007, quedando como nombre del mismo: iPhone OS. A partir de la presentación del iPhone 4 empezó a llamarse iOS.

Es el sistema operativo usado por los dispositivos iPhone, iPad e iPod Touch. Las aplicaciones nativas se construyen usando el framework de iOS y los lenguajes Objective-C o Swift, y se ejecutan directamente en iOS.

4.1.1 Arquitectura de iOS

Esta distribuido en cuatro capas diferenciadas por su funcionalidad. Las capas inferiores tienen servicios y tecnologías fundamentales mientras que las superiores se apoyan en las inferiores y ofrecen servicios y tecnologías más avanzadas.

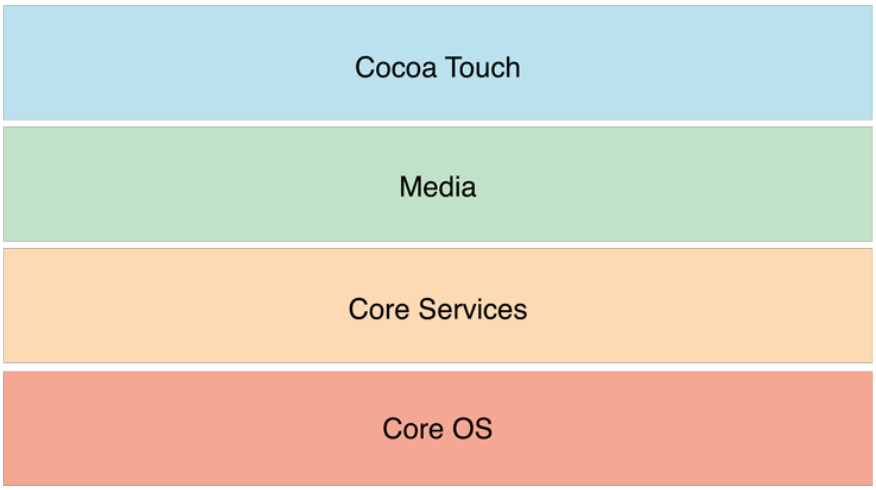


Ilustración 3. Arquitectura iOS

4.1.1.1 Core OS Layer

Esta capa incluye las características de bajo nivel sobre las que se basan la mayoría de las capas superiores. Es necesaria para interactuar con dispositivos hardware externos y para tratar la seguridad.

Incluye los siguientes frameworks:

- **Accelerate Framework**: contiene interfaces para realizar procesamiento digital de la señal (DSP), algebra lineal, y procesamiento de imagen. La ventaja de usar este framework es que está optimizado para el hardware presente en los dispositivos iOS.
- **Core Bluetooth Framework**: permite interactuar con dispositivos Bluetooth.
- **External Accessory Framework**; proporciona soporte para comunicaciones con accesorios hardware conectados al dispositivo iOS. Los accesorios pueden estar conectados físicamente o inalámbricamente mediante Bluetooth.
- **Generic Security Services Framework**: proporciona un conjunto estándar de servicios de seguridad

para las aplicaciones. Las interfaces básicas están especificadas en la RFC 2743 y RFC 4401. Además, incluye la gestión de credenciales requerida por muchas aplicaciones.

- **Local Authentication Framework**: permite usar el Touch ID para autenticar a un usuario. Permite a las aplicaciones restringir el acceso a todo el contenido de la aplicación, a determinadas opciones o información.
- **Network Extension Framework**: proporciona soporte para configurar y controlar túneles VPN (Virtual Private Network).
- **Security Framework**: permite garantizar la seguridad de los datos que maneja la aplicación. Proporciona interfaces para gestionar credenciales, claves públicas y privadas, encriptación, etc.
- **System**: maneja el entorno del kernel, los drivers y las interfaces de bajo nivel del sistema operativo UNIX. Las aplicaciones accedes a estas funciones a través de la librería *LibSystem*. Las interfaces soportan:
  - Concurrencia
  - Red (BSD sockets)
  - Acceso a sistema de ficheros
  - Entrada y salida estándar
  - Bonjour (mecanismo de descubrimiento de servicios en la LAN) y servicios DNS
  - Información de localización
  - Reserva de memoria
  - Cálculos matemáticos
- **Soporte de 64 bits**: iOS fue diseñado originalmente para soportar ficheros binarios en dispositivos que utilizaran una arquitectura de 32 bits. A partir de iOS 7, se añadió soporte para compilar, enlazar y depurar binarios en arquitecturas de 64 bits. Todas las librerías y frameworks pueden usarse para aplicaciones de 32 y 64 bits.

4.1.1.2 Core Services Layer

Contiene servicios fundamentales del sistema

Algunas de las prestaciones de alto nivel son:

- **Servicios P2P**
- **Almacenamiento iCloud**
- **Objetos Block**: es una función anónima con sus datos. Son muy utilizados como callbacks
- **Protección de datos**
- **Soporte de compartición de archivos**
- **Grand Central Dispatch (GCD)**: mecanismo alternativo para la ejecución de tareas asíncronas en lugar de hilos
- **Gestión de compras dentro de la aplicación**
- **SQLite**
- **Soporte XML**

Frameworks que ofrece:

- **Accounts Framework**: proporciona un modelo para la gestión de login de usuarios.

- **Address Book Framework**: proporciona acceso a la base de datos de contactos.
- **Ad Support Framework**: gestión de la publicidad.
- **CFNetwork Framework**: son un conjunto de interfaces para trabajar con protocolos de red, como comunicaciones con FTP y HTTP.
- **CloudKit Framework**: proporciona una vía para pasar información entre la aplicación y iCloud.
- **Core Data Framework**: tecnología para gestionar el modelo de datas en la arquitectura MVC.
- **Core Foundation Framework**: es un conjunto de interfaces que permiten la gestión de datos y servicios básicos.
- **Core Location Framework**: ofrece información de localización y orientación.
- **Core Media Framework**: proporciona tipos multimedia de bajo nivel.
- **Core Motion Framework**: proporciona un conjunto de interfaces para el acceso a datos de movimiento disponibles.
- **Core Telephony Framework**: proporciona interfaces para interactuar con la información de dispositivos que dispongan de teléfono.
- **Event Kit Framework**: proporciona una interfaz para el acceso a eventos de calendario.
- **Foundation Framework**: proporciona un encapsulado de la mayoría de las prestaciones del Core Foundation Framework.
- **HealthKit Framework**: permite la gestión de la información del usuario relacionada con la salud.
- **JavaScript Core Framework**: proporciona un encapsulado de la mayoría de objetos JavaScript estándar.
- **Mobile Core Services Framework**: define los tipos de bajo nivel usados en UTIs (uniform type identifiers).
- **Multipeer Conectivity Framework**: soporte del descubrimiento de dispositivos cercanos y la comunicación directa con ellos sin Internet.
- **Newsstand Kit Framework**: gestión acceso para que el usuario lea revistas y periódicos.
- **Pass Kit Framework**: proporciona un lugar en el que almacenar cupones, tickets de descuento, tarjetas de embarque, etc.
- **Quick Look Framework**: proporciona una interfaz directa para la previsualización de contenidos que la aplicación no soporta directamente.
- **Safari Framework**: proporciona soporte para añadir URLs a la lista de lectura del navegador Safari.
- **Social Framework**: proporciona una interfaz simple para acceder a las cuentas del usuario en redes sociales.
- **Store Kit Framework**: proporciona soporte para la compra de contenidos y servicios dentro de la aplicación.
- **System Configuration Framework**: proporciona una interfaz para la gestión de la configuración de red de un dispositivo.
- **WebKit Framework**: permite mostrar contenido HTML en la aplicación.

4.1.1.3 Media Layer

Contiene las tecnologías de gráficos, audio y video que usan las aplicaciones para implementar multimedia:

- **Gráficos**: iOS proporciona numerosas tecnologías para ayudar a incluir gráficos en la pantalla. Se

puede usar tanto vistas estándar de alta calidad como personalizadas usando algunas de las tecnologías proporcionadas para conseguir una calidad alta.

- **Audio**: las tecnologías de audio permiten obtener una calidad alta de reproducción y grabación de audio, gestionar contenidos MIDI y trabajar con dispositivos de sonido. Soporta formatos de audio estándar como: AAC, Linear PCM o DVI/Intel IMA ADPCM.
- **Video**: proporciona soporte para gestionar contenido de video estático o reproducir contenido en streaming desde Internet. También permite grabar video e incorporarlo a la aplicación. Soporta formatos de video estándar como: H.264 o MPEG-4.
- **AirPlay**: permite enviar contenido de audio y video en streaming hacia el Apple TV y enviar audio en streaming a altavoces compatibles con AirPlay.

Frameworks que ofrece:

- **Assets Library Framework**: proporciona acceso a las fotos y vídeos gestionados por la aplicación Photos del dispositivo.
- **AV Foundation Framework**: proporciona clases para reproducir, grabar y gestionar contenidos de audio y vídeo.
- **Core Audio Framework**: son un conjunto de frameworks que proporciona soporte nativo para el manejo de audio.
- **Core Graphics Framework**: contiene interfaces para la API de dibujo Quartz 2D, un motor de dibujo vectorial usado en OS X.
- **Core Image Framework**: proporciona un potente conjunto de filtros para manipular vídeos e imágenes estáticas.
- **Core Text Framework**: ofrece una simple interfaz para la presentación de textos y el manejo de fuentes.
- **Core Video Framework**: proporciona la gestión de buffer para el Core Media Framework.
- **Game Controller Framework**: permite el descubrimiento y configuración de hardware de control de juegos.
- **GLKit Framework**: contiene un conjunto de clases que simplifican el esfuerzo requerido para crear una aplicación OpenGL ES.
- **Image I/O Framework**: proporciona interfaces para importar y exportar datos y metadatos de imágenes.
- **Media Accessibility Framework**: gestiona la presentación de subtítulos en el contenido multimedia.
- **Media Player Framework**: proporciona soporte de alto nivel para la reproducción de contenido de audio y vídeo.
- **Metal Framework**: proporciona un alto rendimiento en el renderizado de gráficos y tareas computacionales. Está diseñado para aprovechar las arquitecturas modernas, que permite la paralelización de los comandos de GPU.
- **OpenAL Framework**: es una interfaz multiplataforma estándar para la entrega de audio posicional.
- **OpenGL ES**: proporciona herramientas para el dibujo de contenidos 2D y 3D.
- **Photos Framework**: proporciona nuevas API para trabajar con recursos de audio y vídeo.
- **Quartz Core Framework**: contiene interfaces para el uso de Core Animation. Una tecnología que permite crear animaciones rápidas y eficientes.
- **SceneKit Framework**: permite desarrollar juegos simples y aplicaciones con interfaces que utilicen gráficos 3D.

- **Sprite Kit Framework**: proporciona un sistema de animación acelerado por hardware para juegos 2D y 2.5D.

4.1.1.4 Cocoa Touch Layer

Contiene los frameworks fundamentales para el desarrollo de las aplicaciones.

Algunas de las prestaciones de alto nivel son:

- **AirDrop**: permite compartir fotos, documentos, URLs, y otro tipo de datos entre dispositivos cercanos.
- **TextKit**: es un conjunto de clases de alto nivel para el manejo de textos y tipografía.
- **UIKit Dynamics**: especificación del comportamiento dinámico de objetos uiview. Mejora el comportamiento de la experiencia de usuario incorporando comportamientos y características del mundo real.
- **Multitasking**: gestiona el estado de las aplicaciones que pasan a background para el ahorro de batería.
- **Auto Layout**: ayuda a la construcción de interfaces dinámicas con muy poco código.
- **Storyboard**: mecanismo recomendado para el diseño de interfaces de usuario. Permite diseñar todas las vistas y controladores en un mismo lugar.
- **UI State Preservation**: gestiona la apariencia de las aplicaciones en ejecución cuando no lo están, guarda la información del estado para restaurarlo cuando aparezca la aplicación de nuevo.
- **Apple Push Notification**: proporciona un mecanismo de alerta a usuarios de alguna información nueva, incluso cuando la aplicación no se está ejecutando.
- **Local Notifications**: complementa las notificaciones externas proporcionando un mecanismo de gestión de notificaciones de forma local.
- **Gesture Recognizers**: permite la detección de gestos.
- **Standard View Controllers**: define controladores de vistas para interfaces de usuario estándar. Se pretende usar estos en lugar de crear uno propio.

Frameworks que ofrece:

- **Address Book UI Framework**: proporciona interfaces estándar para la gestión de contactos.
- **EventKit UI Framework**: proporciona controladores de vistas para ver y editar eventos de calendario.
- **GameKit Framework**: implementa soporte para Game Center, que permite compartir la información relacionada con los juegos de los usuarios a través de la red.
- **iAd Framework**: permite la gestión de anuncios en la aplicación.
- **MapKit Framework**: proporciona un mapa que se puede añadir a la aplicación.
- **Message UI Framework**: proporción soporte para redactar emails o SMS en la aplicación.
- **Notification Center Framework**: proporciona soporte para crear widgets que muestra información en el centro de notificaciones.
- **PushKit Framework**: proporciona soporte para aplicaciones VoIP.
- **UIKit Framework**: proporciona la infraestructura para implementar aplicaciones graficas basadas en eventos.

4.2 Swift

El lenguaje elegido para el desarrollo de la aplicación es Swift. Es un lenguaje creado por Apple gratuito, de código abierto y disponible bajo la licencia libre de Apache 2.0. Combina lo mejor de C y Objective-C, sin las limitaciones de la compatibilidad con C. Adopta patrones de programación seguros y añade características modernas para hacer la programación más sencilla y flexible.

Se ha elegido este en lugar de Objective-C por diversas razones:

- Al no conocer ninguno de los dos lenguajes, es más fácil aprender Swift. Es un lenguaje mucho más intuitivo.
- Tiene mejor rendimiento que Objective-C. Según Apple, es hasta 2,6 veces más veloz.
- El IDE Xcode permite trabajar con los dos lenguajes al mismo tiempo, por lo que si se necesita una librería que sólo exista para Objective-C, es posible utilizarla.
- En un futuro, será el lenguaje que se utilice para todas las aplicaciones de Apple, e incluso para otras plataformas.

4.3 Estructura de una aplicación

Las aplicaciones desarrolladas en iOS siguen el patrón denominado Modelo Vista Controlador (MVC). Este patrón se divide en tres campos:

- **Modelo**: Es la representación especifica de la información, con la cual va a operar la aplicación.
- **Controlador**: Responde a los eventos, normalmente acciones del usuario e invoca peticiones al Modelo y a la Vista.
- **Vista**: Objeto subordinado del Controlador que es presentado al usuario de forma gráfica.

Se comunican de la siguiente manera: el Controlador puede hablar directamente con su Modelo y con su Vista, sin embargo, el Modelo y la Vista no pueden hablar directamente el uno con el otro.

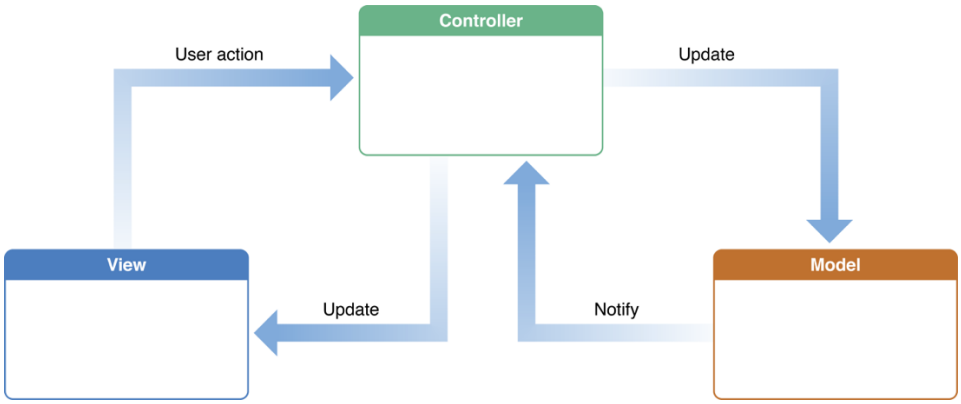


Ilustración 4. Modelo-Vista-Controlador

Durante el arranque de una aplicación, la función UIApplicationMain establece varios objetos clave e inicia la aplicación. En el corazón de toda aplicación iOS se encuentra el objeto UIApplication, cuyo trabajo consiste en facilitar las interacciones entre el sistema y otros objetos en la aplicación.

La siguiente imagen muestra los objetos que aparecen normalmente en una aplicación:

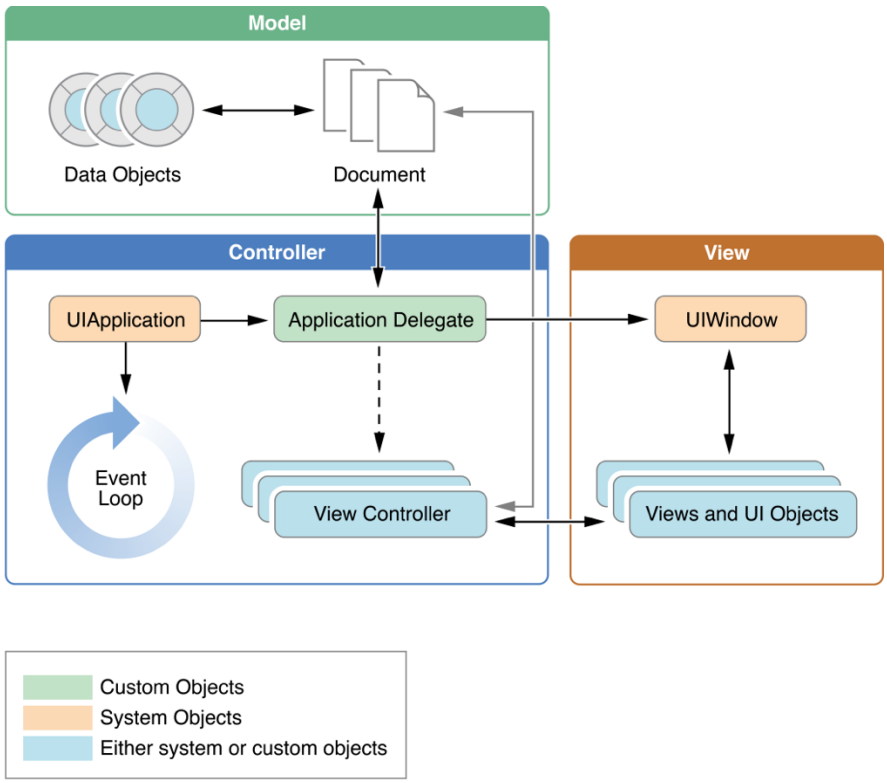


Ilustración 5. Estructura de aplicación iOS

A continuación detallamos el papel de los objetos en una aplicación:

- **UIApplication object**: gestiona el ciclo de eventos y otros comportamientos de alto nivel de la aplicación. También informa sobre transiciones clave de la aplicación y eventos especiales a su delegado, que es un objeto personalizado que se tiene que definir.
- **App delegate object**: contiene una serie de métodos que permite que el objeto *UIApplication* dialogue con él. Trabajan en conjunto para gestionar la inicialización de la aplicación, las transiciones y eventos de alto nivel. Cuando suceda algo importante el objeto *UIApplication* llama a los métodos del *App delegate* para responder adecuadamente.
- **Documents and data model objects**: almacenan el contenido de la aplicación y son específicos para cada aplicación. Las aplicaciones también puede usar documents objects para gestionar todos o algunos de los objetos *data model*.
- **View controller objects**: gestiona la presentación del contenido de la aplicación en la pantalla. Un view controller gestiona una única vista y su colección de subvistas. Cuando se muestra, hace visible sus vistas instalándolas en la ventana de la aplicación. La clase *UIViewController* es la base de todos los objetos *view controller*.
- **UIWindow object**: coordina la presentación de una o más vistas en la pantalla. Para cambiar el contenido de la aplicación, se usa un *view controller* para cambiar las vistas mostradas en la correspondiente ventana.
- **View, control and layer objects**: las vistas y los controles proporcionan la representación visual del contenido de la aplicación. Una vista es un objeto que dibuja contenido en un área rectangular y responde a eventos en esa área. Los controles son un tipo de vista especializado en implementar interfaces de objetos familiares como botones o campos de texto. Las capas son objetos de datos que representan contenido visual, se pueden añadir capas personalizadas para implementar animaciones complejas y otros efectos visuales.

El bucle principal de la aplicación es el responsable del procesamiento de todos los eventos relacionados con el usuario. Los eventos se tratan ordenados según la recepción y se entregan en un puerto especial establecido por el UIKit.

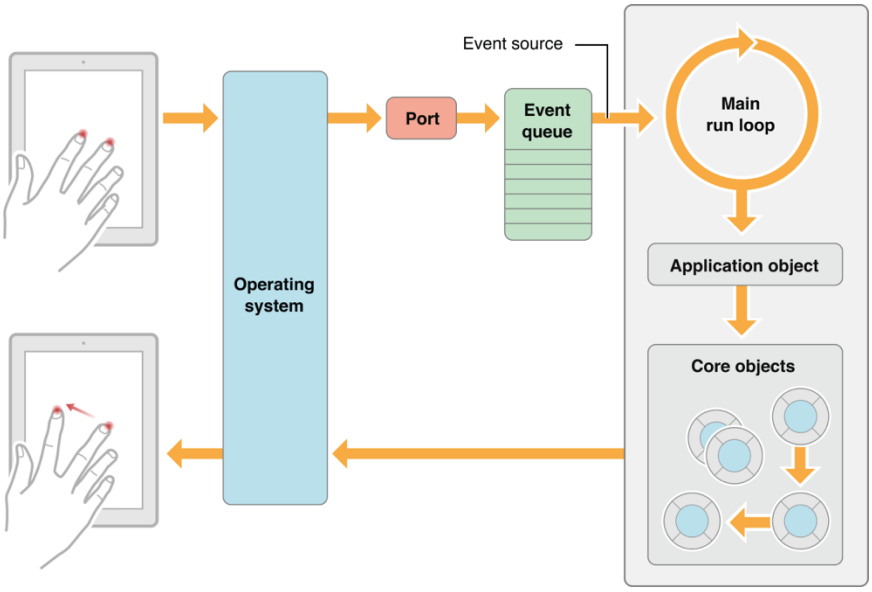


Ilustración 6. Procesamiento de eventos en una aplicación iOS

En todo momento, la aplicación está en un estado que puede ir cambiando respondiendo a las acciones que se produzcan. Por ejemplo, cuando se pulsa el botón Home o se recibe una llamada.

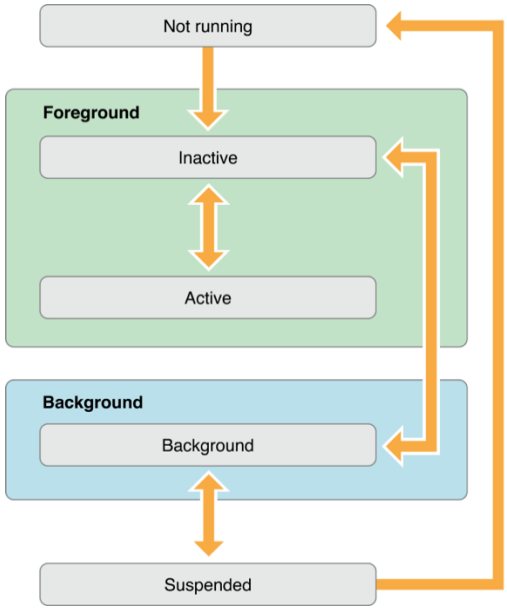


Ilustración 7. Cambios de estado en una aplicación iOS

A continuación se describen los estados en los que se puede encontrar una aplicación:

- **Not running**: la aplicación no se ha iniciado o estaba ejecutándose y el sistema finalizó la ejecución.
- **Inactive**: la aplicación se está ejecutando en primer plano pero no está recibiendo eventos.
- **Active**: la aplicación se está ejecutando en primer plano y está recibiendo eventos.

- **Background**: la aplicación se encuentra en segundo plano y ejecutando código. La mayoría de las aplicaciones entran en este estado brevemente antes del estado suspendido.
- **Suspended**: la aplicación se encuentra en segundo plano pero sin ejecutar código. El sistema mueve a la aplicación a este estado y no lo notifica antes. Si el sistema necesita memoria, el sistema destruye las aplicaciones suspendidas.

4.4 Entorno de desarrollo

Para el desarrollo de la aplicación se usará el IDE ***Xcode*** de Apple. Es una aplicación gratuita que funciona bajo el sistema operativo OS X. Como características principales dispone de:

- Editor de código. Proporciona ayudas como la detección de errores mientras vamos escribiendo el código, además, nos indica como poder solucionarlo.
- Permite diseñar interfaces gráficas sin escribir código.
- Editor de versiones con total soporte de Subversion y Git.
- Un simulador de iOS en el que podemos instalar y probar las aplicaciones.
- Extensa documentación.

La versión utilizada es la 7.2.1, incluye soporte para iOS 9.2, OS X 10.11.2, tvOS 9.1, y watchOS 2.1.

Ilustración 8. Xcode

A continuación se mostrará la creación de un proyecto básico:

Al abrir el programa, creamos un proyecto nuevo. Primero es necesario elegir una plantilla, nos aparecen varias opciones divididas según el sistema operativo para el que queramos desarrollar. En nuestro caso, elegiremos *Single View Application* para iOS.

Una vez elegido, se introducirán los datos del proyecto como el nombre del producto, el de la organización, el identificador de ésta y el identificador único de la aplicación. Además, se elegirá el lenguaje con el que se desarrollará el proyecto, Swift u Objective-C, y el dispositivo para el que esta destinado, iPhone, iPad o ambos (universal).

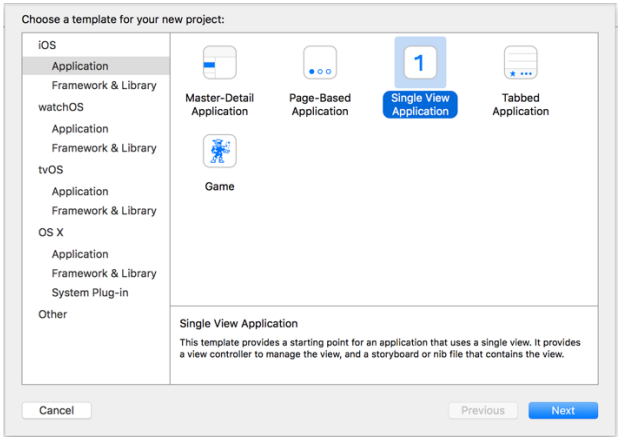


Ilustración 9. Creación de un proyecto (1)

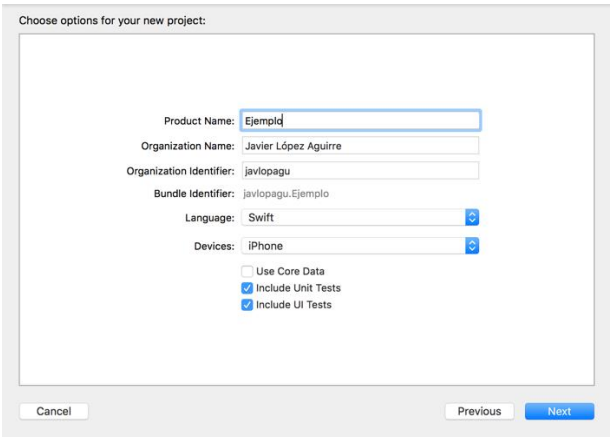


Ilustración 10. Creación de un proyecto (2).

Cuando se hayan introducido estos datos, se creará el proyecto con una serie de archivos que se generarán automáticamente. Entre estos, se encuentran:

- **AppDelegate.swift**: esta clase define unos métodos que gestionan el ciclo de vida de la actividad, como applicationDidEnterBackground o applicationDidBecomeActive.
- **ViewController.swift**: clase principal que se encarga de controlar la interfaz principal de la aplicacion
- **Main.storyboard**: interfaz principal que se ejecuta una vez se ha cargado la aplicación.
- **LaunchScreen.storyboard**: interfaz que se ejecuta al cargar la aplicación.
- **info.plist**: archivo que usa el sistema operativo para almacenar datos necesarios para configurar la aplicación.

Sobre la información del proyecto, se pueden configurar ciertos parámetros que hemos visto antes más otros como:

- Versión del proyecto.
- Versión del SO objetivo: se especifica la versión mas baja del sistema operativo en la que puede desplegarse la aplicación.
- Interfaz principal: El archivo que se ejecuta una vez se ha cargado la aplicación, por defecto Main.storyboard.
- Orientaciones del dispositivo: se puede habilitar o deshabilitar la opción de que una aplicación se oriente hacia un lado u otro.

- Icono de la aplicación.
- Fichero que se mostrará mientras se carga la aplicación, por defecto LaunchScreen.storyboard.

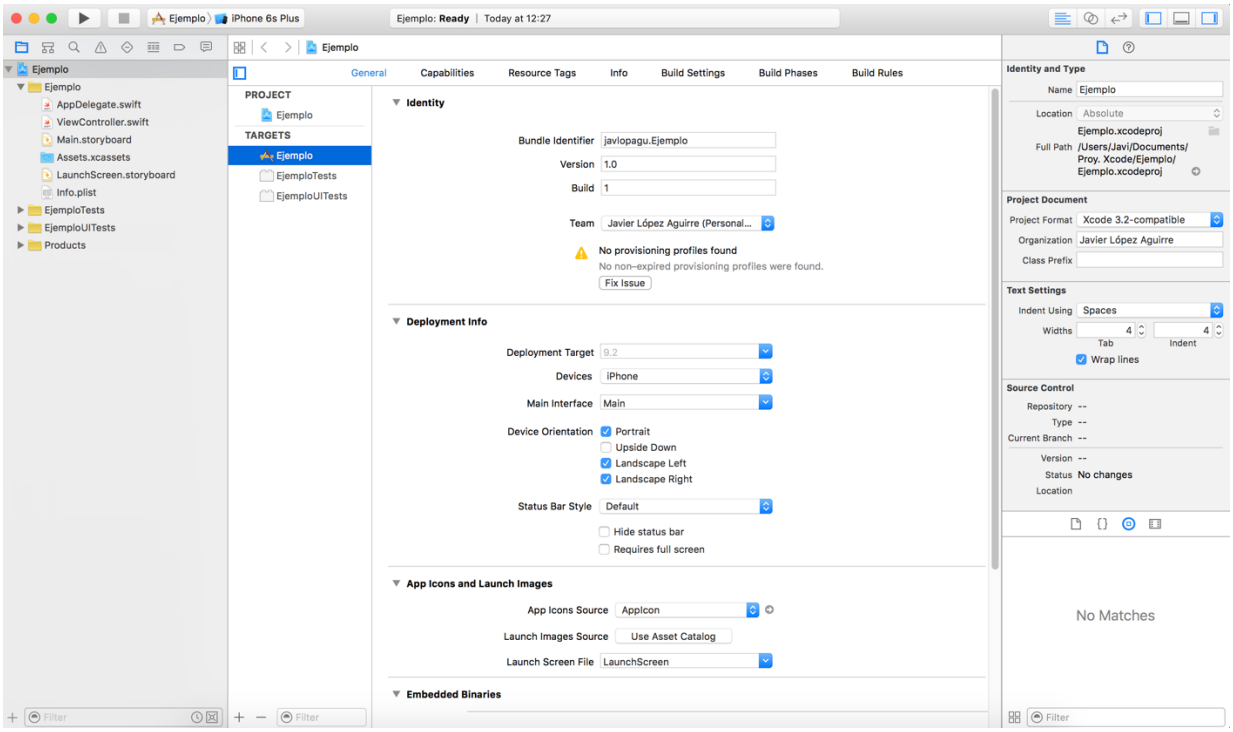


Ilustración 11. Informacion general proyecto

Para editar la interfaz gráfica disponemos del Interface Builder. Nos aparecen las vistas de la aplicación y podemos ir colocando elementos, como botones, textos, imágenes, etc., y ajustar las restricciones para que se dispongan en el sitio que deseamos.

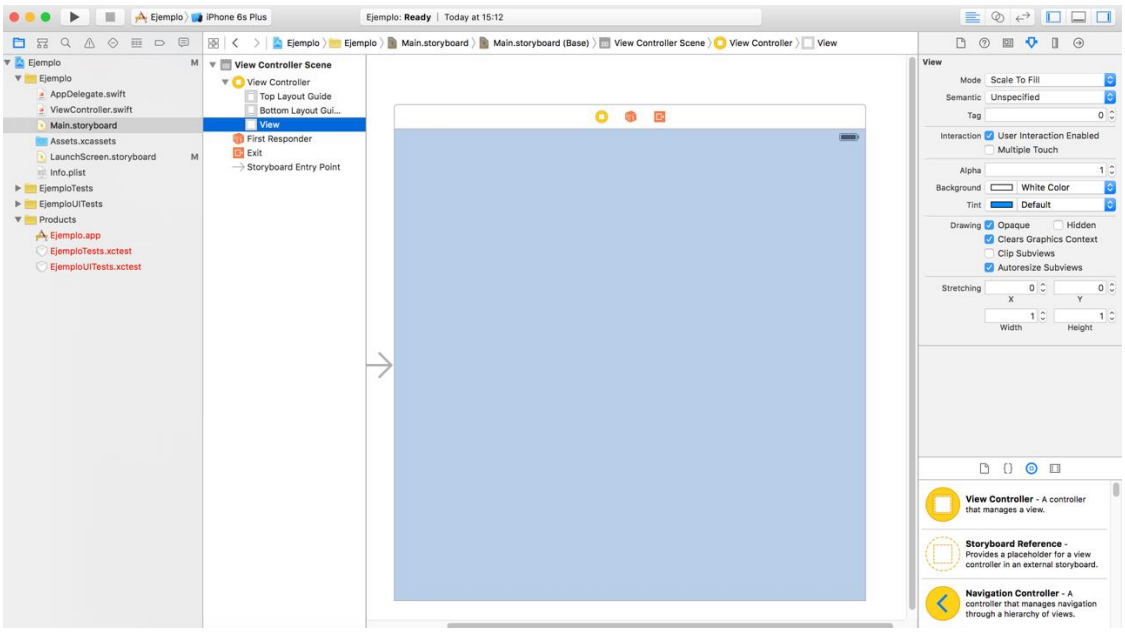


Ilustración 12. Interfaz principal Xcode

Por ejemplo, añadimos un botón y ajustamos las restricciones. Existen varias opciones para ajustarlo, como añadir restricciones de tamaño un elemento, distancia hacia el siguiente elemento, ya sea hacia izquierda, derecha, arriba o abajo.

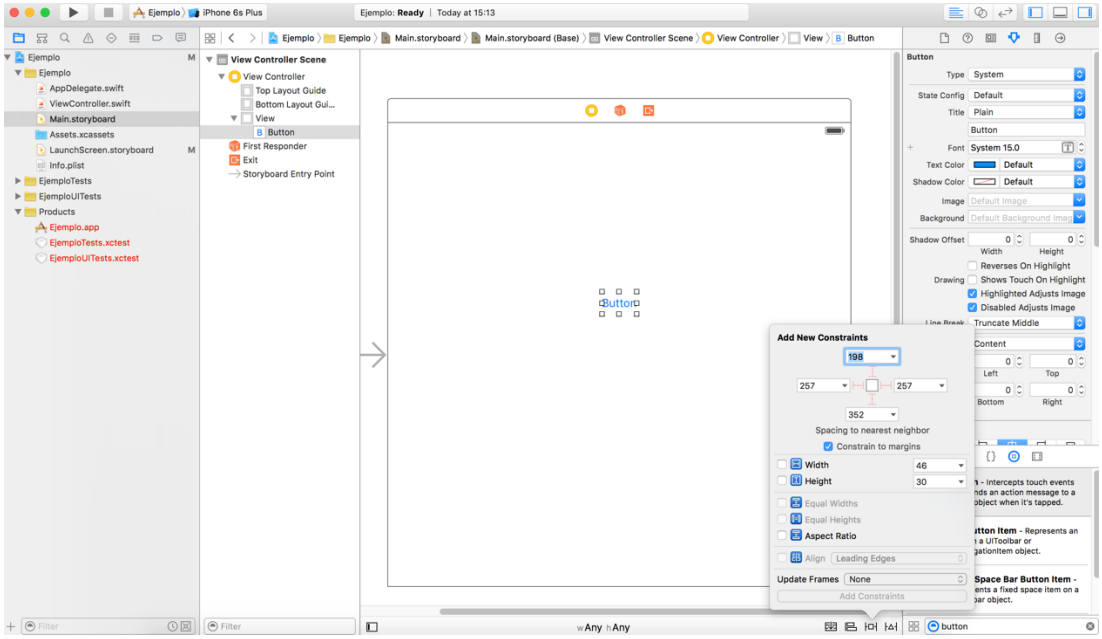


Ilustración 13. Añadir restricciones a botón

Tambien podemos ajustar la posición respecto al centro tanto verticalmente como horizontalmente.

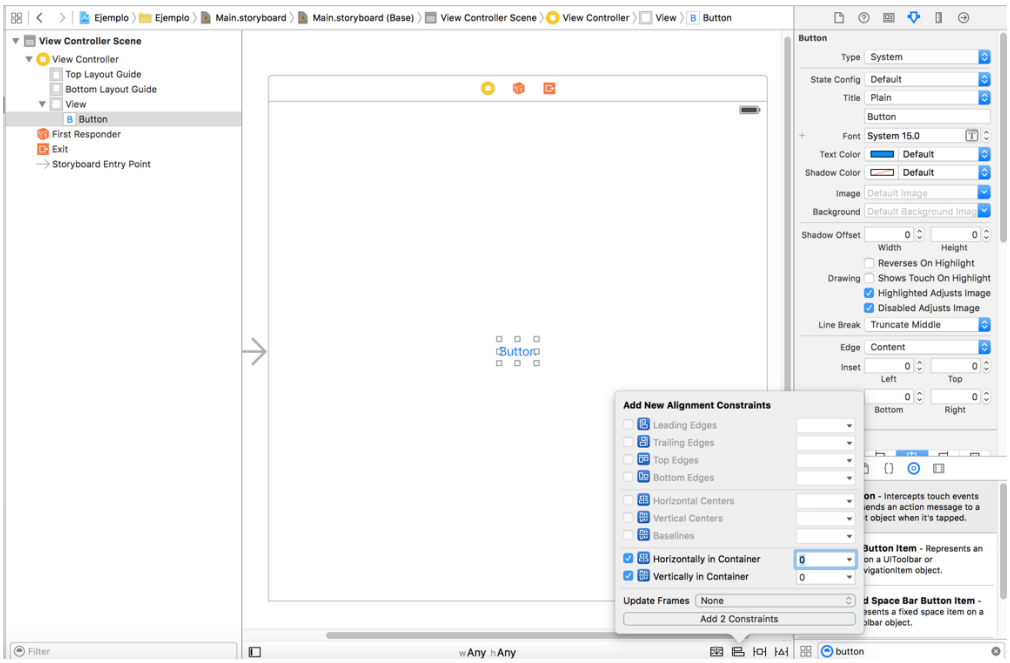


Ilustración 14. Ajuste horizontal y vertical

Una vez establecidas las restricciones, podemos ver en el editor como se ajusta sobre la pantalla.

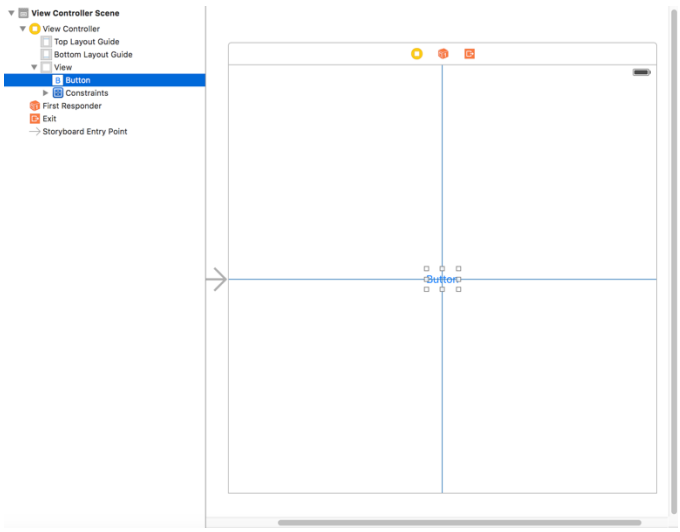


Ilustración 15. Ajuste botón sobre la pantalla

Xcode incluye un simulador en el que se dispone de simuladores de iPhone (desde el 4s hasta el 6s) y de iPad (desde iPad 2 hasta iPad Pro). Si ejecutamos la aplicación, por ejemplo, sobre el iPhone 4s comprobamos que aparece el botón donde habíamos colocado.

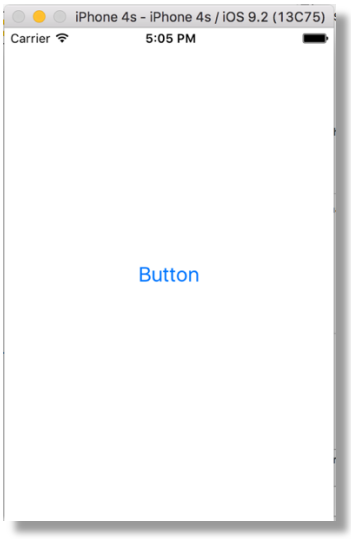


Ilustración 16. Ejecución aplicación simple

Xcode nos permite enviar mensajes a un objeto de la interfaz de usuario. Para ello se crea una conexión entre el objeto y un ***outlet*** del código.

Un ***outlet*** es un atributo marcado con el símbolo ***IBOutlet*** y cuyo valor se puede establecer gráficamente en el *storyboard*. Cuando queremos que un objeto (como el *view controller*) se comunique con otro objeto, designamos el objeto contenido como un ***outlet***.

Por ejemplo, vamos a introducir una etiqueta de texto en nuestra aplicación y vamos a enlazarla con el *view controller*. Para ellos seleccionamos la etiqueta pulsamos la tecla “Control” y arrastramos a la parte del código donde lo queremos introducir.

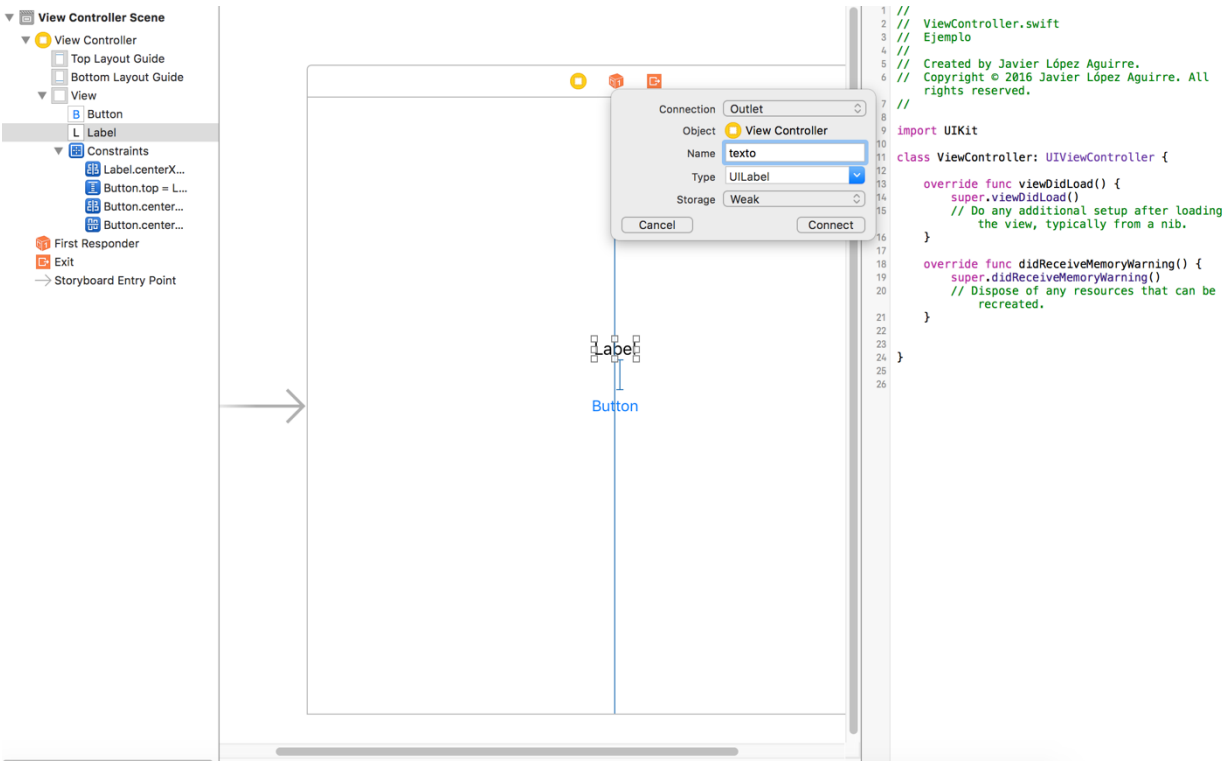


Ilustración 17. Outlet desde interfaz

En el código nos aparece la variable con la palabra reservada al principio, ***@IBOutlet***, este símbolo lo utiliza sólo Xcode para determinar si el atributo es un ***outlet***.

```
9  import UIKit
10
11 class ViewController: UIViewController {
12
13     @IBOutlet weak var texto: UILabel!
14
15     override func viewDidLoad() {
16         super.viewDidLoad()
17         // Do any additional setup after loading the view, t
18     }
19
20     override func didReceiveMemoryWarning() {
21         super.didReceiveMemoryWarning()
22         // Dispose of any resources that can be recreated.
23     }
24
25
26 }
```

Ilustración 18. Outlet en código.

Además, en la ventana derecha del editor podemos comprobar las referencias existentes entre la interfaz y el controlador.

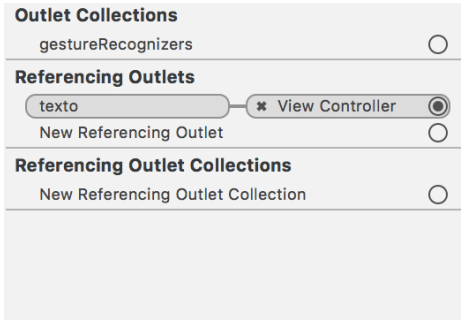


Ilustración 19. Ventana de outlets

También se pueden crear acciones para los elementos. Esto es necesario cuando se quiere enviar un mensaje a un objeto cuando ocurre un evento, como pulsar un botón. Al producirse el evento, se ejecutará la función enlazada.

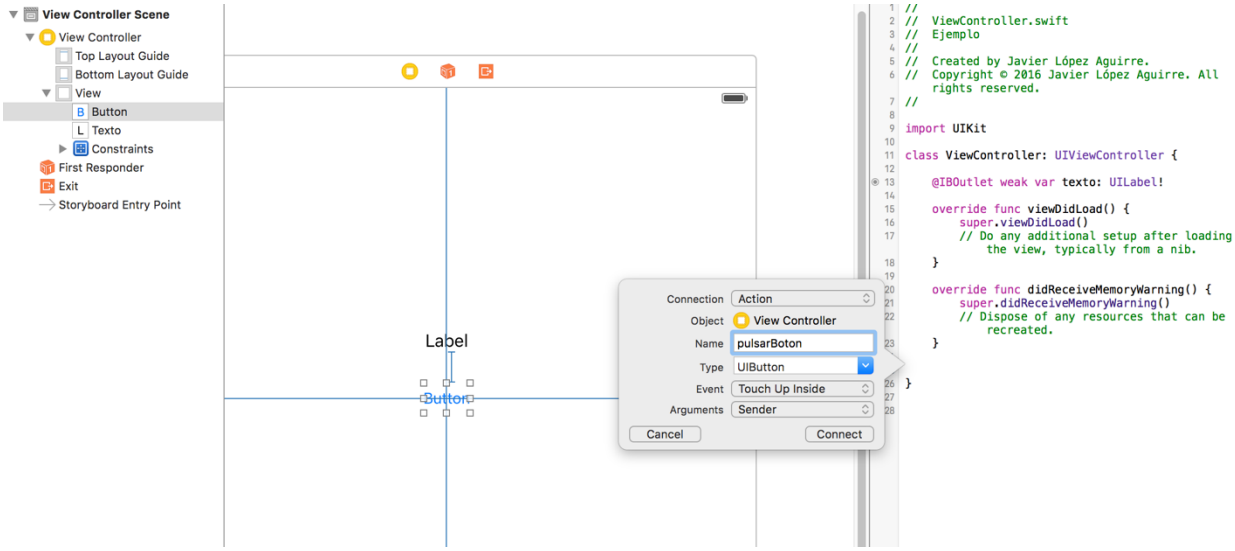


Ilustración 20. Conexión de una acción desde interfaz

En nuestro ejemplo, vamos a añadir una acción al botón para que cuando se pulse, cambie el texto de la etiqueta.

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var texto: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    @IBAction func pulsarBoton(sender: UIButton) {
        texto.text = "Hola mundo!"

    }

}
```

Ilustración 21. Conexión de una acción en el código

En el código nos aparece la función con la palabra reservada al principio, ***@IBAction***, para indicar que el método se puede conectar con un objeto de la interfaz. Para conseguir el cambio de texto, utilizamos la variable conectada anteriormente y cambiamos el valor de la propiedad "text".

Ejecutamos la aplicación y comprobamos que se cambia el texto al pulsar el botón.

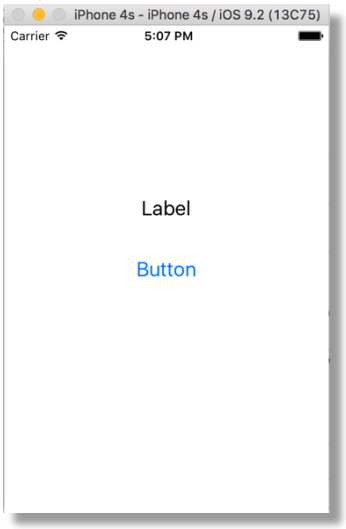


Ilustración 22. Ejecución aplicación con outlets (1)

Al pulsar el botón cambia el texto

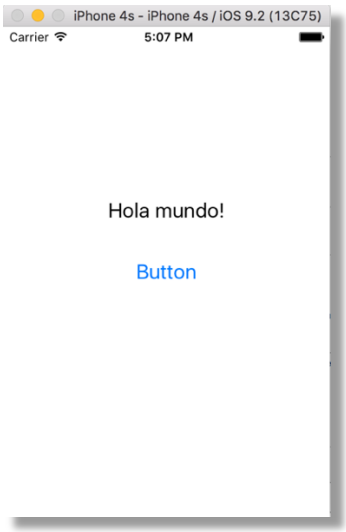


Ilustración 23. Ejecución aplicación con outlets (2)

Se ha visto como funciona iOS, las diferentes capas que lo componen con los frameworks que incorpora cada una y como se estructura una aplicación. Se ha justificado la elección Swift como lenguaje a utilizar y una breve introducción al IDE Xcode, el entorno de desarrollo gratuito de Apple.

5 REQUISITOS DEL SISTEMA

En este apartado se recogerá la especificación de requisitos del sistema. Permite entender lo que desea el cliente y el usuario, analizar las necesidades y evaluar la factibilidad. Además, ayuda a especificar la solución sin ambigüedades. Se describirá la información que se manejará, la funcionalidad y el comportamiento del sistema.

5.1 Requisitos generales del Sistema

A continuación se detallarán una serie de requisitos que se detallarán más posteriormente.

RG_01	Aplicación didáctica
Versión	1.0 (28/09/2015)
Descripción	El sistema a desarrollar deberá contener elementos que aporten información relevante para aprender un lenguaje de programación
Importancia	Alta
Comentarios	Esta información se le proporcionará al usuario de forma sencilla

Tabla 1. RG_01: Aplicación didáctica

RG_02	Niveles
Versión	1.0 (28/09/2015)
Descripción	El sistema dispondrá de una serie de niveles que irán aumentando el contenido y la dificultad
Importancia	Alta
Comentarios	En la interfaz principal aparecerán los niveles para seleccionarlos

Tabla 2. RG_02: Niveles

RG_03	Información de ayuda
Versión	1.0 (28/09/2015)
Descripción	El sistema debe disponer de mecanismos para ayudar al usuario a resolver los distintos niveles.
Importancia	Alta
Comentarios	Esta información se le proporcionará al usuario de forma sencilla

Tabla 3. RG_03: Información de ayuda

RG_04	Resolver niveles
Versión	1.0 (28/09/2015)

Descripción	El sistema deberá proporcionar herramientas para que el usuario resuelva los niveles a través de código que escriba
Importancia	Alta
Comentarios	Se presentará un teclado que ayude a conseguirlo

Tabla 4. RG_04: Resolver niveles

RG_05	Reiniciar sistema
Versión	1.0 (28/09/2015)
Descripción	El sistema podrá reiniciarse de forma que vuelva al estado de la primera ejecución
Importancia	Alta
Comentarios	Ninguno

Tabla 5. RG_05: Reiniciar sistema

RG_06	Seguridad
Versión	1.0 (28/09/2015)
Descripción	El sistema no podrá acceder a contenido existente en el dispositivo que no corresponda con la aplicación
Importancia	Alta
Comentarios	No solicitará permisos al usuario que proporcionen datos personales

Tabla 6. RG_06: Seguridad

RG_07	Portabilidad
Versión	1.0 (28/09/2015)
Descripción	El sistema podrá ejecutarse correctamente en los dispositivos que cumplan con las características necesarias establecidas
Importancia	Alta
Comentarios	Ninguno

Tabla 7. RG_07: Portabilidad

5.2 Requisitos funcionales del Sistema

Se definirán las funcionalidades que debe ofrecer el sistema a los usuarios para conseguir los objetivos establecidos.

5.2.1 Casos de uso

Describen interacciones entre el sistema y los usuarios que utilizan los servicios que ofrece el sistema.

5.2.1.1 Diagramas de casos de uso del Sistema

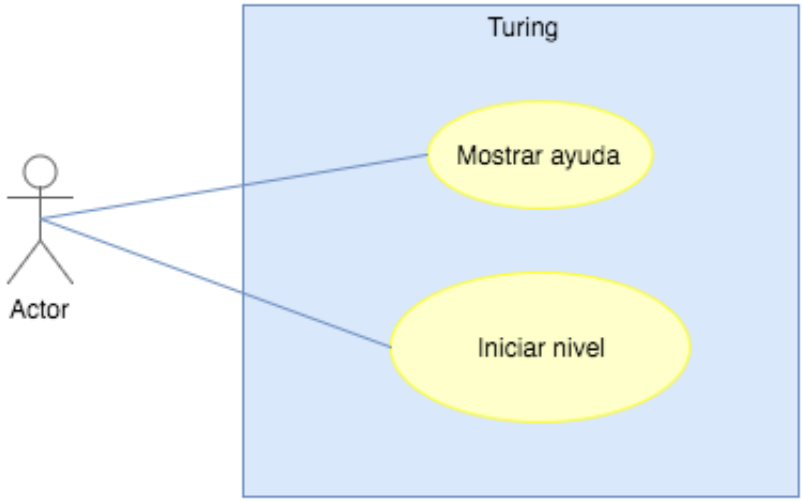


Ilustración 24. Diagrama 1 de casos de uso

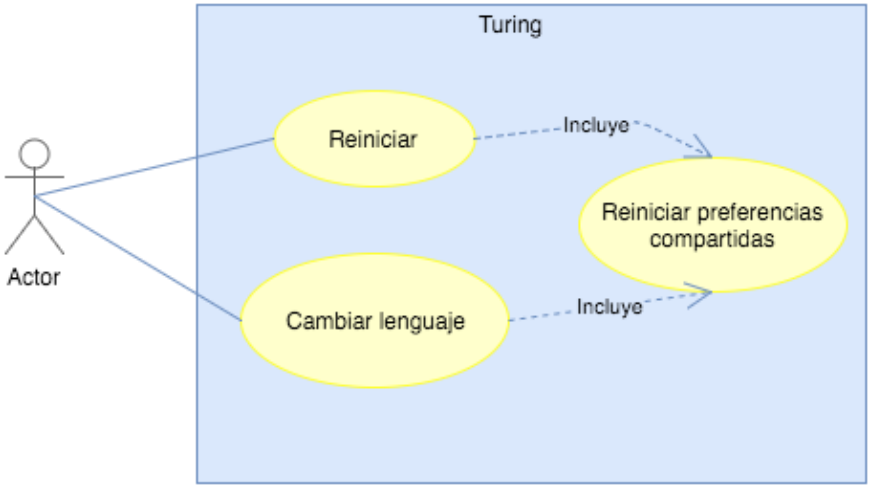


Ilustración 25. Diagrama 2 de casos de uso

5.2.1.2 Especificación de actores del Sistema

En el sistema habrá un sólo actor, que será el usuario final del sistema.

AC_01	Usuario
Versión	Versión 1.0 (15/09/2015)
Descripción	Este actor representa a un usuario final del sistema

Comentarios	Ninguno

Tabla 8. AC_01: Usuario

5.2.1.3 Especificación de casos de uso

CU_01	Mostrar ayuda	
Versión	1.0 (29/09/2015)	
Dependencias	• RG_01 • RG_03	
Precondición	El sistema almacenará información sobre ayudas al usuario	
Descripción	El sistema debe comportarse como se describe cuando el usuario solicite información de ayuda	
Secuencia Normal	**Paso**	**Acción**
	1	El usuario seleccionará leer la ayuda
	2	El sistema cargará el título y el texto de la ayuda
	3	Una vez realizado, se mostrará en la pantalla
Postcondición	El sistema cambiará de vista al seleccionar la ayuda	
Importancia	Alta	
Comentarios	Se podrá ir cambiando de ayudas en esta interfaz	

Tabla 9. CU_01: Mostrar ayuda

CU_02	Iniciar nivel	
Versión	1.0 (29/09/2015)	
Dependencias	• RG_01 • RG_02 • RG_04	
Precondición	El sistema almacenará información sobre varios niveles	
Descripción	El sistema debe comportarse como se describe cuando el usuario desee iniciar un nivel	
Secuencia Normal	**Paso**	**Acción**
	1	El usuario selecciona un nivel y lo inicia
	2	El sistema cargará los datos de ese nivel
	3	Se presentará el nivel al usuario
Postcondición	El sistema cambiará de vista al seleccionar el nivel	
Importancia	Alta	
Comentarios	El nivel comenzará con un diálogo	

Tabla 10. CU_02: Iniciar nivel

CU_03	Reiniciar preferencias compartidas	
Versión	1.0 (29/09/2015)	
Dependencias	• RG_05	
Precondición	El sistema almacenará información sobre el progreso del usuario	
Descripción	El sistema debe comportarse como se describe cuando sea necesario reiniciar la información de progreso del usuario	
Secuencia Normal	**Paso**	**Acción**
	1	El usuario reiniciará la aplicación o cambiará de lenguaje
	2	El sistema procederá a reiniciar las preferencias del usuario
	3	Una vez realizado, se ejecutará la aplicación desde el principio
Postcondición	El sistema debe ejecutarse como si fuera la primera vez que lo ha hecho el usuario	
Importancia	Alta	
Comentarios	El proceso será transparente al usuario	

Tabla 11. CU_03: Reiniciar preferencias compartidas

CU_04	Reiniciar	
Versión	1.0 (29/09/2015)	
Dependencias	• RG_05 • CU_03	
Precondición	El sistema almacenará información sobre el progreso del usuario	
Descripción	El sistema debe comportarse como se describe cuando el usuario desee reiniciar la información de progreso	
Secuencia Normal	**Paso**	**Acción**
	1	El usuario solicitará reiniciar la información de progreso
	2	El sistema procederá a reiniciar las preferencias del usuario
	3	Una vez realizado, se ejecutará la aplicación desde el principio
Postcondición	El sistema debe ejecutarse como si fuera la primera vez que lo ha hecho el usuario	
Importancia	Alta	
Comentarios	Se preguntará al usuario, antes de realizarlo, si está seguro de hacerlo	

Tabla 12. CU_04: Reiniciar

CU_05	Cambiar lenguaje
Versión	1.0 (29/09/2015)
Dependencias	• RG_05 • CU_03

Precondición	El sistema debe permitir la opción de elegir un lenguaje u otro	
Descripción	El sistema debe comportarse como se describe cuando el usuario desee cambiar de lenguaje de programación	
Secuencia Normal	**Paso**	**Acción**
	1	El usuario solicitará cambiar de lenguaje
	2	El sistema reiniciará las preferencias y establecerá el lenguaje elegido
	3	Una vez realizado, se ejecutará la aplicación desde el principio
Postcondición	El sistema debe ejecutarse como si fuera la primera vez que lo ha hecho el usuario	
Importancia	Alta	
Comentarios	Se preguntará al usuario, antes de realizarlo, si está seguro de hacerlo	

Tabla 13. CU_05: Cambiar lenguaje

5.2.2 Requisitos de información

Describe qué información debe almacenar el sistema para poder ofrecer los servicios necesarios.

RFI_01	Guardar datos del usuario
Versión	1.0 (28/09/2015)
Descripción	El sistema guardará la información necesaria para que el usuario, cuando vuelva a ejecutar la aplicación, se encuentre en el mismo estado que la última vez
Importancia	Alta
Comentarios	El estado que se guardará corresponde al progreso que lleve el usuario en la aplicación

Tabla 14. RFI_01: Guardar datos del usuario

RFI_02	Almacenar información de ayuda
Versión	1.0 (28/09/2015)
Descripción	El sistema almacenará información que ayude a al usuario a progresar en los niveles
Importancia	Alta
Comentarios	Esta información se le proporcionará al usuario de forma sencilla

Tabla 15. RFI_02: Almacenar información de ayuda

5.3 Requisitos no funcionales del Sistema

Definen las condiciones que se le imponen al sistema a desarrollar como aspectos de calidad o seguridad.

5.3.1 Requisitos de usabilidad

El sistema deberá solucionar las dificultades que se puedan encontrar los usuarios al utilizar la aplicación.

RNFU_01	Tutorial
Versión	1.0 (28/09/2015)
Descripción	Se incluirá un pequeño tutorial en el que se explique el funcionamiento de la aplicación
Importancia	Alta
Comentarios	Se lanzará la primera vez que se ejecute la aplicación

Tabla 16. RFNU_01: Tutorial

RNFU_02	Indicaciones de ayuda
Versión	1.0 (28/09/2015)
Descripción	Se añadirán indicaciones en la interfaz para ayudar de forma visual al usuario a realizar determinadas acciones
Importancia	Media
Comentarios	Las indicaciones básicas aparecerán una sola vez, siendo ésta la primera vez que aparezca

Tabla 17. RNFU_02: Indicaciones de ayuda

5.3.2 Requisitos de mantenibilidad

El sistema deberá permitir facilidades para ampliar la funcionalidad, modificar o corregir errores de la aplicación.

RNFM_01	Nuevos niveles
Versión	1.0 (28/09/2015)
Descripción	Se debe permitir la posibilidad de añadir nuevos niveles de forma sencilla
Importancia	Media
Comentarios	Se mantendrá el formato usado actualmente

Tabla 18. RNFM_01: Nuevos niveles

5.3.3 Requisitos de portabilidad

El sistema deberá ser compatible con los dispositivos que ejecuten el sistema operativo iOS.

RNFP_01	Compatibilidad con otros dispositivos
Versión	1.0 (28/09/2015)
Descripción	La aplicación debe poder ejecutarse en dispositivos que dispongan de iOS 8.0 o posterior.
Importancia	Alta
Comentarios	Ninguno

Tabla 19. RNFP_01: Compatibilidad con otros dispositivos

RNFP_02	Adaptabilidad a los dispositivos
Versión	1.0 (28/09/2015)
Descripción	El sistema deberá responder de la misma forma en los distintos dispositivos compatibles
Importancia	Alta
Comentarios	Se adaptará a características del dispositivo, como el tamaño de la pantalla, para que se pueda utilizar correctamente

Tabla 20. RNFP_02: Adaptabilidad a los dispositivos

5.3.4 Requisitos de seguridad

El sistema cumplirá aspectos relacionados con la protección de datos y la privacidad de los usuarios.

RNFS_01	Privacidad del usuario
Versión	1.0 (28/09/2015)
Descripción	El sistema no accederá a ningún contenido privado del usuario
Importancia	Alta
Comentarios	No se accederá ni a la localización, a archivos o terceras aplicaciones que disponga el usuario

Tabla 21. RNFS_01: Privacidad del usuario

Se han analizado los requisitos, se han establecido algunas características que deben disponer el sistema, el comportamiento en determinadas situaciones y algunas restricciones.

6 DISEÑO

En este apartado se va a desarrollar el diseño del sistema. Se detallará la estructura, las interfaces y el comportamiento del sistema.

6.1 Clases

Primero vamos a mostrar un diagrama de clases del sistema para ver cómo interactúan las distintas clases. A continuación, se detallará cada una de las clases.

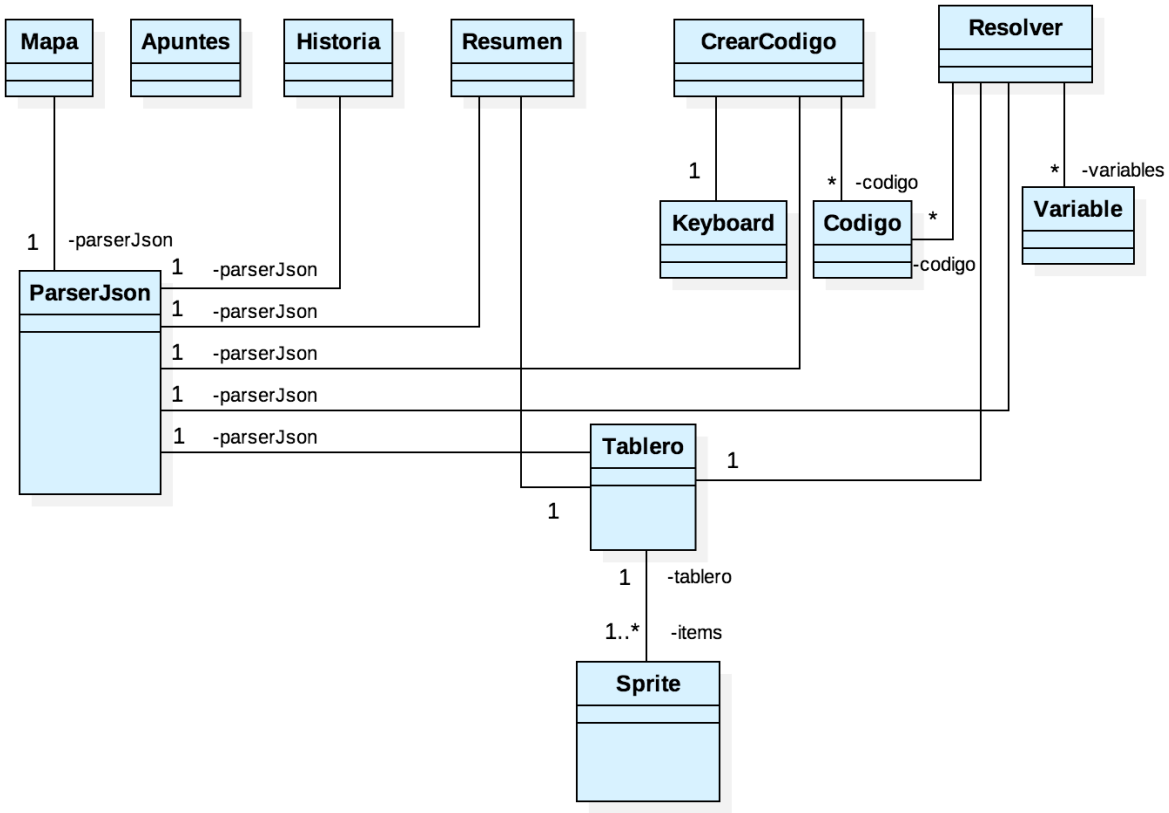


Ilustración 26. Diagrama de clases

6.1.1 Clase Mapa

Esta clase es la principal del programa. Se carga al principio, y muestra una especie de menú en el que se pueden seleccionar los niveles, los ajustes o la ayuda desbloqueada.

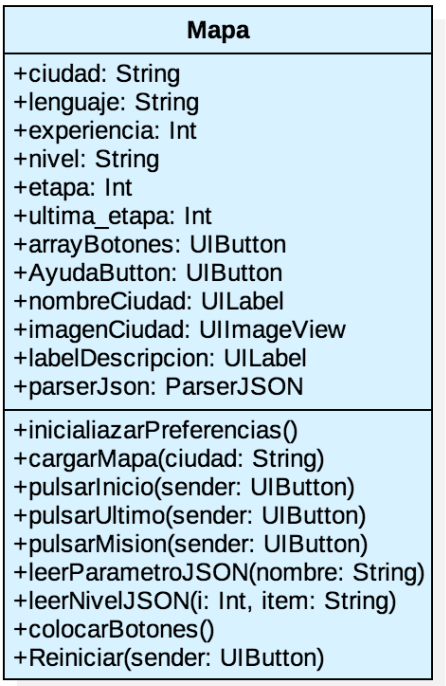


Ilustración 27. Clase Mapa

6.1.2 Clase Apuntes

Esta clase se encarga de mostrar las ayudas desbloqueadas. Muestra el título y la descripción de la ayuda, para ello, utiliza un UIWebView en el que se escribirá el texto en html. Se podrá ir pasando de una ayuda a otra mediante botones que irán hacia la página siguiente o yendo directamente a la ayuda deseada.

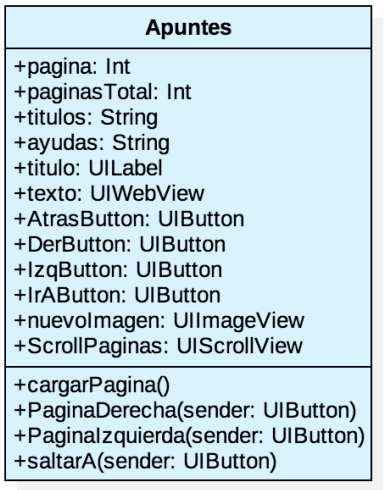


Ilustración 28. Clase Apuntes

6.1.3 Clase Historia

Esta clase se encargara de mostrar un dialogo entre dos personas. Para ello, se mostrara un fondo y las imágenes de las dos personas, y una especie de chat.

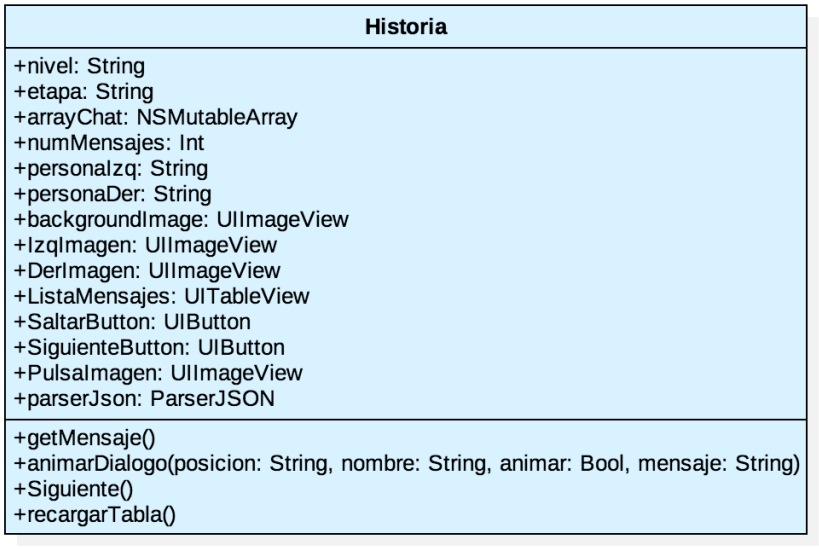


Ilustración 29. Clase Historia

6.1.4 Clase Resumen

Esta clase muestra un tablero con el contenido del nivel. Mostrará también un pequeño resumen del nivel, el objetivo primario y el secundario, si existe.

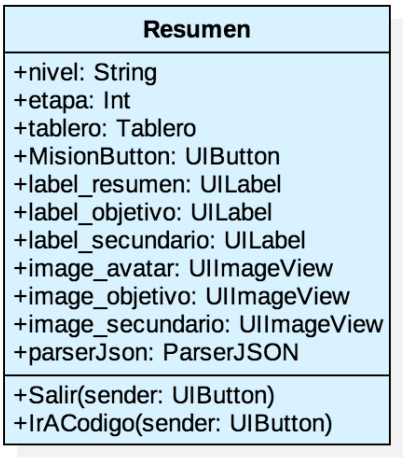


Ilustración 30. Clase Resumen

6.1.5 Clase CrearCodigo

Esta clase se encarga de escribir el código necesario para resolver el nivel. Incluye un teclado y una tabla en la que se muestra el código escrito. Además, permite volver hacia el resumen, para ver el tablero, y ver la ayuda disponible para el nivel.

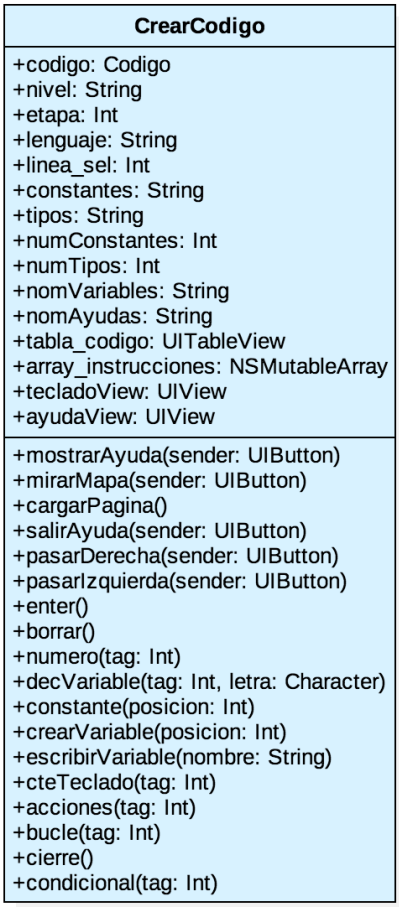


Ilustración 31. Clase CrearCodigo

6.1.6 Clase Resolver

Esta clase se encarga de leer el código y mostrar en el tablero los resultados. Si no se consigue pasar el nivel, permite volver hacia atrás para arreglar el código. Si se resuelve, se avanza hacia la siguiente etapa o al menú.

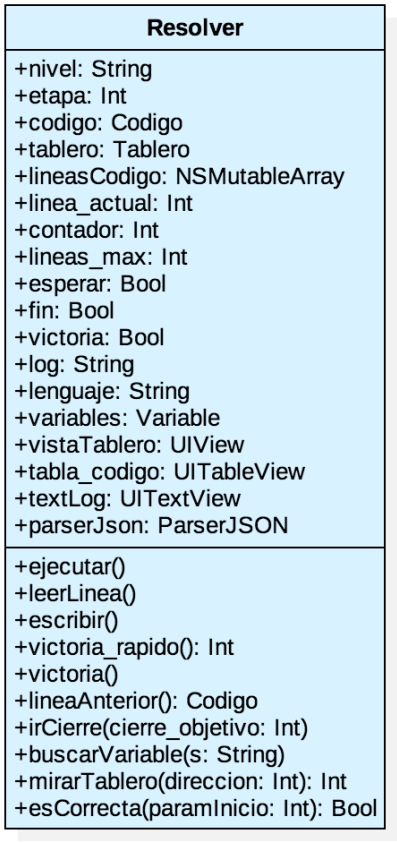


Ilustración 32. Clase Resolver

6.1.7 Clase Variable

Cuando en el código se crea una variable, se crea un objeto de este tipo.

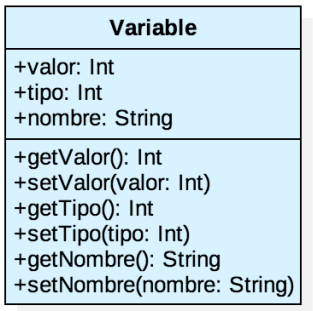


Ilustración 33. Clase Variable

6.1.8 Clase Codigo

Esta clase representa una línea escrita por el jugador. Se encarga de almacenar lo que se ha escrito y de convertirlo en una cadena, para mostrarlo en pantalla con el formato correcto.

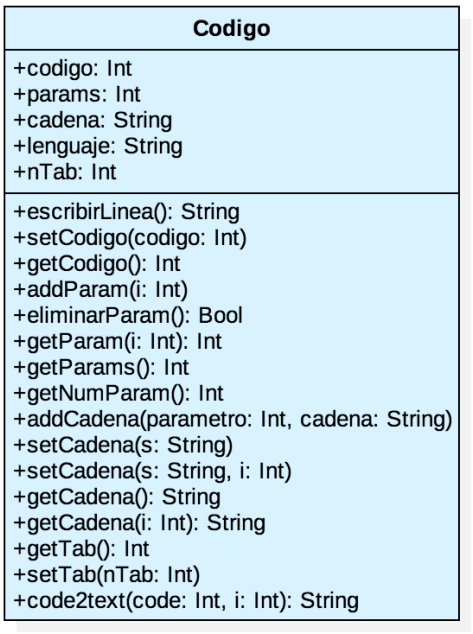


Ilustración 34. Clase Codigo

6.1.9 Clase Tablero

Esta clase dibuja el tablero en el que se va a desarrollar el juego. Coloca las baldosas y los sprites.

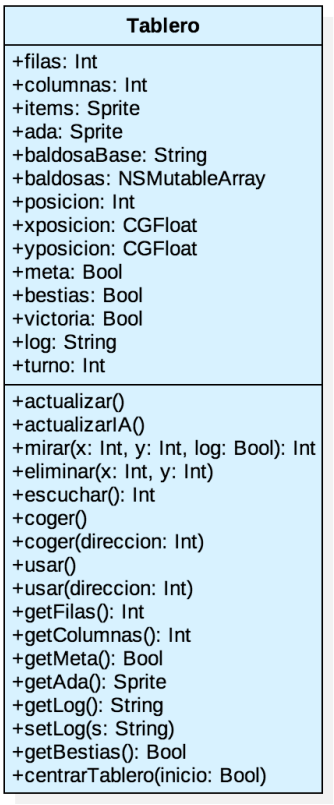


Ilustración 35. Clase Tablero

6.1.10 Clase Sprite

Esta clase se encarga de dibujar los elementos que existen en el nivel.

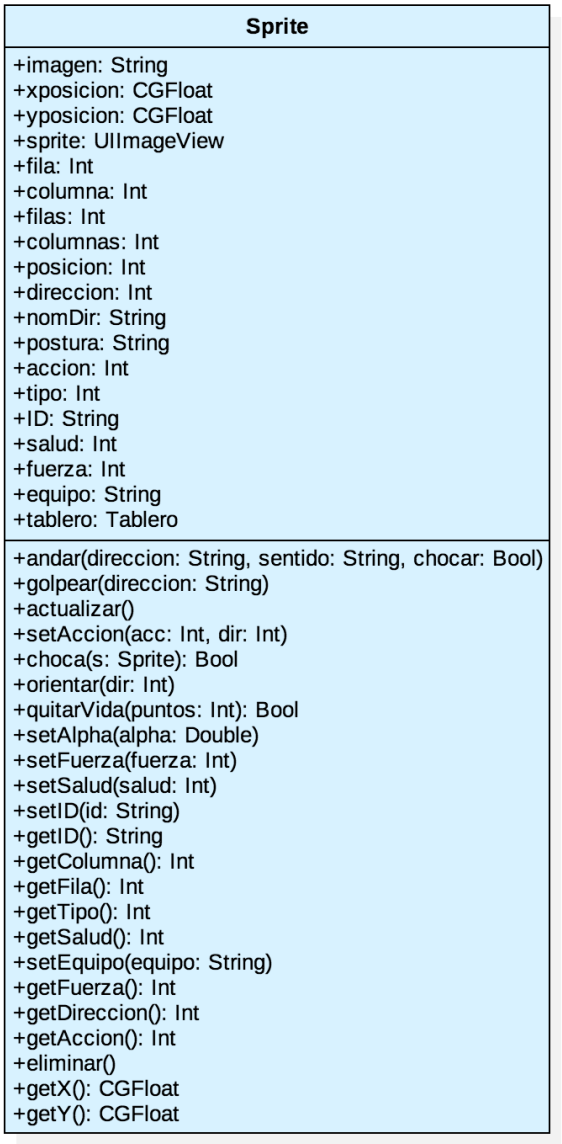


Ilustración 36. Clase Sprite

6.1.11 Clase ParserJson

Esta clase es la encargada de leer el archivo JSON y extraer información sobre los niveles.

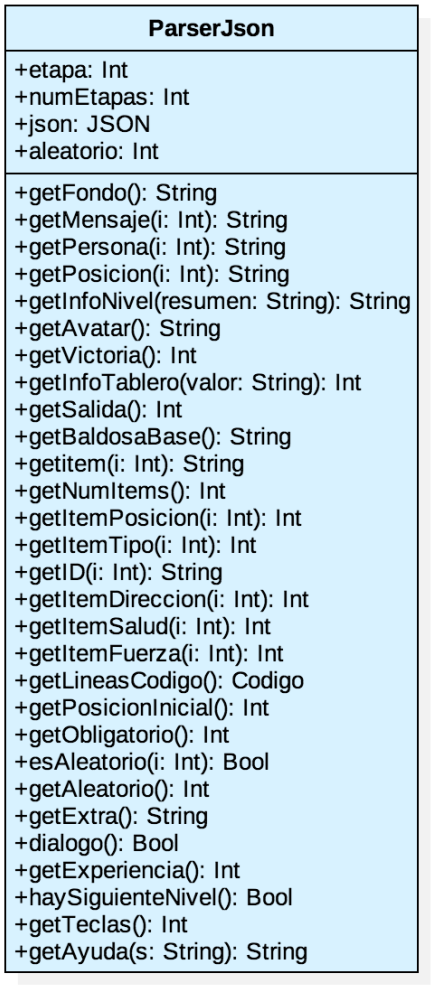


Ilustración 37. Clase ParserJson

6.1.12 Clase Keyboard

Esta clase se encarga de gestionar el teclado que permite escribir el código.

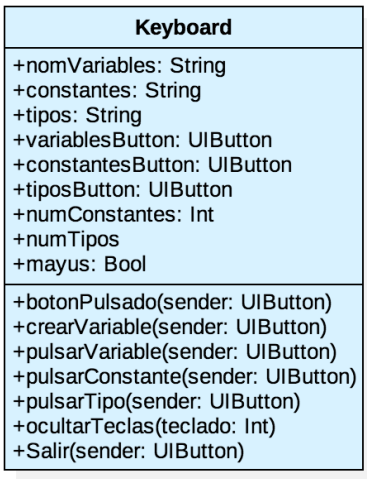


Ilustración 38. Clase Keyboard

6.2 Interfaz gráfica

En esta sección se va a detallar los prototipos de las interfaces de las distintas vistas que componen el sistema.

6.2.1 Interfaz Mapa

Esta vista será la principal del programa. Se cargará al ejecutar la aplicación y se podrán ejecutar distintas acciones como comenzar un nivel o cambiar a la vista en la que podemos leer las ayudas. En esta vista se verá el título de la ciudad junto con una imagen que la identifique, se mostrará una lista de niveles junto con una descripción de ellos. Además, existirán botones que permitan realizar ajustes o cambiar de vista.

Ilustración 39. Interfaz Mapa

6.2.2 Interfaz Apuntes

Esta interfaz se accederá desde la vista del mapa. Mostrará un título y un texto que explicará un aspecto de la aplicación o de la programación. Existirá un botón que permita saltar a la ayuda que se desea y dos botones para pasar hacia delante o hacia atrás.

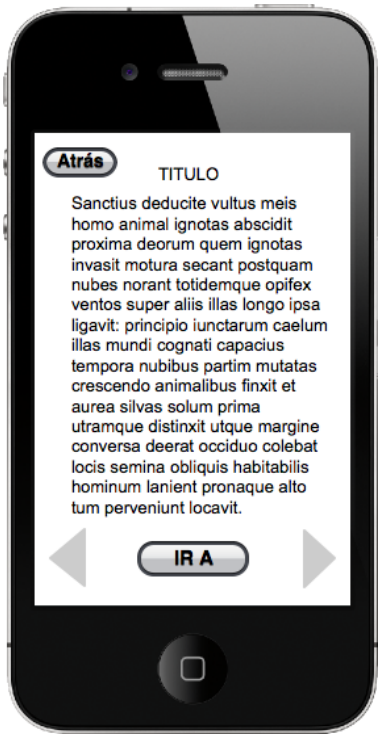


Ilustración 40. Interfaz Apuntes

6.2.3 Interfaz Historia

Antes de comenzar el nivel se mostrará un diálogo. Esta interfaz lo mostrará mediante dos imágenes de las personas que hablen y un chat. Tras el último mensaje, aparecerá un botón para pasar a la siguiente vista.

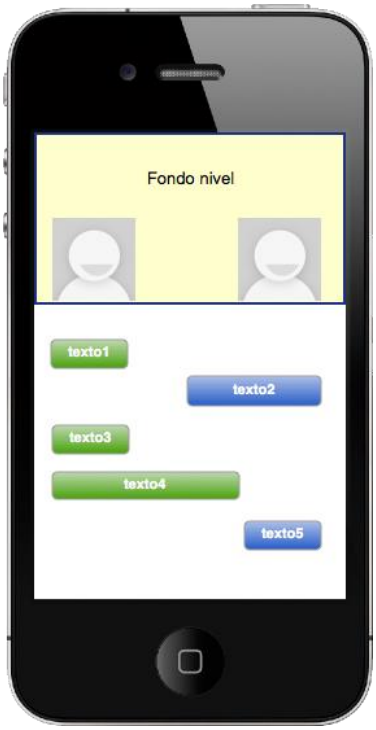


Ilustración 41. Interfaz Historia

6.2.4 Interfaz Resumen

Una vez terminado el dialogo, comenzará el nivel mostrando el tablero, un resumen y los objetivos a conseguir.

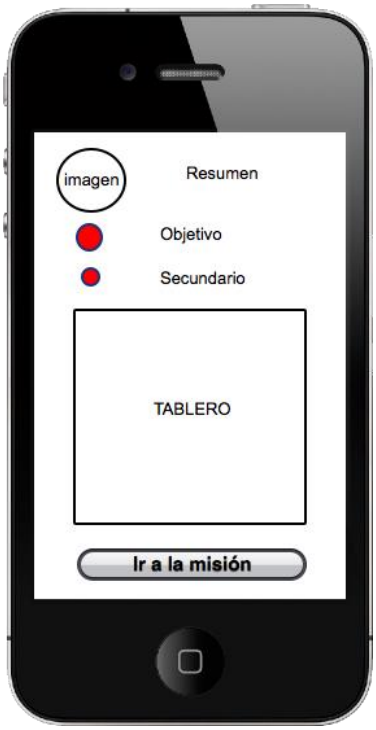


Ilustración 42. Interfaz Resumen

6.2.5 Interfaz CrearCodigo

En esta interfaz será donde se escriba el código con el que se resolverá el nivel. Tendrá un teclado personalizado y las distintas líneas de código. Además, se podrá mostrar la ayuda específica de ese nivel pulsando un botón, volver atrás para ver el tablero o pasar a resolver el nivel.

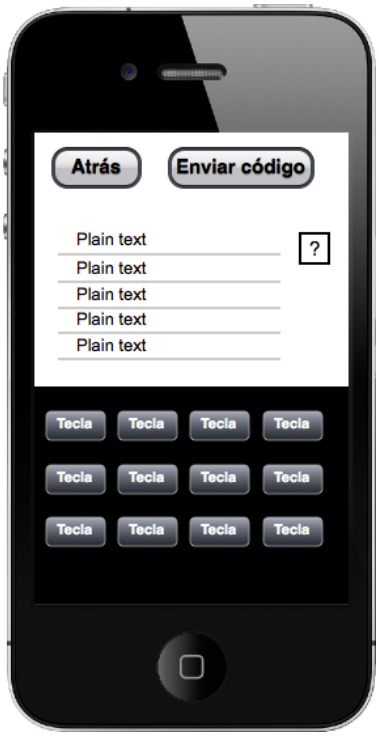


Ilustración 43. Interfaz CrearCodigo

6.2.6 Interfaz Resolver

Esta será la última interfaz en la que se mostrará el resultado del código escrito. En la parte superior, aparecerá un texto en el que se irá indicando el progreso del nivel. En la parte inferior se muestra la línea del código que se está ejecutando actualmente, cada vez que se ejecute una, el tablero reflejará los cambios producidos.

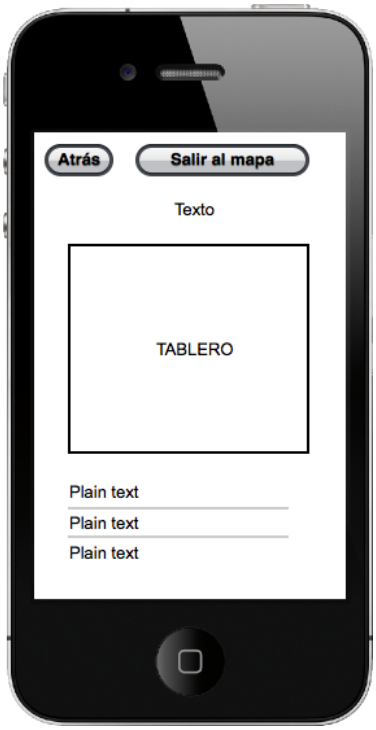


Ilustración 44. Interfaz Resolver

6.3 Comportamiento del sistema

A continuación se va a detallar el comportamiento del sistema mediante la ayuda de diversos diagramas.

6.3.1 Arranque del sistema

Este diagrama muestra el arranque del sistema. Si es la primera vez que se ejecuta la aplicación, hay que inicializar las preferencias del usuario y mostrar un tutorial. Si no, se muestra el mapa en el que se pueden elegir los niveles.

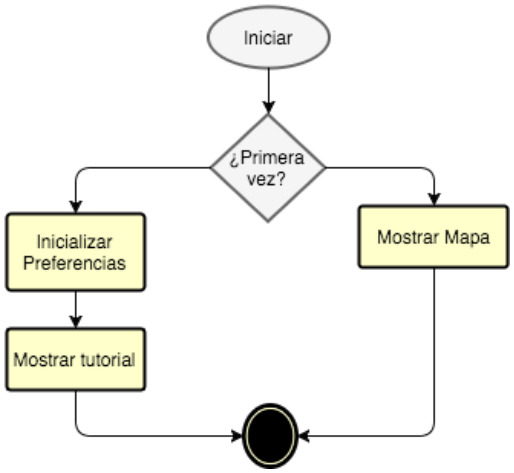


Ilustración 45. Diagrama arranque aplicación

6.3.2 Diálogo

En este diagrama se va a detallar el proceso de lectura de mensajes del dialogo. Cada vez que se pulse la pantalla, se cargará un nuevo mensaje. Según el mensaje corresponda a la persona que aparece a la izquierda o a la derecha, se animará la persona correspondiente y el mensaje aparecerá en ese lado. Tras mostrar el último mensaje, se podrá pasar a la siguiente vista.

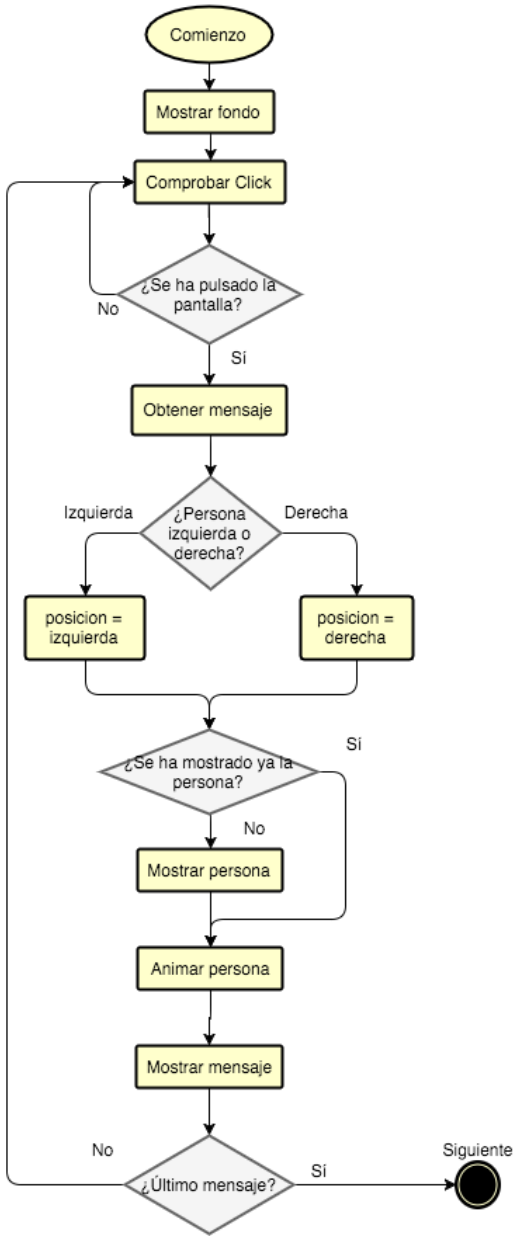


Ilustración 46. Diagrama diálogo

6.3.3 Resolver nivel

Este diagrama se divide en dos partes. La primera corresponde a una comprobación que realiza el sistema para ver si el código escrito por el usuario resuelve el problema, pero probando todos los valores aleatorios. Se ejecutará el nivel varias veces variando el valor del componente aleatorio. Si hay alguna variante que no se resuelva, se forzará que sea esa la que se muestre al usuario.

Una vez se haya realizado la comprobación, se mostrará el tablero al usuario y se esperará a que se toque la pantalla. Cada vez que se pulse, se leerá una línea y se ejecutará mostrando el resultado en pantalla, al finalizar la ejecución se comprobará si se ha ganado. Si es así, se termina la ejecución, si no, se comprueba si es la última línea. Si lo es, se termina la ejecución, si no, se vuelve a esperar a que se pulse la pantalla.

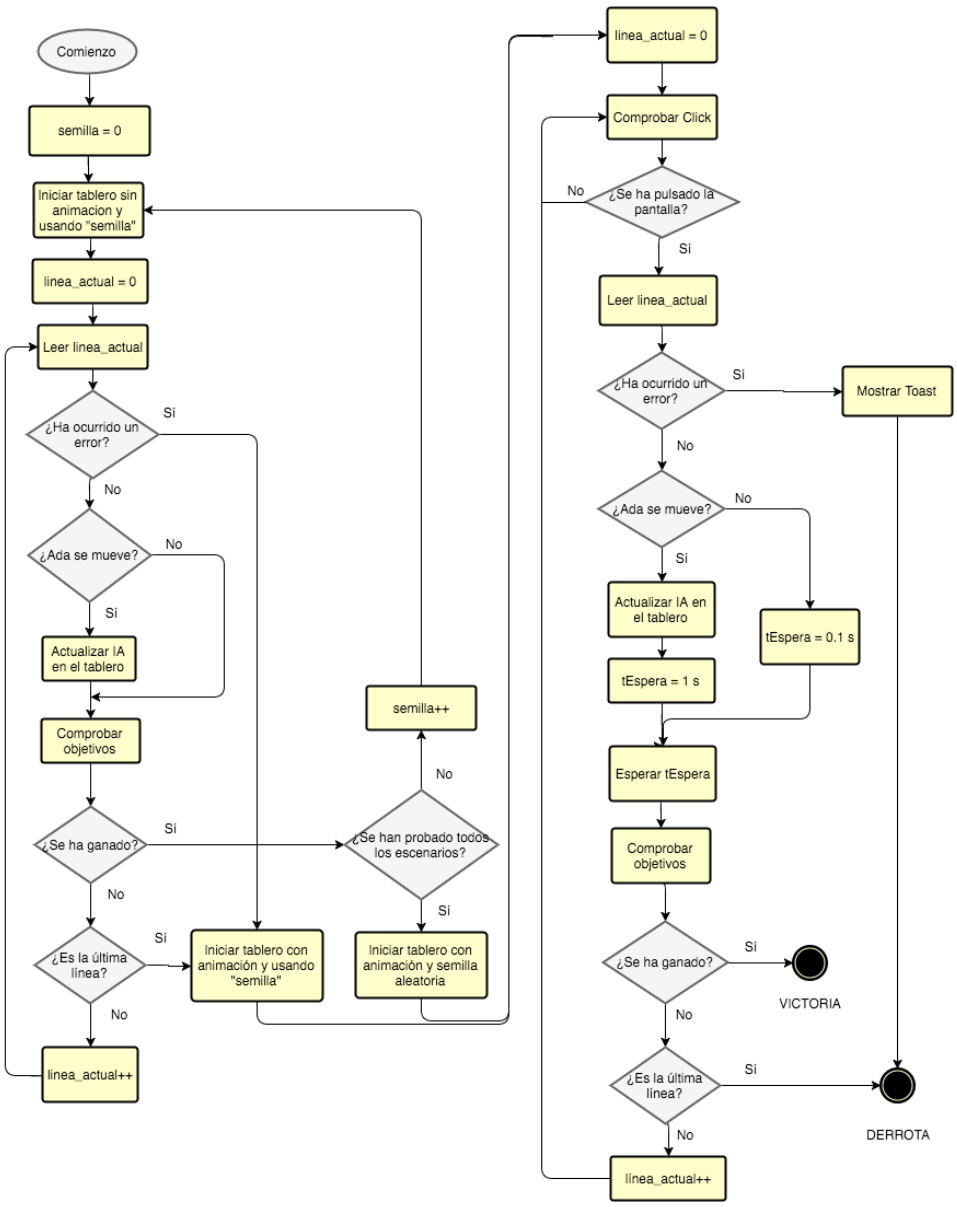


Ilustración 47. Diagrama resolver nivel

6.3.4 Fin nivel

En este diagrama se detalla el comportamiento cuando se pulsa el botón que permite finalizar el nivel. Si se pulsa y todavía no ha finalizado el nivel, se saldrá al mapa. Si ha finalizado, se comprueba si hemos ganado, saliendo al mapa si no lo hemos conseguido. Si se ha ganado, se comprueba si es la última etapa, si lo es se sale al mapa, si no se pasa a la siguiente etapa.

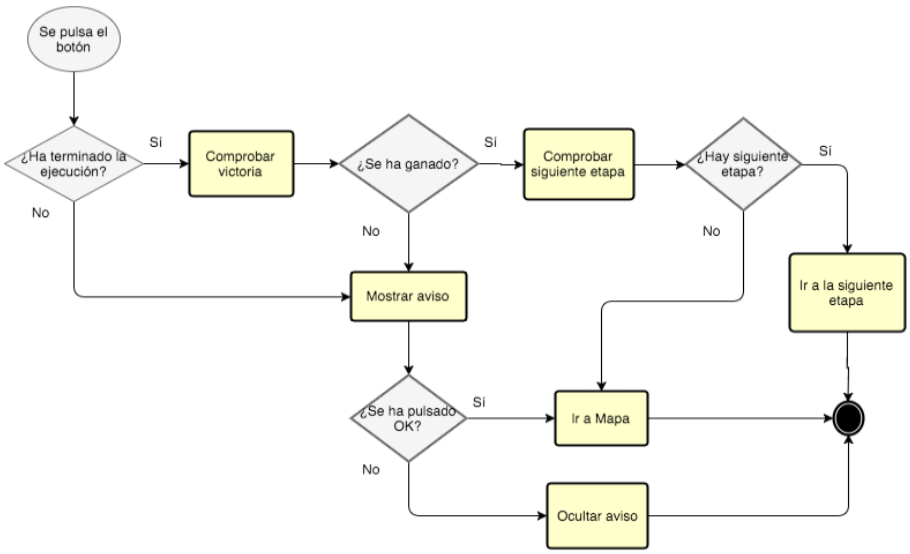


Ilustración 48. Diagrama fin de nivel

En esta sección se ha realizado el diseño del sistema, se han definido las distintas clases que componen el sistema y la relación entre ellas. También, se ha obtenido un prototipo de las diferentes interfaces de la aplicación y por último, se ha detallado el comportamiento de la aplicación en las principales vistas.

7 IMPLEMENTACIÓN

En este apartado se va a explicar cómo se ha implementado el sistema.

7.1 Interfaz gráfica

Primero vamos a detallar como se ha realizado la interfaz gráfica de la aplicación. Como se vio en el diseño, disponemos de varias vistas y cada una de ellas dispondrá de un controlador.

Para diseñar las vistas, utilizamos la herramienta que proporciona Xcode, el Storyboard. Es una representación visual de todas las vistas de la aplicación. Contiene las escenas, que pueden ser vistas simples o controladores de vistas, y las transiciones entre estas vistas.

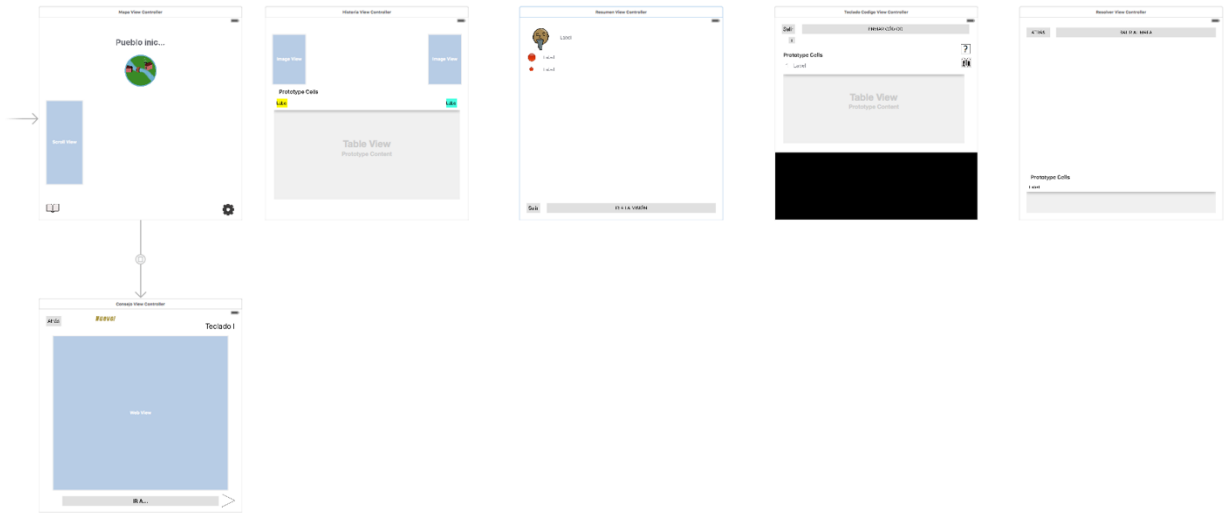


Ilustración 49. Vistas de la aplicación.

Se va a detallar la vista en la que se escribe el código. Tenemos elementos como botones, textos, una tabla y una vista en la que se colocará el teclado.

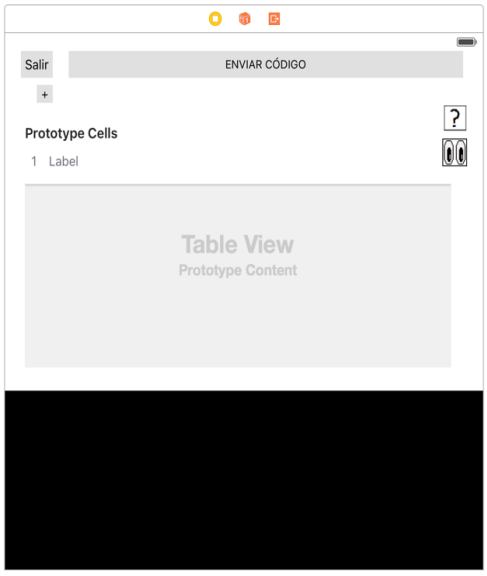


Ilustración 50. Interfaz de vista en Xcode

En la interfaz vemos que se han colocado diversos botones. En algunos se ha puesto una imagen sobre el botón, como el botón que muestra la ayuda en el que se ha puesto la imagen *boton_ayuda.png*.

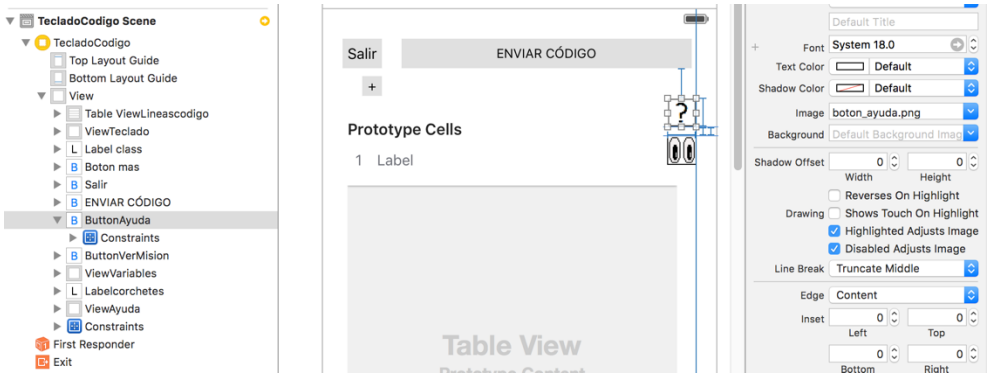


Ilustración 51. Detalle botón en Xcode

Cada uno de los botones realiza una función que se detalla en el controlador de la vista. Por ejemplo, el botón que se utiliza para mostrar la ayuda, representado por la imagen con una interrogación.

```
// Boton ayuda
@IBAction func mostrarAyuda(sender: UIButton) {

    // Cargar pagina
    nombreAyudas = parserJson.getAyuda("ayuda")
    if (!nombreAyudas.isEmpty){
        // Se muestra ayuda
        ayudaView.hidden = false

        ayudaView.layer.borderWidth = 1
        ayudaView.layer.borderColor = UIColor.blackColor().CGColor

        // Cargar pagina
        pagina = 0
        cargarPagina()
    }
}
```

Ilustración 52. Función mostrarAyuda

En este código podemos observar algunos detalles como el prefijo ***@IBAction***, que indica una acción que se

ejecuta cuando ocurre algo en la vista. El prefijo ***IB*** indica que se ha creado con el *Interface Builder*. El parámetro sender nos devuelve el botón que ha sido pulsado.

A continuación se muestra como queda la vista en ejecución.

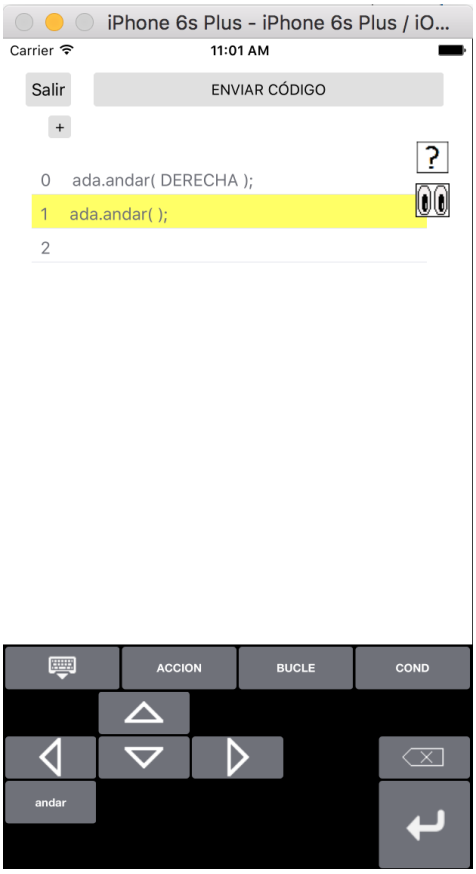


Ilustración 53. Vista de la clase CrearCodigo

7.1.1 UITableView

Este elemento permite mostrar información en una tabla. En la vista comentada, se ha utilizado para mostrar las líneas de código. Está compuesta de celdas, cada una de las celdas se han personalizado para que muestre dos textos, el número de la línea más el código.

Para rellenar la tabla con datos es necesario implementar tres métodos:

- **numberOfSectionsInTableView**: Indica a la vista de la tabla cuántas secciones debe mostrar.
- **tableView: numberOfRowsInSection**: Indica a la vista de la tabla cuántas columnas debe mostrar en la sección indicada.
- **tableView: cellForRowAtIndexPath**: Se muestra la información deseada para la celda en la fila indicada.

A continuación mostramos el método **tableView: cellForRowAtIndexPath** implementado en la tabla. Primero se obtiene la celda personalizada y el texto, luego se coloca en la celda el texto y alguna personalización adicional como el fondo de la celda.

```
//Se establece el texto de la celda
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {

    //Celda personalizada
    let cell: CeldaCodigo = self.tabla_codigo.dequeueReusableCellWithIdentifier(CellIdentifier, forIndexPath: indexPath) as! CeldaCodigo

    // Se obtiene texto
    let texto = array_instrucciones[indexPath.row]["texto"] as! String

    //texto linea y numero de linea
    cell.label_codigo.text = texto
    cell.id_label.text = String(indexPath.row)

    //Ajustamos color del fondo de la celda al seleccionarla
    let custombackground = UIView()
   // custombackground.backgroundColor = UIColor.yellowColor()
    custombackground.backgroundColor = UIColor(red: 255.0/255.0, green: 255.0/255.0, blue: 102.0/255.0, alpha: 1)
    cell.selectedBackgroundView? = custombackground

    // comprobar si existe ya esa celda de antes para borrar linea inferior
    for layer in cell.layer.sublayers! {
        if(layer.name == String(indexPath.row)){
            layer.removeFromSuperlayer()
        }
    }

    //Separador celdas
    let bottomLine = CALayer()
    bottomLine.frame = CGRectMake(0.0, cell.frame.height - 1, cell.frame.width, 0.75)
    bottomLine.backgroundColor = UIColor(red: 227.0/255.0, green: 229.0/255.0, blue: 236.0/255.0, alpha: 1).CGColor
    bottomLine.name = String(indexPath.row)
    cell.layer.addSublayer(bottomLine)

    return cell
}
```

Ilustración 54. Colocar contenido en UITableView

Para conectar el controlador con los elementos de estas celdas, es necesario un archivo que contenga las referencias. Los dos textos se conectan mediante un *outlet*, como vemos aparece el prefijo ***@IBOutlet*** y la referencia ***weak***, que utiliza el compilador para gestionar la memoria.

```
class CeldaCodigo: UITableViewCell {

    @IBOutlet weak var id_label: RDLabel!
    @IBOutlet weak var label_codigo: RDLabel!
}
```

Ilustración 55. Código celda personalizada

7.1.2 Teclado

En la parte inferior se ha colocado una vista en la que aparecerá el teclado. Se han diseñado varios teclados para los distintos elementos de código.

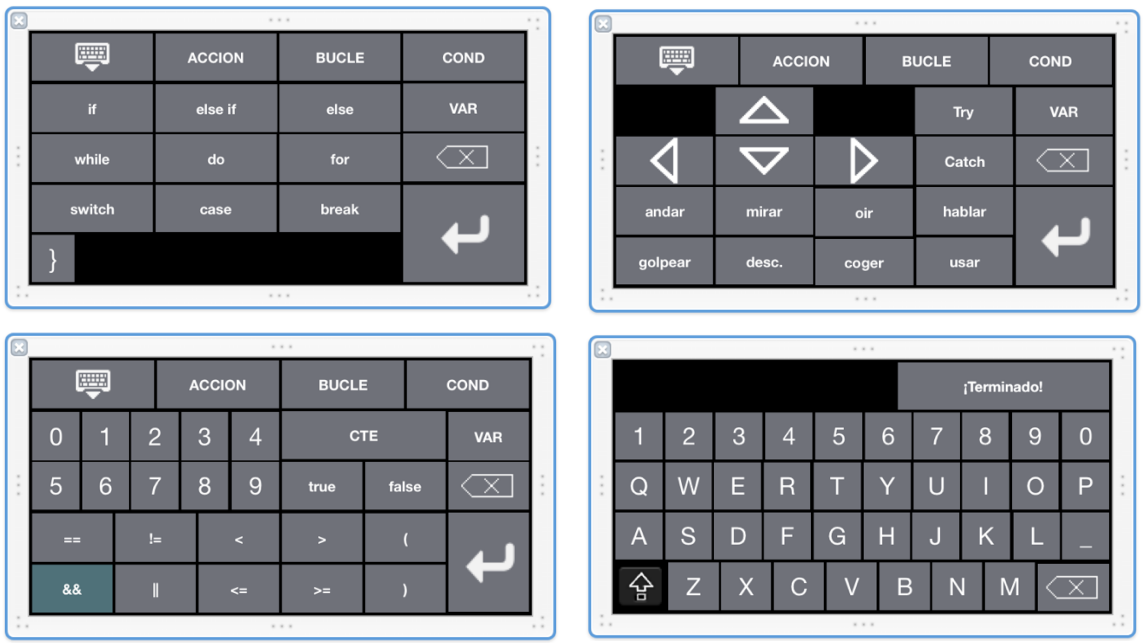


Ilustración 56. Teclados

El teclado se cargará en la vista que se ha colocado. Para ello, se obtiene una vista que muestre la interfaz diseñada y haga uso de la clase *Keyboard*.

```
// Cargamos teclado
if let custView = NSBundle.mainBundle().loadNibNamed("Keyboard1", owner: nil, options: nil).first
    as? Keyboard {
    customView = custView


    //Ajustamos alto y ancho para que llenen view
    customView.frame = CGRectMake(0,0,tecladoView.frame.width,tecladoView.frame.height)
    //Añadimos a la vista
    tecladoView.addSubview(customView)
    custView.ocultarTeclas(1)
}
```

Ilustración 57. Cargar teclado

Cada botón estará manejado por la clase *Keyboard*. Los botones del teclado dispondrán de un atributo en el que cada uno tendrá un número. Esta clase detectará que se ha pulsado un botón y actuará de distinta forma según cual haya sido.

```
@IBAction func buttonClicked(sender: UIButton)
{
    switch sender.tag {

    //Ocultar teclado
    case -3:
        keyboardView.hidden = true
        //Borramos de la vista
        customView.removeFromSuperview()
        // Se pone a cero para desplazar la tabla hacia abajo
        alturateclado.constant = 0
        //Ocultamos vista por si hubiera algo
        varView.hidden = true

        //Borramos botones de constantes anteriores si existen
        if (!botonesCte.isEmpty){
            for i in botonesCte {
                i.removeFromSuperview()
            }
        }
```

Ilustración 58. Detectar botón pulsado en el teclado.

Para ocultar o cambiar de teclado, habrá que pulsar uno de los botones que aparece en la parte superior del teclado, estos botones se mantendrán en los tres teclados principales. El cuarto teclado aparecerá solo cuando se vaya a escribir una variable.

```
// Cambiar a teclado 2
case -11: if let custView = NSBundle.mainBundle().loadNibNamed("Keyboard2", owner: nil, options: nil).first
    as? Keyboard {
    customView.removeFromSuperview()
    //Ajustamos alto y ancho para que llenen view
    customView = custView
    customView.frame = CGRectMake(0,0,keyboardView.frame.width,keyboardView.frame.height)
    keyboardView.addSubview(customView)
```

Ilustración 59. Cambiar de teclado

Mientras se pulse un botón se resaltará cambiando el fondo del mismo por un color azul.

```
// Se llama a iniciobutton on touchdown y a soltarbutton en touchupinseide
@IBAction func inicioButton(sender: UIButton) {
    sender.highlighted = false
    sender.backgroundColor = UIColor(red: 0.0/255.0, green: 160.0/255.0, blue: 255.0/255.0, alpha: 1.0)
}

@IBAction func soltarButton(sender: UIButton) {
    sender.highlighted = false
    if sender.tag != 52{
        sender.backgroundColor = UIColor(red: 111.0/255.0, green: 113.0/255.0, blue: 121.0/255.0, alpha: 1.0)
    }
    else {
        sender.backgroundColor = UIColor(red: 0.0/255.0, green: 0.0/255.0, blue: 0.0/255.0, alpha: 1.0)
    }
}
```

Ilustración 60. Cambio de fondo de un botón

Según se vaya avanzando en el juego, se irán desbloqueando teclas nuevas que permitirán escribir un código mas complejo. Por ello, la clase *Keyboard* tendrá un método "OcultarTeclas" que se llamará cada vez que se carga el teclado. Este método busca las teclas desbloqueadas, leyendo las preferencias del usuario, y oculta las que no lo están.

```
//Si la tecla no está desbloqueada (no está esta preferencia) se
//cambia la altura de la tecla a 0 y se elimina el nombre
// for var i = 0; i <= maxTeclas; i += 1 {
for i in 0 ..< maxTeclas+1 {
    if(!preferences.boolForKey("tecla_\(teclado)_\(i)")) {
        customView.subviews[i].hidden = true
    }
}
```

Ilustración 61. Ocultar teclas

7.1.3 UIWebView

La aplicación mostrará ayudas al usuario, el texto de estas se codificará en un archivo HTML Para mostrar este contenido se va a utilizar un *UIWebView*. Este elemento nos permite incluir contenido web en nuestra aplicación.

7.1.3.1 Archivos plist

El texto HTML se va a almacenar en un archivo *plist*. Este archivo nos permite guardar información que podemos usar en la aplicación, normalmente aparece en formato XML. Guarda la información de forma jerárquica y está compuesto por claves, que sirven como identificadores, y valores que se pueden acceder desde el código posteriormente.

Xcode nos permite crear estos ficheros de forma sencilla. Cada fila contiene una clave, el tipo de dato y el valor.

Key	Type	Value
▼ Root	Dictionary	(66 items)
ayuda_trycatch	String	<font size="3" face="Arial"><p>El bloque try - catch sirve
ayuda_trycatch_titulo	String	Try - Catch
ayuda_excepciones	String	<font size="3" face="Arial"><p>Las excepciones son erro
ayuda_excepciones_titulo	String	Excepciones
ayuda_descansar	String	<font size="3" face="Arial"><p>El método descansar hace
ayuda_descansar_titulo	String	Descansar
ayuda_booleano	String	<font size="3" face="Arial"><p>Otro tipo de dato (y, por lo tanto, de variable

Ilustración 62. Archivo plist en Xcode

También podemos modificar el XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>ayuda_trycatch</key>
    <string>&lt;font size=&quot;3&quot; face=&quot;Arial&quot;&gt;&lt;p&gt;El bloque &lt;strong&gt;try -
        catch&lt;/strong&gt; sirve para capturar excepciones.&lt;/p&gt;
    &lt;p&gt;Si una zona de código es propensa a producir una excepción, se escribe en el interior de un
        bloque &lt;strong&gt;try&lt;/strong&gt;.&lt;/p&gt;
    &lt;p&gt;Si la excepción se produce, en vez de terminar, el programa salta al bloque
        &lt;strong&gt;catch&lt;/strong&gt;. En esta zona del código se hace lo que sea necesario para
        corregir la excepción (o nada, si no hace falta).&lt;/p&gt;
       &lt;p&gt; &lt;font color=&quot;#B40404&quot;&gt;&lt;em&gt;try {&lt;br&gt;
        \t /*Zona propensa a errores*/
        &lt;br&gt;}&lt;br&gt;
        catch ( Exception e ) {&lt;br&gt;
        \t /*Código a ejecutar si se produce una excepción.*/&lt;br&gt;}&lt;br&gt;
      &lt;/em&gt;&lt;/font&gt;&lt;/p&gt;&lt;/font&gt;</string>
    <key>ayuda_trycatch_titulo</key>
    <string>Try - Catch</string>
    <key>ayuda_excepciones</key>
```

Ilustración 63. Archivo plist en XML

7.1.3.2 Lectura textos

Se va a mostrar cómo se realiza la lectura del archivo *plist* y se carga el contenido en el *UIWebView*.

Primero se obtiene un diccionario que contiene las claves y valores del archivo plist.

```
if let path = NSBundle.mainBundle().pathForResource("Strings", ofType: "plist"), dict =
    NSDictionary(contentsOfFile: path){
```

Luego obtenemos el texto del diccionario.

```
//Obtenemos texto y titulo de plist
let text = dict.valueForKey(ayudas[pagina]) as! String
let tit = dict.valueForKey(titulos[pagina]) as! String
```

Finalmente se carga en el *UIWebVIew* el código HTML. Además, le indicamos la ruta de la aplicación para poder cargar las imágenes en el código html que sean necesarias.

```
//ruta para obtener imagenes
let path = NSBundle.mainBundle().bundlePath
let baseUrl  = NSURL.fileURLWithPath("\(path)")

// lo copiamos en webview
texto.loadHTMLString(text,baseURL:baseUrl)
self.titulo.text = tit
```

Como resultado, mostramos la vista “Apuntes”, en la que se muestran todas las ayudas desbloqueadas por el usuario.

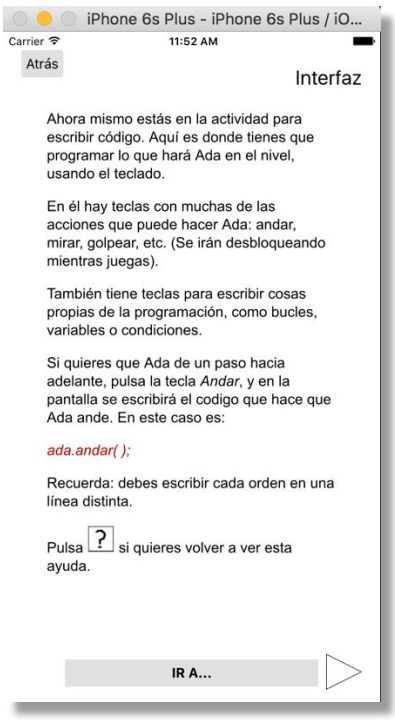


Ilustración 64. Vista de la clase Apuntes

7.1.4 Alerta

Cuando se desee salir del nivel se mostrara una alerta para que el usuario confirme que desee salir, ya que perderá todo el progreso realizado. Para conseguirlo utilizaremos un UIAlertController.

```
//Funcion para salir al mapa, se muestra alerta
@IBAction func SalirButton(sender: UIButton) {

    // Alerta para confirmar que queremos salir
    let refreshAlert = UIAlertController(title: "Salir al mapa", message: "Salir ahora hará que
        pierdas todo tu progreso. ¿Estás seguro?", preferredStyle: UIAlertControllerStyle.Alert)

    // Si damos a OK se pasa a la vista principal
    refreshAlert.addAction(UIAlertAction(title: "OK", style: .Default, handler: { (action:
        UIAlertAction!) in
        // Pasar a vista principal
        let story : UIStoryboard = UIStoryboard(name: "Main", bundle: nil)

        let miVistaDos = story.instantiateViewControllerWithIdentifier("mapa") as!
            MapaViewController
        self.presentViewController(miVistaDos, animated:false, completion:nil)

    }))
    // Si pulsamos cancelar no se realizará nada
    refreshAlert.addAction(UIAlertAction(title: "Cancelar", style: .Default, handler: { (action:
        UIAlertAction!) in
        //no hacemos nada
    }))

    //Se muestra alerta
    presentViewController(refreshAlert, animated: true, completion: nil)
}
```

Ilustración 65. Código alerta salir de nivel

Se obtiene el siguiente resultado al intentar salir.

Ilustración 66. Alerta salir de nivel

7.2 Niveles

En esta sección se va a detallar cómo se han implementado los niveles y la información necesaria para mostrar el menú principal.

7.2.1 JSON

Para almacenar esta información se han usado ficheros JSON. Es un formato que nació como alternativa a XML y ha conseguido un gran número de seguidores.

Está compuesto por dos estructuras:

- Una colección de pares de nombre/valor.
- Una lista ordenada de valores.

Un ***objeto*** es un conjunto desordenado de pares nombre/valor. Un objeto comienza con "**{**"*y* termina con "**}**". Cada nombre es seguido por "**:**" y los pares nombre/valor están separados por "**,**".

Un ***array*** es una colección de valores. Comienza con "**[**"y termina con "**]**". Los valores se separan por "**,**".

A continuación se muestra el archivo JSON que muestra el mapa principal de la aplicación y en el que podemos encontrar los elementos descritos.

```
{
    "ciudad_este": "bosque",
    "exp_este": "30",

    "nombre": "Pueblo inicio",

    "miniatura": "pueblo_inicio",

    "niveles": [
        {
            "nombre": "Tutorial",
            "id": "tutorial",
            "experiencia": "0",
            "descripcion": "Aprende a jugar.",
            "lenguajes": ["Java","C"]
        },
        {
            "nombre": "Interludio I",
            "id": "interludio1",
            "experiencia": "5",
            "descripcion": "Aprende cómo funciona Java.",
            "lenguajes": ["Java"]
        },
        {
            "nombre": "Herrería",
            "id": "herreria",
            "experiencia": "10",
            "descripcion": "Ayuda al herrero a fabricar una espada. \nAprende qué puede hacer Ada.",
            "lenguajes": ["Java"]
        },
        {
            "nombre": "Terremoto",
            "id": "terremoto",
            "experiencia": "20",
            "descripcion": "¡Ha habido un terremoto! Visita el pueblo y comprueba que todo el mundo está
                bien.\nAprende a usar las estructuras de control.",
            "lenguajes": ["Java"]
        },
        {
            "nombre": "Interludio II",
            "id": "interludio2",
            "experiencia": "25",
            "descripcion": "Aprende las variables.",
            "lenguajes": ["Java"]
        }
    ]

}
```


Ilustración 67. pueblo_inicio.json

7.2.2 Estructura mapa

Como se puede ver, el archivo mostrado tiene una estructura que se va a explicar. Está compuesto por los siguientes campos:

- **"ciudad_este":** indica el nombre del archivo json que es necesario cargar cuando se desee mostrar ese mapa
- **"exp_este":** indica el valor mínimo de experiencia que debe tener el usuario para desbloquear el mapa que está al este.
- **"nombre":** indica el nombre del mapa que se utilizará para mostrar el título en la interfaz.
- **"miniatura":** indica el nombre de la imagen con formato ".png" que se utilizará para mostrar una imagen en la interfaz.
- **"niveles":** indica los niveles que existen en ese mapa, por cada nivel aparecerá un botón en el mapa. La información que contiene es:
    - **"nombre":** texto que aparecerá en el botón.
    - **"id":** nombre del archivo json que se cargara para desarrollar el nivel.
    - **"experiencia":** indica la experiencia necesaria que debe disponer el usuario para desbloquear el nivel.
    - **"descripcion":** texto que se mostrará al pulsar el botón ofreciendo una breve descripción del nivel.
    - **"lenguajes":** indica los lenguajes para los que está disponible el nivel, Java y/o C.

7.2.3 Estructura nivel

La estructura de los niveles es más compleja y se va a dividir en dos, una para los diálogos, y otra para el juego.

7.2.3.1 Diálogo

El diálogo se muestra al iniciar el nivel, se va a explicar un fragmento de un nivel.

```
[    {
        "dialogo": "true",
        "fondo": "fondo_bosque",
        "mensajes": [
            {
                "imagen": "boole",
                "posicion": "derecha",
                "texto": "Me llamo Boole. Vengo de la capital, muy lejos en el norte."
            },
            {
                "imagen": "0",
                "posicion": "derecha",
                "texto": "Hace un par de semanas, el código de nuestra tierra se rompió."
            },
            {
                "imagen": "0",
                "posicion": "derecha",
                "texto": "Es algo que ocurre cada cientos de años. Hay un error en el código de nuestro
                    mundo, un fallo en el sistema que provocar una reacción muy violenta en el mundo que
                    vemos."
            },
```

Ilustración 68. Diálogo en JSON

En esta imagen se observan algunos campos como:

- **"dialogo":** indica si el nivel es solo de diálogo o no.
- **"fondo":** indica el nombre de la imagen con formato ".png" que se cargará como fondo de la conversación en la interfaz.
- **"mensajes":** contiene varios elementos que indican los distintos mensajes de los personajes. Cada mensaje contiene los siguientes elementos:
    - **"imagen":** indica el nombre del personaje que habla. Cuando el valor es cero indica que no cambia la imagen que aparece en esa posición.
    - **"posicion":** indica la posición en la que se encuentra el personaje que habla.
    - **"texto":** contiene el mensaje del personaje.

7.2.3.2 Juego

En este apartado se va a comentar los campos que aparecen en los archivos JSON para el desarrollo del juego.

```
"teclas": [
    {
        "teclado": "1",
        "teclas": ["12"]
    }
],
```

```
"avatar": "monje_mini",
"resumen": "¿Eres capaz de escribir unas líneas de código que lleven a Ada a la meta?",
"objetivo": "Objetivo: Llega a la baldosa de color rojo.",
"secundario": "",
"victoria": "meta",
"lineas": [ ["401" , "303"], ["401"], ["0"] ],
"obligatorio": [ ["401"] ],
"posicion_inicial": "1",
"mapa": {
    "filas": "2",
    "columnas": "3",
    "baldosaBase": "baldosa_hierba",
    "item": [
        {
            "posicion": "3",
            "direccion": "abajo",
            "tipo": "meta",
            "bitmap": "sprite_meta"
        },
        {
            "posicion": "4",
            "direccion": "derecha",
            "tipo": "ada",
            "bitmap": "sprite_ada"
        }
    ]
},
"ayuda": [ "interfaz", "teclado" ]
```

Ilustración 69. JSON de un nivel

En este nivel podemos observar algunos campos como:

- **"teclas":** indica las teclas que se desbloquean.
- **"avatar":** indica la imagen con formato ".png" que aparece como avatar en la vista Resumen.
- **"resumen":** contiene el texto que aparecerá en el campo resumen de la vista Resumen.
- **"objetivo":** contiene el texto que aparecerá en el campo objetivo de la vista Resumen
- **"victoria":** indica la forma en la que es posible resolver el nivel. "meta" indica que hay que poner el personaje sobre la baldosa que indique la meta.
- **"lineas":** indica las líneas que se escribirán automáticamente en la vista en la que procedemos a escribir el código.
- **"obligatorio":** indica las líneas obligatorias que deben aparecer en el código para resolver el nivel. Si no aparecen no se puede resolver el nivel.
- **"posicion_inicial":** indica la línea que se resaltara al principio de escribir el código.
- **"mapa":** describe el tablero de juego. Se especifican las filas y columnas, las baldosas que lo componen y los "items". Estos últimos indican los elementos que habrá sobre el tablero, indicando la posición, dirección, tipo y la imagen que representa a cada uno.
- **"ayuda":** indican las ayudas que se muestran en el nivel para poder resolverlo.
- **"ayuda_extra":** indica las ayudas que se desbloquearán al resolver el nivel y aparecerán sólo en el libro de apuntes.

7.3 Código

Esta clase es la base que permite resolver el nivel. Este objeto representa a una línea de código escrita por el usuario.

Utiliza una serie de constantes para identificar los elementos que se pueden escribir en una línea, ya sean variables o estructuras de código como bucles. Estos enteros son:

- Del número 301 a 400 se utiliza para constantes dentro del código (arriba, derecha, enemigo etc.).

Esto no quiere decir que haya cien constantes y se utilicen todos los números disponibles; se ha dividido así para hacer más intuitiva la separación y añadir más elementos en caso de que sea necesario.

- Del 401 al 500 están los métodos que puede usar Ada como “andar”, “mirar”, “escuchar”, etc.
- Del 501 al 550 están las estructuras de control como “if”, “while”, “switch”, etc.
- Del 601 al 700 se asignan los comparadores como “igual”, “mayor que”, “AND”, etc.
- A partir del 1000, se asignan variables.

Un objeto contiene los siguientes atributos para identificar lo que se ha escrito en una línea.

```
var codigo:Int
var params:[Int]
var nTab:Int
var cadena:[String]
var lenguaje:String
```

Ilustración 70. Atributos de Codigo

- **“codigo"**: guarda un entero con el elemento principal. Es el que se usa para identificar qué se hace en esa línea (se inicia un bucle, se realiza una acción, etc).
- **“params”:** es una lista de todo lo que se ha escrito en la línea, en orden.
- **“cadena”:** es una lista auxiliar de “params”, ya que algunos elementos de “params” necesitan una cadena auxiliar. Por ejemplo, una variable se identifica en “params” con 1001, pero necesita un nombre que se guarda en “cadena”.
- **“nTab”:** guarda el número de tabulaciones. Se utilizará a la hora de obtener la cadena que debe aparecer en la interfaz, ya que dentro de los bucles el código aparecerá tabulado.
- **“lenguaje”:** es el lenguaje en el que se está jugando.

La clase ofrece métodos para establecer y recuperar estos parámetros como *setCadena*, *setTab*, *addParam* o *getCodigo*, *getCadena*.

El método más destacado es “escribirLinea”. Este método convierte en caracteres, con la sintaxis correcta, la línea escrita. Lo hace en tres pasos:

1) Añade las tabulaciones necesarias.
2) Recorre la lista de parámetros. Convierte cada elemento a una cadena de caracteres y añade algún carácter más si es necesario, como un cierre de paréntesis si es un método.
3) Termina de escribir la línea, normalmente añadiendo un punto y coma o la apertura de unas llaves.

Las líneas escritas se mostrarán en la tabla definida en la clase *CrearCodigoViewController*, reflejando lo que lleva escrito el jugador.

El código que el jugador tiene que crear será una lista de objetos de esta clase, que después se ejecutará. Se escribirá el código en la clase *CrearCodigoViewController* y se ejecuta en *ResolverViewController*.

7.3.1 Crear código

Como se ha comentado, para escribir el código se utilizara la clase clase *CrearCodigoViewController*. Algunos de los atributos que utiliza son:

```
//Cada elemento es una línea de código
var codigo:Array<Codigo>!
  // Variables y constantes creadas. Tipos tambien
var nomVariables = [String]()
var constantes = [String]()
var tipos = [String]()
var numTipos:Int!
var numConstantes: Int!
//Linea seleccionada
var linea_sel:Int = 0
var lenguaje:String!
```

Ilustración 71. Atributos CrearCodigo

Estos atributos identifican las líneas de código, los nombres de las variables creadas, las constantes y tipos desbloqueados, la línea seleccionada para editarla y el lenguaje de codificación.

Para escribir el código se utiliza el teclado personalizado. Existen tres teclados principales, y uno que se utiliza para escribir el nombre de las variables. El primer teclado contiene los métodos que puede realizar el personaje, el segundo contiene las estructuras de control, y el tercero contiene los comparadores y los números.

Existen teclas especiales como la tecla “Enter”, que aparece en los teclados principales, o “Borrar”, que aparece en todos. A continuación se va a mostrar el detalle del código que se ejecuta al pulsar “Enter”.

```
//Enter
func enter() {
    let c = codigo[linea_sel]
    //Se aumenta una linea
    linea_sel = linea_sel + 1

    codigo.insert(Codigo(lenguaje: lenguaje, c: 0), atIndex: linea_sel)

    //Si la línea anterior era el inicio de un bucle, se aumenta la tabulación de la siguiente
    if (c.getCodigo() >= Codigo.IF && c.getCodigo() <= Codigo.FIN_BUCLE_INICIO){
        codigo[linea_sel].setTab(c.getTab() + 1)
    } else {
        codigo[linea_sel].setTab(c.getTab())
    }

    let texto = ["texto" : codigo[linea_sel].escribirLinea()]
    array_instrucciones.insertObject(texto, atIndex: linea_sel)

    //recargamos tabla
    tabla_codigo.reloadData()

    //Mantenemos seleccion
    let index = NSIndexPath(forRow: linea_sel, inSection: 0)
    tabla_codigo.selectRowAtIndexPath(index, animated: true, scrollPosition: UITableViewScrollPosition.Middle)

    //Se ajusta scroll de tabla
    autoscroll()
}
```

Ilustración 72. Código de Enter

Primero se obtiene el código de la línea actual y se aumenta la línea. Luego se inserta un código vacío en la lista de códigos y se aumenta la tabulación. Finalmente, se actualiza la tabla en la que aparece el código para que aparezca la nueva línea y se mantenga seleccionada.

Las teclas están agrupadas por bloques. Cada uno realiza una acción similar, como las estructuras de control.

```
//Iniciar un bucle: if, else if, else, while.
func bucle(tag:Int) {

    var c = codigo[linea_sel]
    if (c.getCodigo() == Codigo.CIERRE) {
        // Añadimos una linea
        let texto = ["texto" : ""]
        array_instrucciones.insertObject(texto, atIndex: linea_sel+1)

        // Insertamos codigo
        codigo.insert(Codigo(lenguaje:lenguaje,c: 0), atIndex: linea_sel+1)
        // Aumentamos linea seleccionada
        linea_sel = linea_sel + 1

        c = codigo[linea_sel]
        c.setTab(codigo[linea_sel - 1].getTab())
    }
    c.setCodigo(tag)

    //Y añado el cierre de la llave, con un parámetro extra para saber qué cierra.
    codigo.insert(Codigo(lenguaje:lenguaje,c: Codigo.CIERRE), atIndex: linea_sel+1)
    codigo[linea_sel + 1].addParam(tag + 50)
    codigo[linea_sel + 1].setTab(c.getTab())

    // Actualizamos linea de la tabla
    var texto = ["texto" : codigo[linea_sel].escribirLinea()]
    array_instrucciones.replaceObjectAtIndex(linea_sel, withObject: texto)
    texto = ["texto" : codigo[linea_sel + 1].escribirLinea()]
    array_instrucciones.insertObject(texto, atIndex: linea_sel + 1)
```

Ilustración 73. Código crear estructuras de control

- Primero se comprueba si la línea actual es un cierre de otro bucle. Si es así, se añadirá una línea en blanco debajo para comenzar ahí el código de la tecla que se ha pulsado.
- Esto es debido la forma en la que se ha desarrollado el proyecto, los cierres de llaves y los inicios de nuevos bucles han de estar en líneas separadas. Por ejemplo, no se puede escribir "} else {" en una misma línea. Como esto puede resultar confuso para los jugadores, que tendrán la intención de iniciar el siguiente elemento del bloque en el cierre, se ha tenido en cuenta y se realiza automáticamente.
- Luego se añade el cierre de la llave con un parámetro extra para identificar el bucle que cierra posteriormente.

Para escribir una acción se realiza de forma parecida, pero con detalles particulares.

```
//Acciones: andar, mirar, hablar, golpear
func acciones(tag:Int) {
    let c = codigo[linea_sel]

    if (c.getCodigo() == 0){
        c.setCodigo(tag)

        // Actualizamos linea de la tabla
        let texto = ["texto" : codigo[linea_sel].escribirLinea()]
        array_instrucciones.replaceObjectAtIndex(linea_sel, withObject: texto)
    }  else if (c.getCodigo() < Codigo.FIN_ACCIONES) {
        // Se añade una linea
        linea_sel = linea_sel + 1

        codigo.insert(Codigo(lenguaje:lenguaje,c: tag), atIndex: linea_sel)
        codigo[linea_sel].setTab(c.getTab())

        // Actualizamos linea de la tabla
        let texto = ["texto" : codigo[linea_sel].escribirLinea()]
        // Se inserta linea
        array_instrucciones.insertObject(texto, atIndex: linea_sel)
    } else if (c.getCodigo() != Codigo.CIERRE || c.getParam(1) == Codigo.DO + 50) {
        c.addParam(tag)

        // Actualizamos linea de la tabla
        let texto = ["texto" : codigo[linea_sel].escribirLinea()]
        array_instrucciones.replaceObjectAtIndex(linea_sel, withObject: texto)
    } else {
        Toast("¡No puedes escribir ahí!")
    }
```

Ilustración 74. Código crear acción

- Si la línea estaba vacía, se añade el código de la tecla como el código principal de la línea.
- Si en la línea existe otra acción, se añade el código en una línea nueva. Así, no es necesario que el usuario pulse la tecla "Enter" para escribir una acción nueva, ya que en una línea solo puede haber una acción.

- En otros casos, como dentro de una condición, se añade como un parámetro al código existente.

Para añadir constantes o variables, el teclado mostrara una lista en la que el usuario seleccionara el elemento que desee. Por ejemplo, la tecla "CTE" muestra una lista con las constantes desbloqueadas.

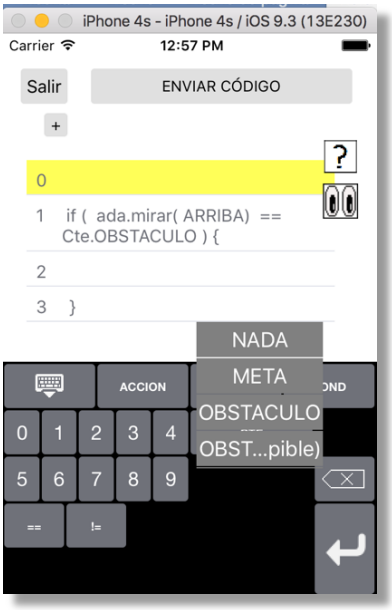


Ilustración 75. Lista constantes desbloqueadas

Al pulsar una de estas teclas se llamará a la función "teclaCte" que se encargará de que se introduzca la constante en el código. Las constantes estarán identificadas por un número que variará entre cero y el máximo de constantes desbloqueadas.

```
// mostrar vista
varView.hidden = false

var button : UIButton
// El ancho sera igual a la vista
let width = scrollView.frame.width
// Varibales que determinan la posicion en la vista
let x :CGFloat = 0.0
var y :CGFloat = 0.0

// Si hay alguna constante
if numConstantes != 0 {

    for posicion in 0 ..< numConstantes {
        button = UIButton()
        button.frame = CGRectMake(x, y, width, 30.0)
        button.backgroundColor = UIColor.grayColor()
        button.setTitle(constantes[posicion], forState: UIControlState.Normal)
        button.tag = posicion
        button.addTarget(self, action: #selector(Keyboard.pulsarCte(_:)), forControlEvents:
            UIControlEvents.TouchUpInside)
        // añadimos boton a la vista
        scrollView.addSubview(button)

        // Añadimos botones al array para borrarlos despues
        botonesCte.append(button)
        // Incrementamos y para posicionar el boton debajo
        y = y + 31
    }
}
```

Ilustración 76. Código creación de lista

Esta lista también dispone de un botón "Salir", que oculta la lista.

Cuando se quiere crear una variable, ocurre algo parecido. Se muestra la lista en la que aparecen las variables creadas, pero, además, nos permite crear una variable nueva. Si se decide crear una, se mostrará otra lista en la que se debe seleccionar el tipo de dato de la variable.

Tras seleccionarlo aparecerá el cuarto teclado, que es muy similar a uno normal, ya que contiene solo letras y números para poder escribir el nombre de la variable.

Ilustración 77. Escribir variable

Cuando se vayan pulsando las letras se ira guardando en una cadena el nombre. La tecla que aparece en la parte inferior izquierda permite escribir en mayúsculas, cuando este pulsada se verá con un toque azulado.

```
} else {
    //Se añade una letra
    s.append(letra)
    c.setCadena(s)
}
```

Ilustración 78. Añadir letra al nombre de una variable

La tecla “Terminado” finaliza la creación de la variable, incluyéndola como seleccionable en la lista. Tras esto, se cargará teclado principal y se añadirá una línea en blanco debajo.

```
    //Acabar
} else if (tag == 257) {

    //Se comprueba si está vacío
    if(s == ""){
        bien = false
    }

    if(bien) {
        nomVariables.append(s)

        //Y se hace enter
        linea_sel = linea_sel + 1
        codigo.insert(Codigo(lenguaje: lenguaje, c: 0), atIndex: linea_sel)
        codigo[linea_sel].setTab(c.getTab())

    }else {
        Toast("Nombre de variable inválido")
        codigo[linea_sel].setCodigo(0)
    }

    if let custView = NSBundle.mainBundle().loadNibNamed("Keyboard1", owner: nil, options: nil).first as? Keyboard
        {
        customView.removeFromSuperview()
        //Ajustamos alto y ancho para que llenen view
        customView = custView
        customView.frame = CGRectMake(0,0,tecladoView.frame.width,tecladoView.frame.height)
        tecladoView.addSubview(customView)
```

Ilustración 79. Crear variable

Cuando se vaya a resolver el código, se le pasar la lista con los códigos escritos.

```
@IBAction func irAResolver(sender: UIButton) {

    //pasamos a vista historia
    let story : UIStoryboard = UIStoryboard(name: "Main", bundle: nil)
    let miVistaDos = story.instantiateViewControllerWithIdentifier("resolver") as! ResolverViewController

    miVistaDos.nivel = nivel
    miVistaDos.etapa = etapa
    miVistaDos.codigo = codigo
    miVistaDos.linea_sel = linea_sel
    miVistaDos.array_instrucciones = array_instrucciones
    self.presentViewController(miVistaDos, animated:false, completion:nil)
}
```

Ilustración 80. Pasar a resolver código

7.4 Lectura JSON

Como se ha comentado, se dispone de ficheros JSON de los que se va a obtener información. Para leer los datos se va a utilizar la librería de código abierto **SwiftyJSON**.

Primero obtenemos la ruta del archivo y luego creamos un objeto utilizando la librería.

```
let path = NSBundle.mainBundle().pathForResource("/NivelesTFG/"+nombre, ofType: "json")
let jsonData = NSData(contentsOfFile:path!)
let jsonArray = JSON(data: jsonData!)
```

Para obtener un valor se accede como en un diccionario seguido del tipo de dato que queramos obtener, como string o int. A continuación se muestra como obtenemos el nombre del fondo del diálogo.

```
return self.json!["fondo"].string!
```

Se utilizará la clase **ParserJSON** cada vez que sea necesario obtener un valor de un fichero JSON. Esta clase contendrá métodos obtener los parámetros. Algunos de ellos son: *getFondo*, *getMensaje*, *getTeclas* o *getLineasCodigo*.

7.5 Tablero y Sprites

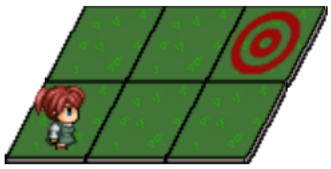


Ilustración 81. Tablero

El juego se desarrollará en un tablero compuesto por baldosas y sprites. Las baldosas componen el suelo, no son interactuables y únicamente determinan por dónde se pueden mover los personajes. Los sprites son los diferentes ítems que puede haber en el tablero (*Ada*, los enemigos, objetos, etc).

Para dibujarlo dispondremos de la clase *Tablero* y la clase *Sprite*.

7.5.1 Sprite

Cada uno de los elementos que se muestren en el mapa, con excepción de las baldosas, serán recogidos en un objeto de la clase *Sprite*.

Esta clase tiene como función controlar el movimiento, la interactuación con los demás elementos del mapa y la animación de los diferentes items. Pueden tener diferentes estructuras, según el tipo de *Sprite*:

Ilustración 82. Imágenes Sprites

- **Simples**: no tienen movimiento, el sprite está compuesto solo por una imagen.
- **Animación estática**: el sprite está quieto pero muestra una animación formada por tres imágenes. Únicamente tienen la animación de reposo y siempre mirarán en un único sentido.
- **Animación dinámica**: corresponde a los personajes que se mueven por el tablero. Las animaciones dependerán de la acción que estén realizando (reposo, andar o golpear), y de la dirección. La combinación de estas estará formada por tres imágenes.

En la clase *Sprite* se han definido diferentes tipos de elementos que puede haber en el tablero. Según el tipo cambiará el comportamiento.

```
//Tipo de Sprite. Modela su comportamiento
static let ROCA = 0           //Se queda quieto. No ataca ni puede pisarse. Puede romperse.
static let DIANA = 2          //Hay que romperlo para ganar (si la victoria es por limpiar). No
    ataca ni se mueve.
static let COCODRILO = 3      //No se mueve. Si lo pisas te ataca.
static let META = 4           //La meta. Si la pisas ganas el juego (si la victoria es llegando
    a la meta).
static let ADA = 5            //El protagonista. Más acciones.
static let BESTIA = 6         //Se mueve y ataca a Ada. Con IA.
static let FIJO = 7           //Se queda quieto y no ataca. Irrompible.
static let OBJETO = 8         //Un objeto. No se puede romper, pero sí coger.
static let NPC = 9            //Personaje amistoso con IA.
static let EXCEPCION = 10     //Excepcion. Si las mira, toca o ataca, muere.
```

Ilustración 83. Tipos de Sprite

La clase *Sprite* se encarga de dibujar correctamente el personaje en el tablero. La posición en el mapa de baldosas está determinada por dos atributos: “fila” y “columna”, contándose desde arriba y desde la izquierda.

A partir de la posición hay que calcular las coordenadas para colocar el elemento en la baldosa que le corresponde.

```
yoriginal = (vista.frame.size.height-(CGFloat(filas)*altoBaldosa))/2
xoriginal = (vista.frame.size.width-(CGFloat(columnas)*ANCHOINF+DESPDER*(CGFloat(filas))))/2
// Se ajusta la posicion horizontal en funcion del numero de filas que haya
let ajustex = DESPDER*(CGFloat(filas)-1)
xoriginal = xoriginal + ajustex

//posicion inicial sprite
xposition = xoriginal
yposition = yoriginal

//Ajustar posicion(mover hacia la derecha y/o hacia abajo)
let ajuste_y = (posicion-1)/columnas
let ajuste_x = posicion - 1 - ajuste_y*columnas

//Desplazamos arriba y/o abajo
xposition = xposition - (DESPDER-CGFloat(anchoBaldosa)) * CGFloat(ajuste_x) - DESPDER * CGFloat(ajuste_y)
yposition = yposition + CGFloat(altoBaldosa) * CGFloat(ajuste_y)
```

Ilustración 84. Cálculo posición Sprite

Primero calculamos la posición de partida del tablero, luego se calcula el número de desplazamientos horizontales y verticales en función de la posición. Finalmente, se ajusta la posición multiplicando el desplazamiento horizontal por el ancho de la baldosa, el desplazamiento vertical por el alto de la baldosa y los ajustes en la posición horizontal debido a la inclinación del tablero.

Una vez calculada la posición, se crea un rectángulo que delimitará el área de la imagen.

```
sprite = UIImageView(image: UIImage(named: imagen+postura+nomDir+"1"))
sprite.frame = CGRectMake(xposition, yposition, sprite.frame.size.width, sprite.frame.size.height)
```

Ahora que se tiene su posición, hay que controlar al personaje. Para cambiar la acción hay que llamar al método "setAccion":

```
func setAccion(acc:Int, dir:Int) {
    //Si es una acción nueva, cargo el nuevo bitmap.
    if (accion != acc) {
        accion = acc
    }
    direccion = dir

    //Se configuran las opciones determinadas de cada acción.
    switch (acc) {
    case Sprite.ATACAR: break

    //Va para atrás lo que lleva andando.
    case Sprite.CHOCAR: break

    case Sprite.ANDAR:
        switch (dir) {
        case Sprite.DERECHA:
            if (columna  > tablero.getColumnas()) {
                andar("derecha", sentido: "derecha", chocar: false, quieto: true)
                accion = Sprite.QUIETO
            }
```

Ilustración 85. Método setAccion de Sprite

Primero fija la acción (un entero de cero a dos), la dirección (un entero de cero a tres, para identificar la fila), y se configuran las opciones necesarias para cada acción.

Los sprites que se animan lo hacen continuamente cambiando entre tres imágenes que dan al elemento la sensación de movimiento. Para ello se utilizan métodos disponibles en la clase *UIImageView*.

```
//animacion
let animationImages:[UIImage] = [UIImage(named: imagen+postura+nomDir+"1")!, UIImage(named: imagen+postura+
    nomDir+"2")!,UIImage(named: imagen+postura+nomDir+"3")!]

//comienzo animacion
sprite.animationImages = animationImages
sprite.animationDuration = 0.4
sprite.animationRepeatCount = 0
sprite.startAnimating()
```

Ilustración 86. Animación Sprite en estático

Primero configuramos las imágenes que van a formar parte de la animación, se ajusta la duración de cada imagen y el número de veces que se va a repetir la animación. Un cero indica que la animación es infinita.

Al actualizar, si el personaje está realizando alguna acción, se hacen cambios adicionales como moverse sobre el tablero o golpear un objeto. Se va a utilizar la clase *CABasicAnimation* para mover a los personajes.

```
let anim = CABasicAnimation(keyPath: "position")

if(!quieto) {
    //Para controlar fin de animacion
    CATransaction.begin()

    //Creamos animacion
    anim.fromValue = NSValue(CGPoint: point)
```

Ilustración 87. Moviemiento Sprite I

Se crea la animación y se fija el punto de partida. Luego se actualiza la posición y se fija el destino, cuando finalice la animación se quedará en esa posición.

```
point = CGPoint(x: point.x - DESPDER, y: point.y + CGFloat(altoBaldosa))
```

```
// Se mueve a este punto
anim.toValue = NSValue(CGPoint: point)

//No se repite
anim.repeatCount = 0
```

Ilustración 88. Movimiento Sprite II

7.5.2 Tablero

Tablero gestiona el movimiento de todos los elementos que hay en juego. Tiene tres funciones principales: dibujar los elementos en juego, controlarlos (su movimiento, las interacciones entre ellos...) y actuar como interfaz para obtener datos del tablero.

Hay dos tipos de tableros diferentes: el de muestra (que se dibuja en *ResumenViewController*) y el definitivo (utilizado en *ResolverViewController*). En el tablero de muestra se dibujarán de forma semitransparente los sprites que tengan una posición aleatoria.

Para ello, utiliza cuatro atributos principales:

```
//Sprite de ada(protagonista)
var ada:Sprite!
// Array de sprites
var items = [Sprite]()
// nombre baldosa base del tablero
var baldosaBase: String!
var baldosas = NSMutableArray()
```

Ilustración 89. Atributos principales del tablero

Para dibujar el suelo utiliza “baldosas”, que es una lista con el el nombre dela imagen de cada baldosa del tablero. Como en la mayoría de los casos casi todas las baldosas son del mismo estilo, la baldosa principal se guarda en “baldosaBase”.

El *ArrayList* “items" contiene a todos los sprites del tablero, excepto *Ada*, que está aparte.

7.5.2.1 Creación

Como se vio anteriormente, el archivo JSON de un nivel contiene los datos necesarios para crear el tablero.

```
"mapa": {
    "filas": "2",
    "columnas": "3",
    "baldosaBase": "baldosa_hierba",
    "item": [
        {
            "posicion": "3",
            "direccion": "abajo",
            "tipo": "meta",
            "bitmap": "sprite_meta"
        },
        {
            "posicion": "4",
            "direccion": "derecha",
            "tipo": "ada",
            "bitmap": "sprite_ada"
        }
    ]
```

Ilustración 90. Ejemplo tablero JSON

El constructor de Tablero coge los diferentes elementos y los crea:

- Obtiene los datos del tablero (fila, columna, baldosaBase).

- Crea el una lista de "baldosas" con tantos elementos como baldosas haya, y en aquella posición en que la baldosa del suelo sea distinta a la base, se pone la imagen correspondiente
- Se crean los *Sprites*, obteniendo los datos de la lista "item".
- Si es un tablero de muestra, se hacen semi transparentes los elementos aleatorios y se dibuja un símbolo en las demás posiciones posibles.

7.5.2.2 Actualizar

El tablero se tiene que actualizar cada vez que se ejecute una línea y mostrar los cambios debidos al código escrito. Se hace en una serie de pasos:

1) Se comprueba si *Ada* choca con algún ítem. En caso de que sea cierto, disminuye la salud de *Ada* si se hace se hace daño (al pisar agua o un cocodrilo, por ejemplo) o cambia la acción a "chocar" si no puede pasar por encima de ese ítem.
2) Si *Ada* está atacando, realiza el ataque y disminuye la salud del objetivo. Si el ataque falla (porque no había nada delante), es Ada quien recibe el daño. Así se penalizan los códigos mal estructurados.
3) Si un enemigo ha iniciado su ataque y está en una posición colindante a la de *Ada*, ésta recibe el daño disminuyendo su salud.
4) Actualiza a *Ada*, utilizando el método de la clase Sprite.
5) Si *Ada* se está moviendo, se actualiza la posición del tablero para que quede centrado en *Ada*. Esto ocurrirá si el tablero es más grande que el recuadro que lo delimita.
6) Se actualizan los ítems.

Los personajes amistosos tienen comportamiento diferente según su naturaleza y el nivel que se esté jugando. Los enemigos suelen tener un único comportamiento: perseguir a *Ada* y atacarle.

7.5.2.3 Otros métodos

La clase *Tablero* también dispone de varios métodos que permiten interactuar al usuario con el juego, como los métodos mirar o escuchar, en los que se obtienen datos del tablero.

Si se desea actuar sobre el personaje, la clase *Tablero* nos permite obtener el *Sprite* de *Ada* y acceder a los métodos para cambiar su comportamiento. Por ejemplo, si el usuario desea que Ada ande, la clase *Resolver* obtiene el *Sprite* a través de *Tablero* y cambia la acción.

Tenemos métodos como "mirar", que devuelve un entero con el código que representa lo que hay en la baldosa. "Escuchar" devuelve un entero que según la situación en la que se utilice significa una cosa. "Coger" permite a Ada obtener equipo y "usar" lo que hace es usar ese equipo.

Estos métodos también suelen indicar un mensaje de log en el que se explica lo que ocurre en cada momento, como indicar lo que se ha escuchado.

7.6 Resolver

La clase *ResolverViewController* será la encargada de mostrar al jugador el resultado del código escrito. Es decir, se mostrara gráficamente las acciones de cada línea escrita. Ya sea mediante texto o cambios en el tablero.

Cada vez que el usuario pulse la pantalla, se irá leyendo una línea y ejecutándola. Esto será así hasta que se gane o se lean todas las líneas. En la pantalla aparecerá un fragmento del código, en concreto, tres líneas apareciendo en el medio la que se va a leer a continuación.

Para poder decidir si se ha ganado, se debe conocer el objetivo del nivel. Cada nivel presentara un reto distinto y esta clase debe detectar cuando se ha cumplido, o cuando no se puede conseguir.

Lo primero que se hace es comprobar que el código sea válido para varias ejecuciones. Para ello se ejecutan diferentes distribuciones del tablero con semillas distintas, estas semillas determinarán el valor de un componente aleatorio. En esta fase el tablero no dibuja nada, sino que el movimiento se realiza instantáneamente, es decir, es una fase transparente al usuario.

Se cambia la semilla hasta que se llegue al final o falle en una ejecución. Si una distribución concreta ha dado error esa es la que se mostrará al jugador, mientras que si ha pasado todas las ejecuciones sin error, se utilizará una semilla aleatoria.

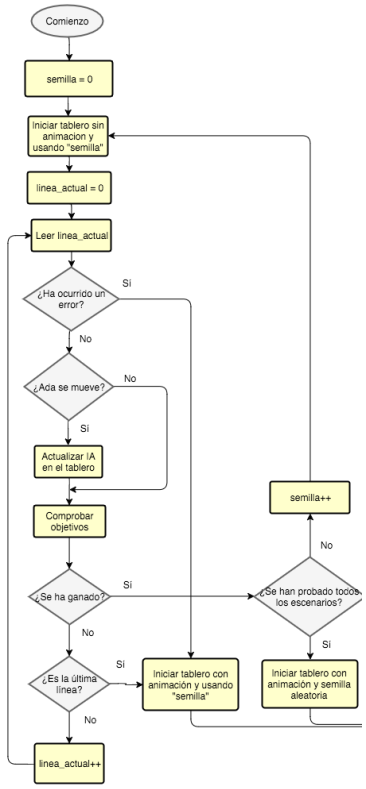


Ilustración 91. Comprobación código válido

Una vez elegida la semilla, se mostrará por pantalla la distribución escogida. En este caso no será una ejecución automática, sino que se leerá una línea cada vez que el jugador pulse la pantalla.

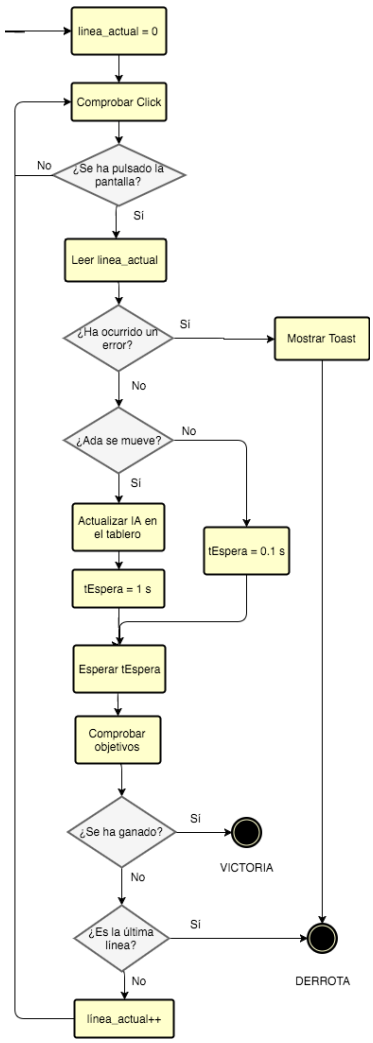


Ilustración 92. Diagrama resolver nivel (II)

Cuando el jugador pulsa la pantalla, se llama a la función "click_Ejec":

```
//Cuando se pulse se llama a funcion ejecutar
// Habilitamos deteccion de toques
self.view.userInteractionEnabled = true
tapRec = UITapGestureRecognizer(target: self, action: #selector(ResolverViewController.click_Ejec))
self.view.addGestureRecognizer(tapRec!)
```

Ilustración 93. Reconocer toque en Resolver

Esta función llamara al método "ejecutar" sólo si las variables booleanas "enEjecucion" y "fin" son "false".

- "fin" es true cuando ya se han leído todas las líneas o el jugador ha ganado o perdido el nivel.
- "enEjecucion" es verdadera mientras se está ejecutando la línea. Se le asigna "true" cuando comienza a leerse la línea de código, y no vuelve a estar "false" hasta que termina la animación del tablero.

Así se consigue que no se solapen varias ejecuciones, y no se ejecuten más líneas cuando se ha terminado la partida.

```
func click_Ejec(){
    if (!enEjecucion && !fin){
        ejecutar()
    }
}
```

Ilustración 94. Control de ejecución

Debido a que la lectura del código es independiente de la ejecución del tablero, es necesario asegurar que las dos tarden el mismo tiempo. Por ello, la función "ejecutar" espera un tiempo después de leer la línea de código, tras este tiempo se pone la variable "enEjecucion" como "false".

```
if (esperar) {
    tEspera = 1.0
    if(codigoRes[linea_actual].getCodigo() == Codigo.DESCANSAR){
        tEspera = 0.1
    }else if (codigoRes[linea_actual].getCodigo() == Codigo.GOLPEAR){
        tEspera = 0.6
    } else if (chocado){
        tEspera = 0.5
        chocado = false
    }

} else {
    tEspera = 0.1
}

///delay
let time = dispatch_time(DISPATCH_TIME_NOW, Int64(tEspera * Double(NSEC_PER_SEC)))
dispatch_after(time, dispatch_get_main_queue()) {
```

Ilustración 95. Retraso en ejecutar

- "esperar" es una variable booleana que es "true" cuando en la línea de código se ordena una acción que necesite tiempo, como un movimiento si se quiere "andar". Según el tipo de animación se esperará más o menos. Si no hay animación se esperará 0.1 segundos.

Tras el tiempo de retraso por la animación, se cambian las variables "enEjecucion" y "esperar" y se comprueba si el jugador ha ganado la partida. Además, se actualiza el movimiento de los demás personajes no controlados por el jugador, estos se mueven en el mismo turno que Ada. Las líneas que contengan estructuras de control no provocarán que los enemigos realicen acciones.

7.6.1 Leer líneas

La función "ejecutar" primero llamará a la función "leerLinea". Cada línea tiene un código principal, que es el que determine si se realiza una acción u otra. Si en el código se ha escrito que Ada ande, se realiza lo siguiente:

```
//Andar
if (c.getCodigo() == Codigo.ANDAR) {
    esperar = true
    //Si tiene un parámetro y es una dirección
    if (c.getNumParam() == 2 && (c.getParam(1) >= Codigo.ABAJO && c.getParam(1) <= Codigo.ARRIBA)) {
        tablero.getAda().setAccion(Sprite.ANDAR,dir: c.getParam(1) - Codigo.ABAJO)
        //O si no tiene parámetros
    } else if (c.getNumParam() == 1) {
        tablero.getAda().setAccion(Sprite.ANDAR)
    } else {
        if (!ejecucion_prev){
            Toast("Error: Parámetros incorrectos en Andar")
        }
        fin = true
    }
```

Ilustración 96. Fragmento leer línea (Código andar)

"c" es un objeto "Codigo" que contiene la línea actual. Se leen los parámetros que contiene, para comprobar si se ha escrito correctamente. Si hay algún fallo, se lanza un *Toast* que notifica del error al jugador.

Al leer este código, podemos establecer la acción que va a realizar Ada a través de su Sprite, que nos lo devuelve Tablero a través del método "getAda".

Otras líneas de código son más complicadas, como los condicionales (if, else if o else). Para ejecutar estos se realizan una serie de pasos:

1) Si es un else if o un else, se comprueba que la línea anterior sea el cierre de un if o de un else if.
2) Se evalúa la condición, comparando poco a poco los parámetros de la línea. Si hay un fallo de sintaxis, se termina la partida. Si no:
    i) Si la condición es correcta, se pasa a la siguiente línea.
    ii) Si es incorrecta, se salta al cierre del bloque y se avanza una línea más.

3) Si la línea contiene un cierre de llaves, significa que se ha recorrido el bucle y hay que saltarse los else if y else siguientes.

7.6.1.1 Evaluar condición

La evaluación de una condición conlleva utilizar una serie de variables. Está implementada en el método "esCorrecta", que recibe como parámetro el número del parámetro de inicio en que empieza la condición. Por ejemplo si es un if, hay que empezar a mirar en el segundo parámetro, puesto que el primero sería el código del if.

```
//Comprueba si una condición es cierta o no.
func esCorrecta(paramInicio:Int) -> Bool {

    var resultado = false
    let c = codigoRes[linea_actual]
    var j = paramInicio

    var cond = [Int](count: 2, repeatedValue: 0)
    var indice = 0
    var comparador = 0
```

Ilustración 97. Evaluar condición

Los parámetros a utilizar son:

- **"resultado":** es lo que devolverá la función.
- **"c":** es la línea de código sobre la que se trabaja.
- **"j":** es una variable para recorrer los parámetros.
- **"cond":** es una tabla de dos enteros. En ella se almacenarán las variables, números y otros elementos que se vayan leyendo en "c", y cada vez que se almacene uno aumentará "indice", para saber dónde guardar el siguiente.
- **"comparador":** se guarda el código del comparador que se usará. Es decir, si es una igualdad, desigualdad, mayor que, etc.

Lo que hace esta función es leer los parámetros, almacenarlos en su variable correspondiente y, cuando se tengan dos elementos en "cond" y un comparador, se evaluará la condición. Ese momento debe coincidir con la lectura del último elemento de la línea, ya que si hay alguno más significa que no se ha escrito correctamente la línea.

```
while (j < c.getNumParam()) {
    if (c.getParam(j) == Codigo.VARIABLE) {
        let v = buscarVariable(c.getCadena(j))

        if (v != nil){
            cond[indice] = v!.getValor()
        }

        indice += 1
    } else if (c.getParam(j) == Codigo.NUMERO) {
        cond[indice] = Int(c.getCadena(j))!
        indice += 1

    } else if (c.getParam(j) >= Codigo.CTE_VACIO && c.getParam(j) < Codigo.FIN_CONSTANTES) {
        cond[indice] = c.getParam(j)
        indice += 1
    } else if (c.getParam(j) >= Codigo.IGUAL && c.getParam(j) <= Codigo.FIN_COMP) {
        comparador = c.getParam(j)
```

Ilustración 98. Almacenar parámetros comparación

Cuando se tengan todos los parámetros, se procede a evaluar la condición y se ponen las variables "indice" y "comparador" a cero para identificar después si ha sido correcta la lectura.

```
//Si ya se tiene un juego completo
if (indice == 2 && comparador != 0) {
    switch (comparador) {
    case (Codigo.IGUAL):
        resultado = (cond[0] == cond[1])

    case (Codigo.DESIGUAL):
        resultado = (cond[0] != cond[1])
```

Ilustración 99. Resultado comparación

Al final, se comprueba si la lectura ha sido correcta. Si el índice no es cero (es decir, si no se ha entrado en el if al final del bucle, donde se reinicia) significa que ha habido un error, a no ser que sea debido a que en la condición únicamente hay un valor booleano.

```
if (indice == 1 && cond[0] == Codigo.TRUE){
    resultado = true
} else if (indice == 1 && cond[0] == Codigo.FALSE){
    resultado = false
}else if (indice != 0){
    fin = true
    if(!ejecucion_prev){
        Toast("Error en los parametros del IF")
    }
    resultado = false
}
```

Ilustración 100. Comprobación lectura correcta

7.6.1.2 Leer variables

El ultimo tipo de línea que nos podemos encontrar es la declaración o asignación de una variable. La declaración es simple, se crea una variable con el tipo y el nombre que aparece en el código de la línea actual y se añade a la lista de variables creadas.

```
    //Creación de una variable
}else if (c.getCodigo() >= Codigo.DECLARAR ) {
    //Se crea una variable del tipo guardado en el parámetro 0 y el nombre de la cadena escrita
    let v = buscarVariable(c.getCadena())
    if(v != nil){
        self.variables.removeAtIndex(self.variables.indexOf{$0 === v}!)
    }
    self.variables.append(Variable(tipo: c.getParam(0),nombre: c.getCadena()))
```

Ilustración 101. Leer declaración de variable

Para la asignación de una variable se realizan más pasos.

1. Primero se comprueba el tipo de la variable, ya que en función del tipo se comprueba si por ejemplo es un número o un valor booleano.
2. Se comprueba el formato, es decir, si el valor a asignar es un número, una variable, el resultado de una acción como “mirar”, etc.
3. Finalmente, se establece el valor en la variable. Si el tipo no es correcto, se muestra un error.

```
    //Asignación
} else if (c.getCodigo() == Codigo.ASIGNACION){

    let v = buscarVariable(c.getCadena(0))
    if (v != nil) {

        if(v!.getTipo() == Codigo.VAR_INT) {
            //Si es del formato 'var = valor'
            if (c.getNumParam() == 2 && (c.getParam(1) == Codigo.VARIABLE || c.getParam(1) == Codigo.NUMERO)) {
                if (c.getParam(1) == Codigo.NUMERO){
                    v!.setValor(Int(c.getCadena(1))!)

                } else if (c.getParam(1) == Codigo.VARIABLE) {
                    let v2 = buscarVariable(c.getCadena(1))

                    if (v2 != nil && v!.getTipo() == v2!.getTipo()){
                        v!.setValor(v2!.getValor())
                    } else {
                        if(!ejecucion_prev){
                            Toast("Error en la asignación")
                        }
                        fin = true
                    }
                }
```

Ilustración 102. Leer asignación de variable

7.6.1.3 Salir

En la parte superior de la pantalla siempre habrá dos botones. Uno de ellos tiene el texto “Atrás”, y lleva al usuario de vuelta a la vista en la que se crea el código. El otro botón, lleva al usuario a la siguiente pantalla que le corresponde jugar, que cambia según el estado de la partida y el tipo de nivel.

```
//Funcion para salir al mapa, se muestra alerta
@IBAction func SalirButton(sender: UIButton) {

    let story : UIStoryboard = UIStoryboard(name: "Main", bundle: nil)

    // Se gana y hay mas niveles
    if (victoria && parserJson.haySiguienteNivel()) {
        //Se actualiza el nivel máximo alcanzado
        preferences.setInteger(etapa+1, forKey: nivel+"_max")
        preferences.synchronize()

        //indicamos vista a la que se pasa
        etapa = etapa + 1

        // Pasar a vista historia
        let miVistaDos = story.instantiateViewControllerWithIdentifier("historia") as! HistoriaViewController

        miVistaDos.nivel = nivel
        miVistaDos.etapa = etapa

        self.presentViewController(miVistaDos, animated:false, completion:nil)

        // Se gana y no hay mas niveles
    } else if (victoria){

        if (!preferences.boolForKey(nivel+"_extra")) {

            //Se añaden las ayudas extra y se suma la experiencia.
            let s = parserJson.getAyuda("ayuda_extra")
```

Ilustración 103. Botón salir en Resolver

Si se gana y hay más niveles se pasa a la vista en la que se muestra el dialogo. Si se gana y no hay mas niveles, se añaden las ayudas extras y la experiencia a las preferencias del usuario y se sale al mapa. Si no se ha ganado se sale al mapa y se pierde el progreso del nivel.

7.7 Preferencias del usuario

Para el desarrollo del videojuego será necesario almacenar la información del progreso que lleva el usuario. Algunos de los parámetros serán la experiencia, las teclas y ayudas desbloqueadas o el lenguaje utilizado.

Existen variables como:

- Las que empiezan por “control_” se utilizan para que únicamente se ejecute algo una vez. Por ejemplo, “control_inicializado” guarda el valor *true* cuando se realiza la configuración inicial del juego.
- Al iniciar por primera vez cada nivel hay que realizar algunas acciones de configuración (como desbloquear las teclas). Cuando esta configuración se realiza, se guarda *true* en la preferencia compartida “nombredelnivel_etapa”. Este campo también se utiliza para saber si se ha entrado en este

nivel alguna vez.

- Algunas son datos personales del personaje y el jugador: “ciudad” guarda la última ciudad visitada, “lenguaje” el lenguaje configurado, “experiencia” el nivel del jugador (al completar con éxito un nivel aumenta la experiencia del jugador, permitiéndole acceder a misiones nuevas), “constantes” y “variables” guardan un entero con el número de constantes y variables desbloqueadas, respectivamente.
- “tecla_i_j” guarda *true* cuando la tecla j del teclado i ha sido desbloqueada.
- “ayuda_titulo_i” y “ayuda_i” se utilizan para mostrar los apuntes ya vistos en el libro de apuntes.

Para conseguir guardar esta información vamos a usar la clase *NSUserDefaults*. Nos permite guardar datos que podremos recuperar la próxima vez que iniciemos la aplicación.

Primero es necesario obtener una referencia a las preferencias del usuario, llamando al método *standardUserDefaults* de la clase *NSUserDefaults*.

```
//Leer datos usuario
preferences = NSUserDefaults.standardUserDefaults()
```

Disponemos de métodos para almacenar los datos según el tipo que sean como: *setBool*, *setInteger*, *setFloat* o *setObject*. A estos métodos se les pasa el valor y la clave que utilizaremos para obtener el valor posteriormente.

```
preferences.setObject("pueblo_inicio", forKey: "ciudad")
```

Para obtener los datos también disponemos de métodos equivalentes como: *boolForKey*, *integerForKey*, *floatForKey*, *objectForKey* o *stringForKey*.

```
ciudad = preferences.stringForKey("ciudad")!
```

Si necesitamos guardar los datos automáticamente antes de la sincronización automática, es necesario llamar al método synchronize tras haber establecido un valor.

```
preferences.synchronize()
```

También existe la posibilidad de borrar los datos del usuario, esto se necesitará cuando deseemos reiniciar el juego o cambiar de lenguaje. La clase *NSUserDefaults* dispone del método *removePersistentDomainForName* para realizarlo.

```
NSUserDefaults.standardUserDefaults().removePersistentDomainForName(NSBundle.
mainBundle().bundleIdentifier!)
```

En este apartado se ha detallado la implementación del sistema. Se han visto los elementos que componen la interfaz gráfica y como funcionan. Se ha explicado la estructura de los niveles, es decir, los diálogos, resumen, como se escribe el código y la forma de resolver los niveles. Por último, se ha detallado como se crea el tablero y las distintas animaciones que realizan los personajes.

8 PRUEBAS

El objetivo de este apartado es describir un conjunto de pruebas que se ejecutarán para comprobar que el sistema cumple con las funciones y la calidad esperada.

8.1 Definición

Se van a definir el conjunto de pruebas a realizar.

Prueba	Prueba 1
Objetivo	Iniciar tutorial en la primera ejecución
Descripción	Se abrirá la aplicación y aparecerá un tutorial. Esto solo ocurrirá la primera vez que se ejecute.
Resultado	Superada

Tabla 22. Prueba 1

Prueba	Prueba 2
Objetivo	Añadir nuevos apuntes
Descripción	Cuando se termine un nivel y se hayan desbloqueado nuevos apuntes, debe aparecer un mensaje en el mapa indicando los apuntes añadidos. Cuando se pulse el botón que permite leer los apuntes, se cargará el apunte nuevo.
Resultado	Superada

Tabla 23. Prueba 2

Prueba	Prueba 3
Objetivo	Elementos aleatorios
Descripción	Los elementos del mapa que sean aleatorios se deben mostrar semitransparentes en la vista Resumen. Luego, en la vista Resolver, se debe mostrar una de las posibilidades.
Resultado	Superada

Tabla 24. Prueba 3

Prueba	Prueba 4
Objetivo	Errores en la ejecución

Descripción	Si existe un error en el código escrito por el jugador cuando se este resolviendo el nivel, se debe mostrar un mensaje indicando el error. Además, si es la primera vez que ocurre, se le indicará al jugador que vuelva hacia atrás.
Resultado	Superada

Tabla 25. Prueba 4

Prueba	Prueba 5
Objetivo	Añadir nuevos niveles
Descripción	Cada vez que se termine un nivel se añadirá el siguiente a la lista de niveles. Además, se debe indicar que el nivel es nuevo.
Resultado	Superada

Tabla 26. Prueba 5

Prueba	Prueba 6
Objetivo	Reiniciar el juego
Descripción	Cuando se pulse el botón “Reiniciar” se deben borrar los datos del usuario y comenzar la aplicación como si fuera la primera vez.
Resultado	Superada

Tabla 27. Prueba 6

Prueba	Prueba 7
Objetivo	Cambio de lenguaje
Descripción	Cuando se seleccione un lenguaje distinto del actual, se debe reiniciar el juego y establecer como lenguaje de codificación del juego el lenguaje seleccionado.
Resultado	Superada

Tabla 28. Prueba 7

Prueba	Prueba 8
Objetivo	Dibujar tablero
Descripción	Se comprobará la correcta representación del tablero definido en el archivo JSON del nivel correspondiente.

Resultado	Superada

Tabla 29. Prueba 8

Prueba	Prueba 9
Objetivo	Escritura de código
Descripción	Cuando se escriba el código con el teclado proporcionado, deberá aparecer en la tabla el texto correspondiente.
Resultado	Superada

Tabla 30. Prueba 9

Prueba	Prueba 10
Objetivo	Animaciones del personaje
Descripción	El personaje debe realizar animación cuando esta en reposo, cuando se mueve por el tablero o cuando realiza un golpe.
Resultado	Superada

Tabla 31. Prueba 10

8.2 Ejecución

A continuación se va a mostrar el resultado de estas pruebas.

8.2.1 Prueba 1

Se va a ejecutar la aplicación por primera vez y avanzar en el nivel. En la vista Historia y en la vista Resolver se indica, mediante la imagen, que es necesario pulsar sobre la pantalla para que se avance. En el primer caso, irán apareciendo mensajes, y en el segundo se irán ejecutando líneas de código. Por último, en la vista CrearCodigo se muestra una ayuda explicando cómo escribir el código.

Ilustración 104. Prueba 1

8.2.2 Prueba 2

Para realizar esta prueba se terminara un nivel en el que haya apuntes nuevos y se pasara a la pantalla principal. Nos aparecerá un texto con los apuntes añadidos y una indicación de que hay apuntes nuevos. Además, se es la primera vez que ocurre, se mostrará una animación indicando que hay que pulsar el botón.

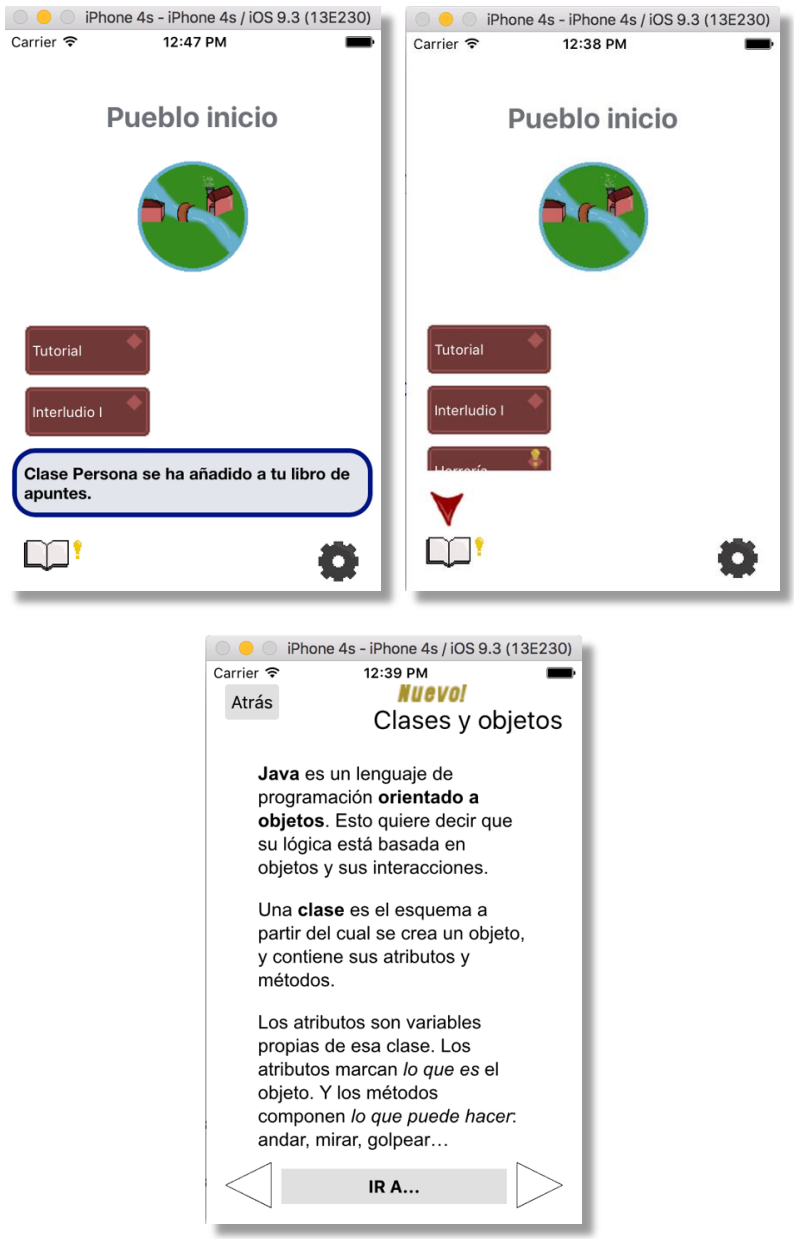


Ilustración 105. Prueba 2

8.2.3 Prueba 3

Se ejecutará un nivel que incluya elementos aleatorios. En la vista Resumen se mostrarán las opciones semitransparentes, una de las posiciones con el objeto y en las demás una imagen con un signo de interrogación. En la vista Resolver aparecerá una de las opciones.

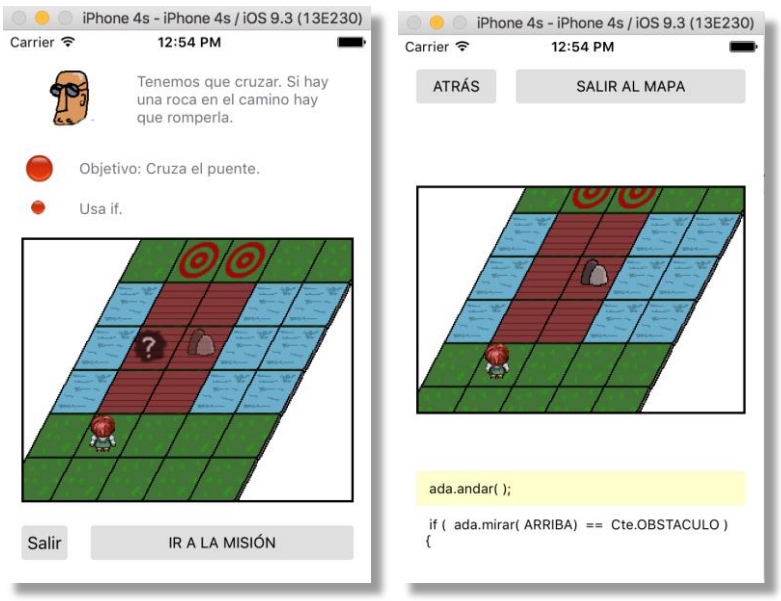


Ilustración 106. Prueba 3

8.2.4 Prueba 4

Para realizar esta prueba se escribirá un código incorrecto. Luego en la vista Resolver , cuando se lea la línea, se debe mostrar un error. Si es la primera vez que ocurre un error o no se resuelve el nivel, se debe indicar al usuario que pulse el botón “atrás” para arreglar el código.

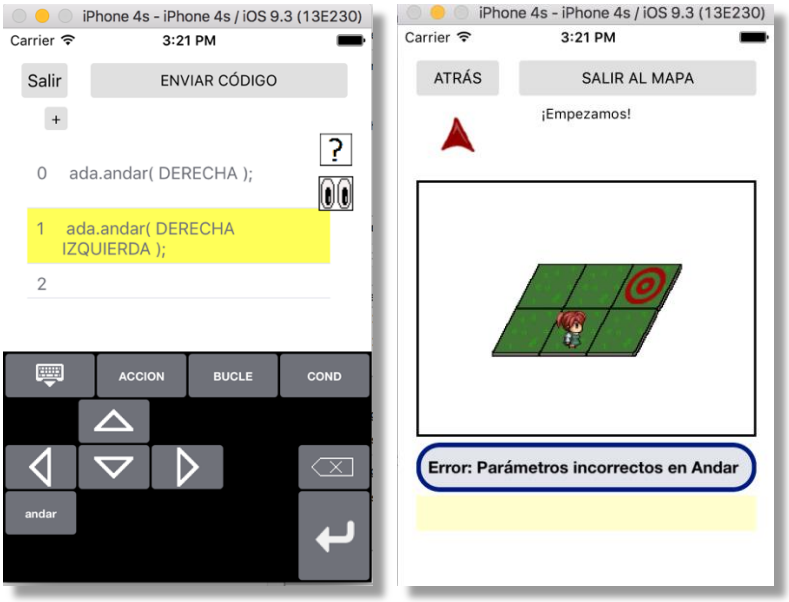


Ilustración 107. Prueba 4

8.2.5 Prueba 5

Para esta prueba resolvemos un nivel y cuando volvamos al mapa aparecerá el nivel nuevo. La imagen será distinta mostrando un signo de exclamación, para indicar que es el nuevo.

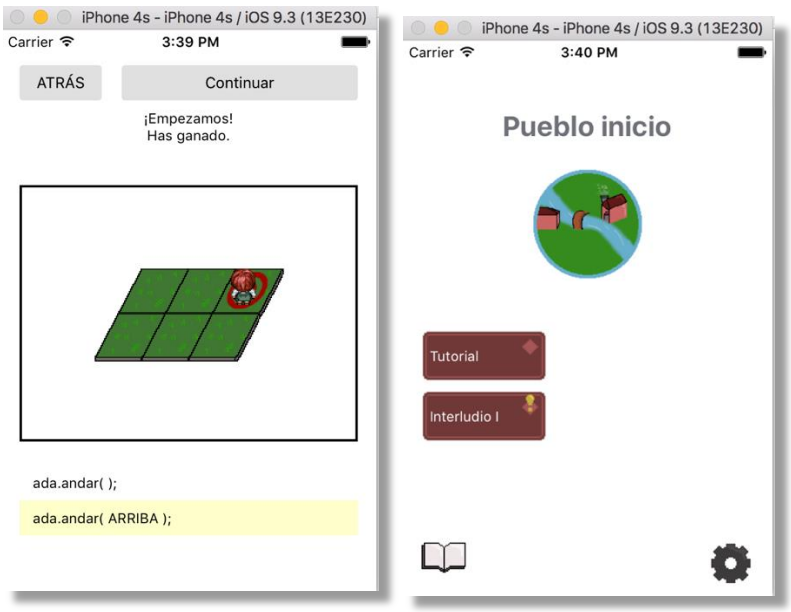
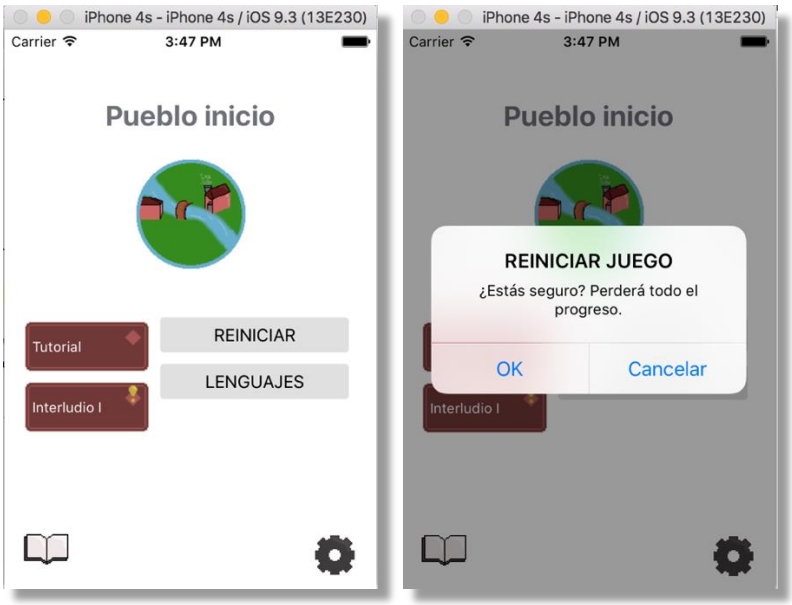


Ilustración 108. Prueba 5

8.2.6 Prueba 6

Para reiniciar el juego pulsamos el botón ajustes y posteriormente el botón reiniciar. Luego se iniciará la aplicación mostrando el tutorial.



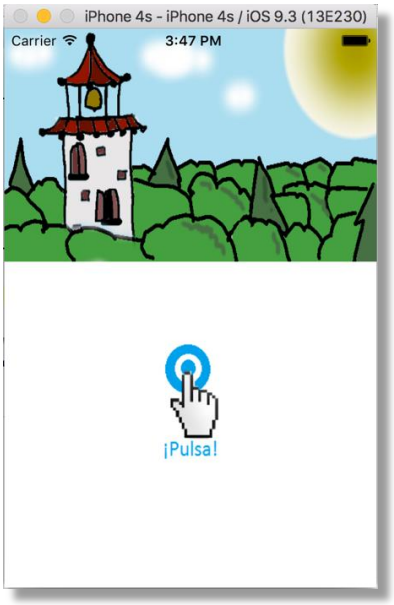
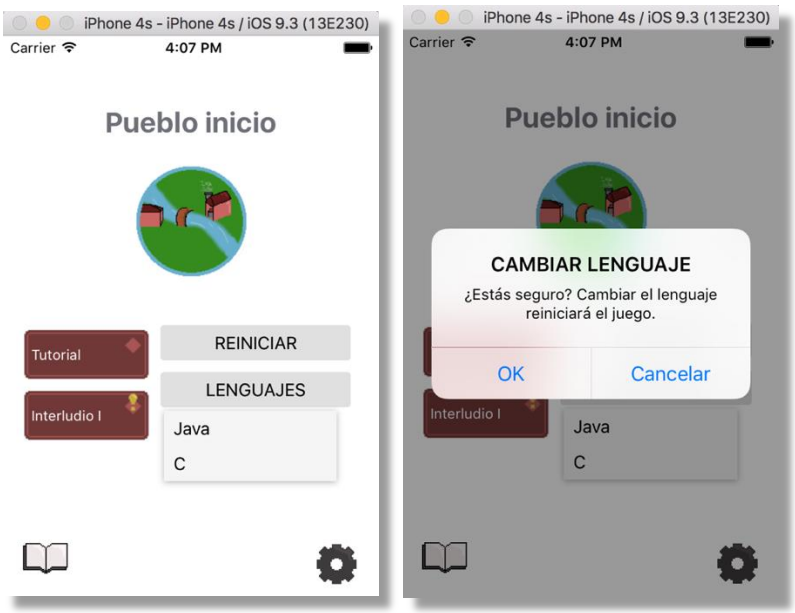


Ilustración 109. Prueba 6

8.2.7 Prueba 7

Para reiniciar el juego pulsamos el botón ajustes, el botón lenguaje y finalmente el lenguaje que no estemos utilizando actualmente. Luego se iniciará la aplicación mostrando el tutorial y comprobamos que se haya cambiado la codificación.



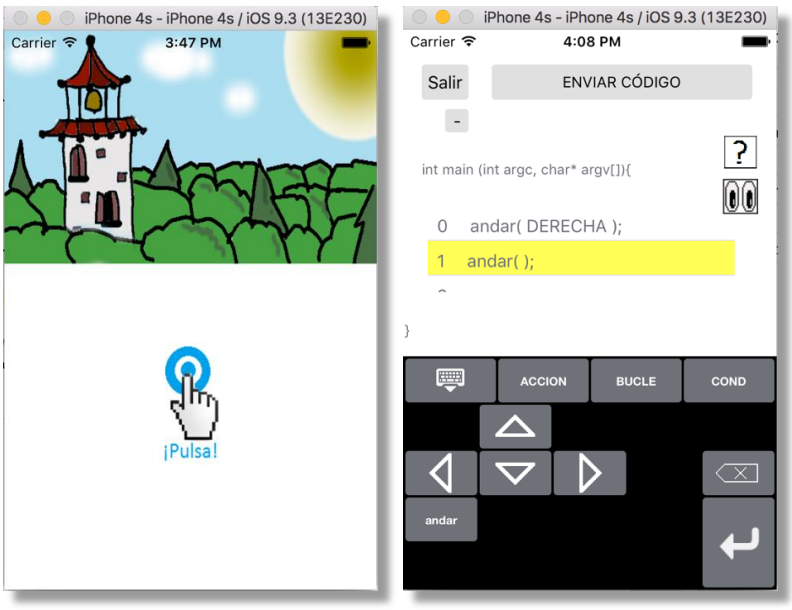


Ilustración 110. Prueba 7

8.2.8 Prueba 8

Para realizar esta prueba se comparara el archivo JSON que define el tablero de un nivel y el resultado que se muestra. En este caso se ve que es un tablero de 5 columnas y 5 filas, el tablero base es el ladrillo y disponemos de ítems en las posiciones 3, 8, 11, 14 y 22.

```
"mapa": {
    "filas": "5",
    "columnas": "5",
    "baldosaBase": "baldosa_ladrillo",

    "item": [
        {
            "posicion":"3",
            "tipo":"fijo",
            "bitmap":"sprite_espadas"
        },
        {
            "posicion":"8",
            "tipo":"NPC",
            "ID":"herrero",
            "bitmap":"sprite_herrero",
            "direccion":"abajo"
        },
        {
            "posicion":"22",
            "tipo":"fijo",
            "bitmap":"sprite_fragua"
        },
        {
            "posicion": "14",
            "direccion": "abajo",
            "tipo": "diana",
            "salud": "1",
            "ID":"yunque",
            "bitmap":"sprite_yunque"
        },
        {
            "posicion": "14",
            "direccion": "abajo",
            "tipo": "roca",
            "salud": "100",
            "ID": "yunque",
            "bitmap": "sprite_yunque"
        },
        {
            "posicion": "11",
            "direccion": "derecha",
            "tipo": "ada",
            "bitmap": "sprite_ada"
        }
```

Ilustración 111. Prueba 8

8.2.9 Prueba 9

En este caso vamos a escribir una condición simple como que el personaje golpee si existe un obstáculo a la derecha. Así comprobamos que podemos escribir código fácilmente.

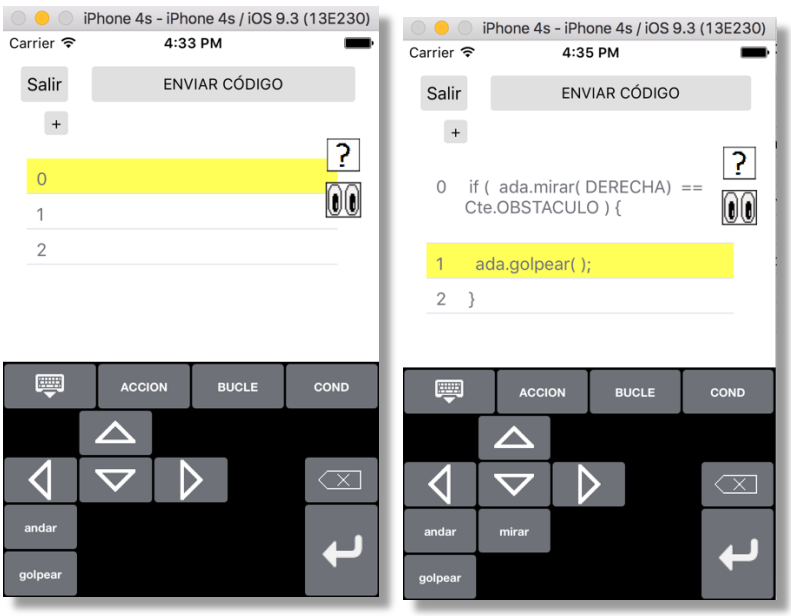
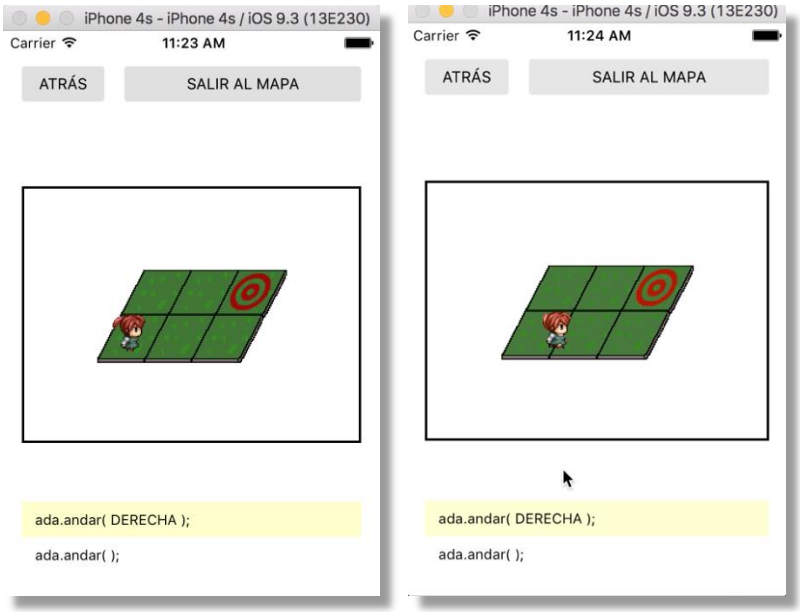


Ilustración 112. Prueba 9

8.2.10 Prueba 10

Para esta prueba se va a observar el comportamiento del personaje cuando está en reposo y cuando se mueve de una posición a otra. Para ello se escribirá un código que mueva al personaje.



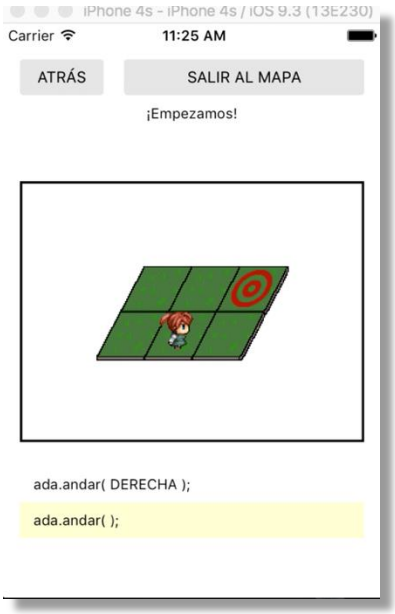
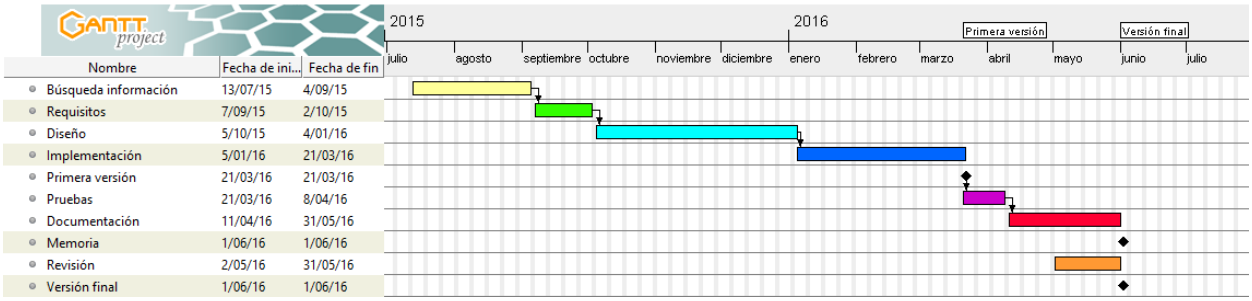


Ilustración 113. Prueba 10

Se han realizado un conjunto de pruebas en los que se ha comprobado el correcto funcionamiento de algunos aspectos importantes del sistema. Entre ellos, podemos destacar, la primera ejecución de la aplicación o la lectura del código escrito y las animaciones correspondientes que se deben mostrar.

9 PLANIFICACIÓN

A continuación se explica el desarrollo temporal de cada uno de los elementos del proyecto. Se ha realizado un diagrama de Gantt para poder entender el proceso de una forma más clara y sencilla, indicando además la dependencia de los elementos.



Nombre	Fecha de ini...	Fecha de fin
Búsqueda información	13/07/15	4/09/15
Requisitos	7/09/15	2/10/15
Diseño	5/10/15	4/01/16
Implementación	5/01/16	21/03/16
Primera versión	21/03/16	21/03/16
Pruebas	21/03/16	8/04/16
Documentación	11/04/16	31/05/16
Memoria	1/06/16	1/06/16
Revisión	2/05/16	31/05/16
Versión final	1/06/16	1/06/16

Ilustración 114. Planificación del proyecto

La primera fase corresponde a la búsqueda de información con una duración de alrededor unos dos meses. Se procederá a estudiar las características del sistema operativo iOS, las herramientas para desarrollar en este sistema y el lenguaje que se utilizará para codificar.

Antes de empezar el diseño, habrá una pequeña fase en la que se analizará el proyecto y como resultado se obtendrá la especificación de requisitos. La duración se ha estimado en tres semanas y consistirá en describir el comportamiento de la aplicación que se va a desarrollar.

El diseño será el proceso más largo, con una duración de tres meses. Se analizarán los requisitos y se describirá la estructura y los componentes de la aplicación.

Una vez finalizada la fase de diseño, se procederá a la implementación de éste. Se codificará el diseño realizado utilizando el lenguaje de programación y la plataforma elegida.

Las pruebas de software se realizarán tras acabar la implementación. Tendrán una duración de poco menos de un mes. Se harán pruebas de distintos tipos para detectar defectos y problemas. Como resultado se deben validar y verificar los requisitos especificados.

La fase de documentación tendrá una duración aproximada de un mes y medio. Se elaborará una memoria que recogerá la información general del proyecto. Se explicarán los objetivos, como se realizará el proyecto y los resultados obtenidos.

Por último, habrá una fase de revisión en la que se corregirán errores detectados y mejoras que necesite la aplicación.

10 CONCLUSIONES

10.1 Dificultades encontradas

Durante el trabajo se han encontrado dificultades debido al desconocimiento del desarrollo de aplicaciones móviles, en este caso en un sistema cerrado como el que ofrece Apple. El lenguaje elegido, Swift, se desarrolló hace pocos años y todavía no dispone de la cantidad de ejemplos y bibliotecas que dispone Objective-C. Sin embargo, es más sencillo y se está impulsando fuertemente desde Apple.

La interfaz gráfica también ha sido una dificultad, debido a no estar acostumbrado a este tipo de programación, aunque el IDE Xcode ofrece una interfaz en la que la mayoría de los elementos se pueden establecer sin escribir código.

Los problemas se han resuelto con la numerosa documentación que se puede encontrar, y mediante páginas web en las que los usuarios plantean dudas y otros les ayudan a resolverlo.

10.2 Líneas futuras

Como mejoras, se podría conseguir una interfaz más profesional y vistosa, con animaciones más complejas. Por otra parte, se podrían introducir elementos relacionados con la programación avanzados, aunque en un juego como este no sea necesario utilizarlo.

El proyecto se ha centrado en el correcto funcionamiento de las diferentes interfaces de la aplicación. Con ello, se han diseñado niveles que se puedan resolver de acuerdo al comportamiento desarrollado. Posteriormente, se podrían añadir más niveles combinando los elementos existentes o introduciendo nuevos.

10.3 Valoraciones finales

El proyecto ha cumplido los objetivos fijados. Se ha adaptado la aplicación anterior desarrollada en Android al sistema iOS, aprovechando las similitudes y las diferencias de ambos sistemas. Esta aplicación puede ayudar a tener un primer contacto con la programación para aquellos que no lo conozcan, y cambiar la forma de pensar, ya que los ordenadores trabajan de una forma estructurada y secuencial que difiere del pensamiento humano.

Un usuario que termine el videojuego debe de disponer los conocimientos básicos de los lenguajes de programación, sobre todo las estructuras de control. En este proyecto no se ha entrado en detalles complejos, pero proporciona una base para poder realizarlo mediante otros medios.

Se ha conseguido crear niveles que permitan mostrar las características de la aplicación. Los apuntes permiten obtener una base teórica para los niveles, y la forma de resolverlos, mediante el teclado proporcionado, resulta intuitiva para el usuario.

Por último, la parte gráfica de los niveles ha conseguido mostrar los efectos de las acciones que se realizan en el código. Las imágenes y animaciones siguen el modelo tradicional de los juegos en dos dimensiones.

Con este proyecto he conseguido aprender a estudiar de forma autodidacta un lenguaje de programación nuevo y tecnologías desconocidas. Resolver problemas más complejos de los encontrados en anteriores proyectos, y ser capaz de afrontar uno de mayor envergadura, similar a lo que se puede encontrar en el mundo laboral.

11 BIBLIOGRAFÍA

[1] The Swift Programming Language (Swift 2.1)
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/

[2] Neuburg, M. (2014). "*Programming iOS 8: Dive Deep into Views, View Controllers and Frameworks"*.

[3] Tutoriales de Swift en español | TutorialSwift.es
http://tutorialswift.es/

[4] Stack Overflow
http://stackoverflow.com

[5] Configuring Your Xcode Project for Distribution
https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/ConfiguringYourApp/ConfiguringYourApp.html

[6] The App Life Cycle
https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html

[7] About the iOS Technologies
https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007898-CH1-SW1

[8] Taxonomía de requisitos | Marco de Desarrollo de la Junta de Andalucía
http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/408

[9] Flowchart Maker & Online Diagram Software
https://www.draw.io/

[10] JSON
http://www.json.org/json-es.html

[11] Programación estructurada: Lenguaje C/C++
http://prog-estructurada.blogspot.com.es/2011/01/lenguaje-c.html

[12] 7 apps para aprender a programar
http://programamos.es/7-apps/

12 ANEXO

En este anexo se proporciona un glosario de términos utilizados en la memoria, un manual de usuario para un correcto uso de la aplicación y una introducción al lenguaje Swift, explicando las bases de este lenguaje.

12.1 Glosario de términos

A continuación se van a definir una serie de términos utilizados en la memoria.

- **Aplicación nativa:** Se denomina así a una aplicación desarrollada en el lenguaje nativo del dispositivo. Por ejemplo, se utiliza Java para desarrollar en Android o Swift para desarrollar en iOS.
- **Framework:** Es un entorno de trabajo para el desarrollo de software. Está compuesto por librerías y recursos que podemos incluir en el proyecto y acceder a todas sus características.
- **JSON:** Es un formato ligero para almacenar e intercambiar datos. Está basado en un subconjunto del lenguaje de programación JavaScript.
- **HTML:** Es un lenguaje de marcado para la elaboración de páginas web. Determina el contenido de una página, se usa para crear y representarla visualmente, no determina su funcionalidad.
- **XML:** Es un lenguaje de etiquetado simple para el intercambio de gran variedad de datos. Es muy similar a HTML, pero su función principal es describir datos.
- **IDE:** Es un programa informático que contiene herramientas para desarrollar código fácilmente, como un editor de código, un compilador, depurador, resaltado de errores. Ayudan a trabajar de una forma más sencilla.

12.2 Manual de usuario

A continuación se van a detallar los pasos para utilizar la aplicación.

A la hora de ejecutar la aplicación existen dos opciones:

- Utilizar el simulador proporcionado por Xcode. En este caso se debe seleccionar el dispositivo que queremos simular, las distintas versiones de iPhone y iPad.

- Utilizar un dispositivo iPhone. En este caso se debe conectar con un cable USB el móvil al ordenador y seleccionar el dispositivo conectado. La aplicación permanecerá instalada en el dispositivo y se podrá ejecutar sin estar conectado.

Primero aparecerá una imagen mientras se carga la aplicación.

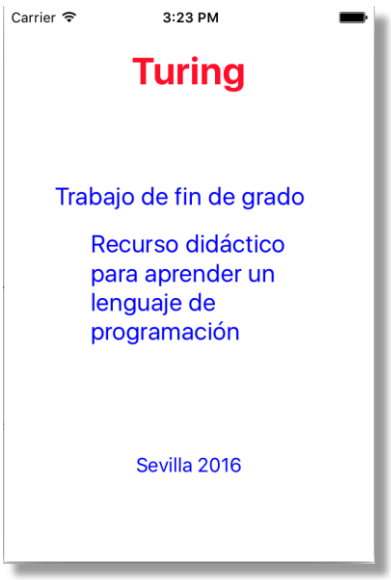


Ilustración 115. Manual: Pantalla de inicio

La vista principal contiene varios elementos como el mapa actual, una lista con las misiones disponibles, un botón para ver los apuntes y otro para los ajustes.

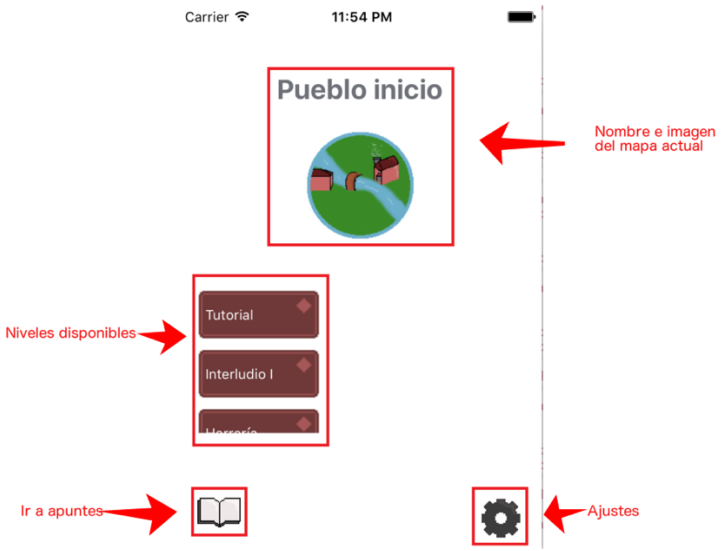


Ilustración 116. Manual: Vista principal

Si pulsamos sobre los ajustes se puede reiniciar el juego o cambiar el lenguaje de codificación que se va a utilizar.

Ilustración 117. Manual: Ajustes

Si se pulsa en los apuntes, pasamos a otra vista. En la parte central aparece un texto que contiene una ayuda sobre un tema que se especifica en la parte superior. Existen varios botones, el primero de ellos es para volver a la vista principal, hay dos a los lados que sirven para moverse hacia la siguiente página o a la anterior.

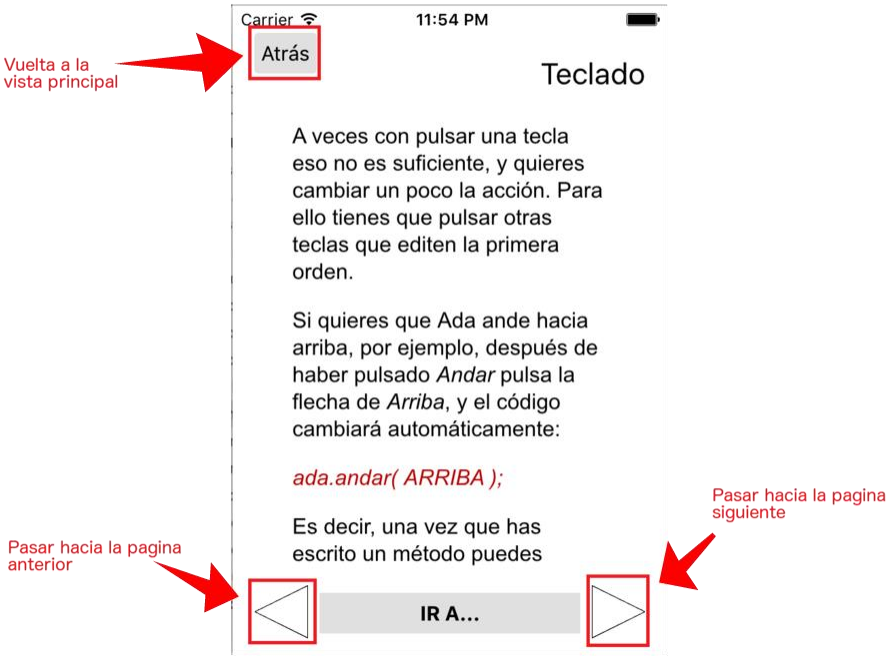


Ilustración 118. Manual: Apuntes

También podemos saltar hacia una página, para ello se pulsa el botón “Ir a” que muestra una lista con las páginas desbloqueadas. Si pulsamos en una de ellas se muestra la página.

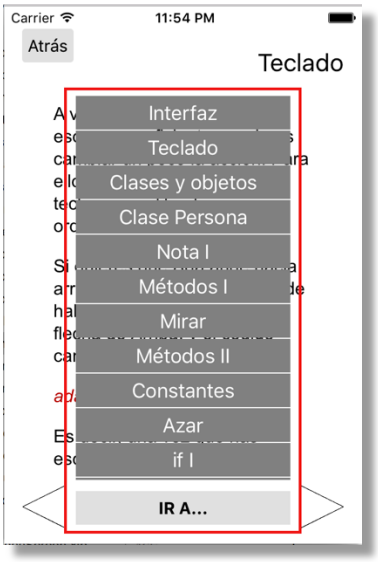


Ilustración 119. Manual: Lista apuntes

Si volvemos a la vista principal y seleccionamos un nivel, aparece una breve descripción y un botón para comenzar el nivel. En el caso de haberlo comenzado superando una etapa, aparecerá otro botón para continuar por la que no se completó.

Ilustración 120. Manual: Descripción nivel

Al comenzar el nivel, aparecerá la vista Historia. Se mostrará a personajes manteniendo un dialogo, cada vez que se pulse en la pantalla aparecerá un nuevo mensaje.

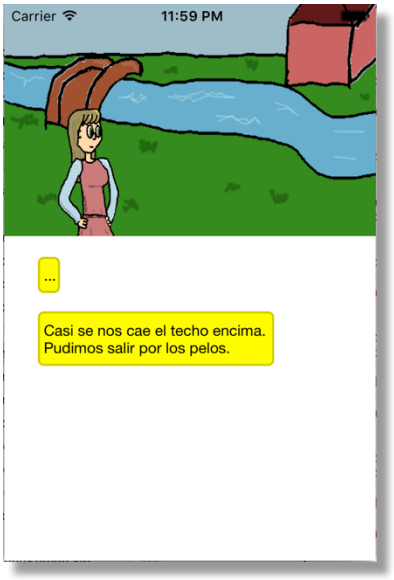


Ilustración 121. Manual: Historia

Si el nivel ya se hubiera realizado, se muestra un botón con la opción de saltar la conversación.

Ilustración 122. Manual: Historia saltar

Cuando se hayan leído todos los mensajes aparecerá un botón para avanzar en el nivel.

Ilustración 123. Manual: Historia siguiente

En la siguiente vista podemos leer un resumen del nivel y lo que se necesita para resolverlo. Se permite salir del nivel o continuar, es decir, pasar a la siguiente vista en la que se escribirá el código.

En este nivel se indica que es necesario golpear el hierro. Por ello, habrá que escribir un código que permita a *Ada* colocarse enfrente y golpear.

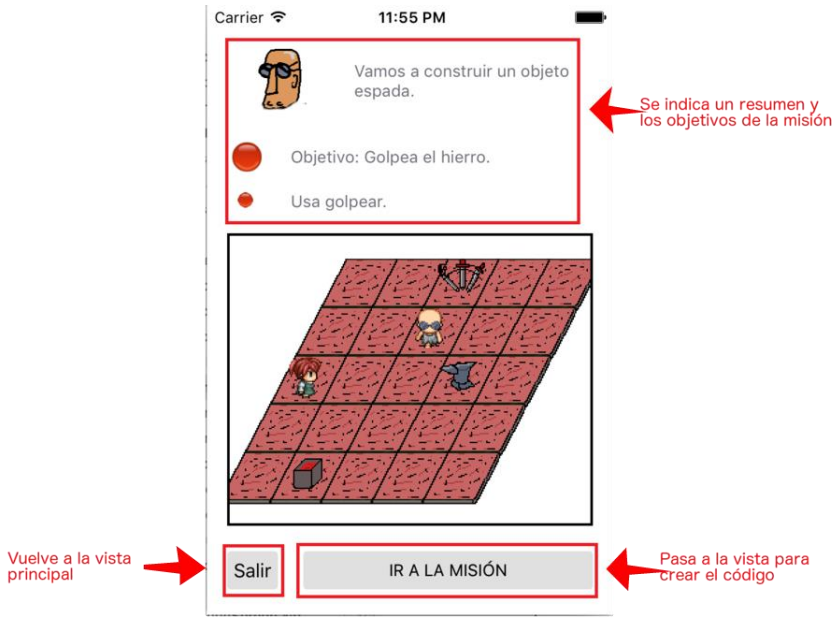


Ilustración 124. Manual: Resumen

Si pulsamos “Ir a la misión”, pasamos a escribir el código. En la vista tenemos varios elementos como dos botones, uno para salir y otro para pasar a resolver el nivel. En el centro aparece una tabla con el código y en la parte inferior está el teclado. Además, existen dos botones en la parte derecha, uno para mostrar una ayuda para el nivel y otro para volver a ver el mapa.

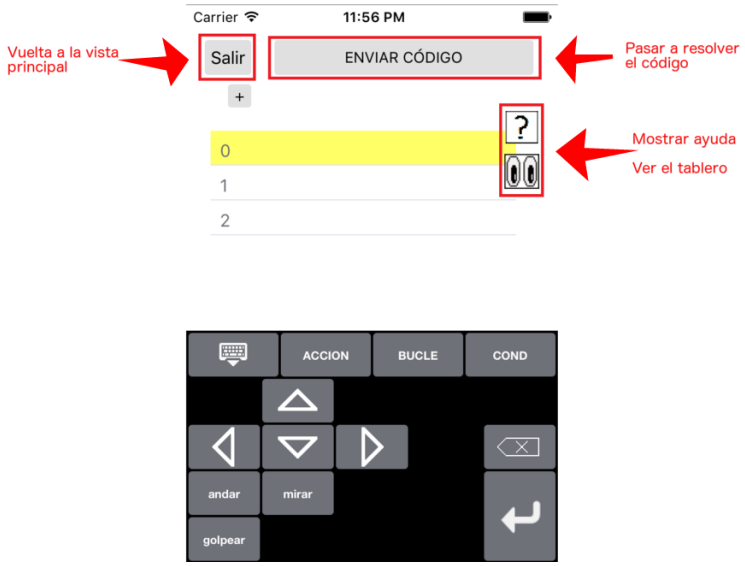


Ilustración 125. Manual: Crear código

La ayuda muestra los apuntes que estén relacionados con el nivel actual. Si todavía no se han desbloqueado, lo harán cuando se complete el nivel.

Ilustración 126. Manual: Ayuda código

Pulsando el botón "+" se muestra el código necesario para que funcionara en un aplicación real, declarando la clase *Turing*. Si el lenguaje elegido es C, aparece otro código.

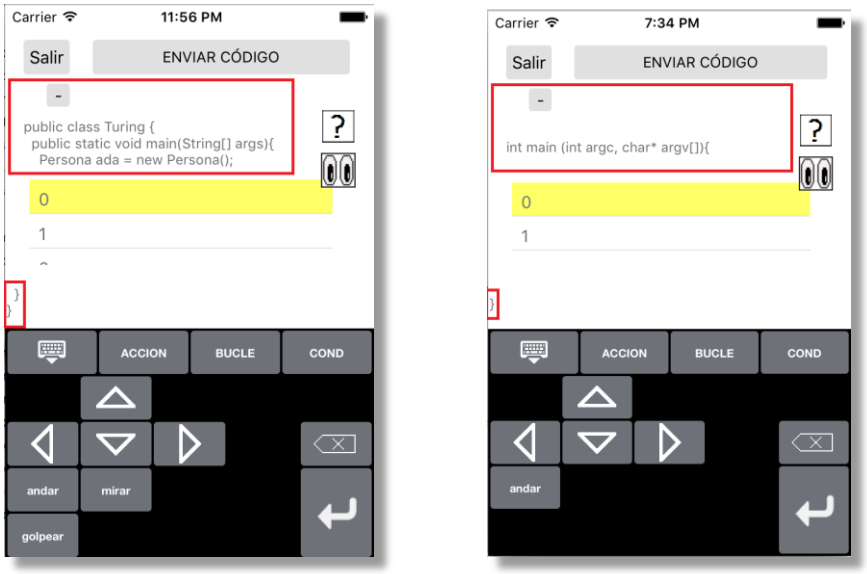


Ilustración 127. Manual: Clase código

El teclado puede cambiar de botones pulsando uno de los botones principales de la parte superior. Permite ocultar el teclado o cambiar el tipo de teclado, cada una de las teclas escribirá un código en la tabla.

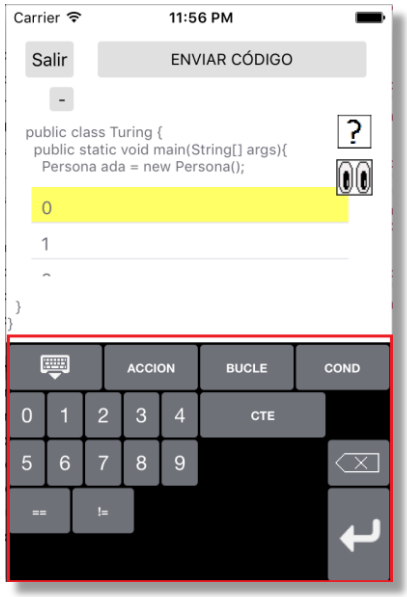


Ilustración 128. Manual: Teclado código

Observando el mapa y los objetivos, escribimos el código necesario mediante el teclado para resolver la etapa.

Ilustración 129. Manual: Código nivel

Finalmente en la vista resolver tenemos un botón para salir y otro para volver atrás. Si pulsamos en la pantalla se van ejecutando las líneas

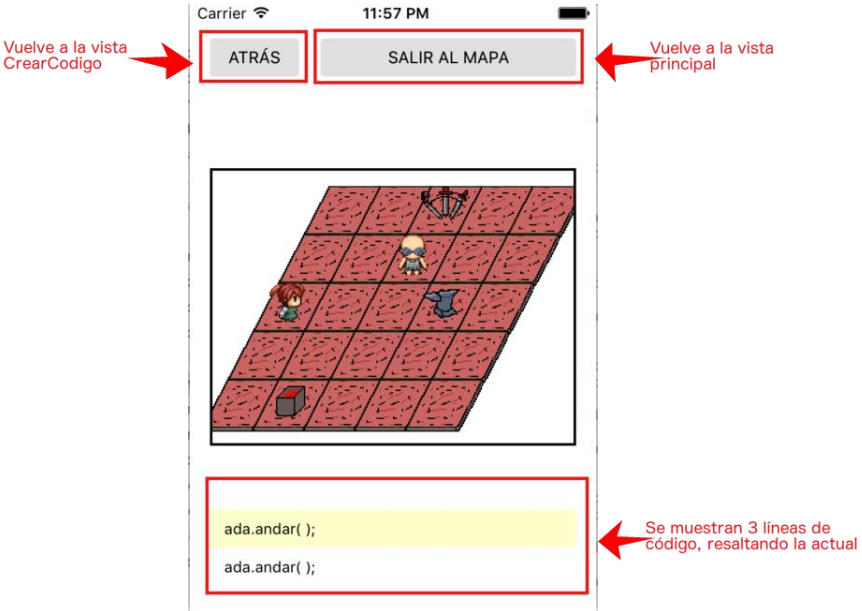


Ilustración 130. Manual: Resolver

Según se vayan ejecutando las líneas, aparecerá información sobre el nivel hasta que se gana o no. Si se gana el botón cambia el texto indicando que se continúe.

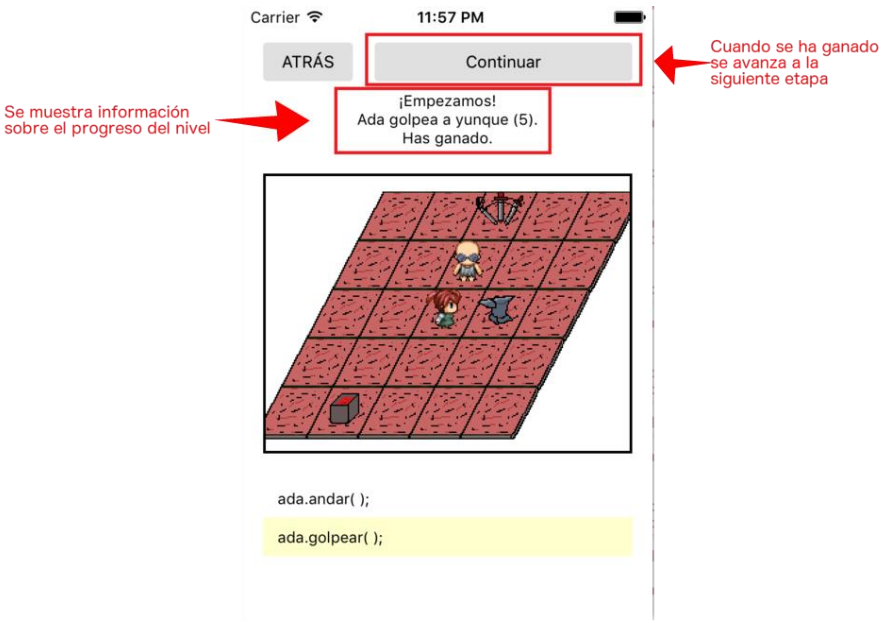


Ilustración 131. Manual: Resolver final

12.3 Swift

En este apartado se va a realizar una pequeña introducción a los elementos básicos de Swift. Un lenguaje creado por Apple que en un futuro debe ser el que domine las aplicaciones para iOS y OS X. Además, en comparación con Objective-C, tiene un mejor rendimiento y es un lenguaje mucho más intuitivo y fácil de aprender.

Combina lo mejor de C y Objective-C, sin las limitaciones de la compatibilidad con C. Adopta patrones de programación seguros y añade características modernas para hacer la programación más sencilla y flexible.

12.3.1 Básico

Swift proporciona sus propias versiones de todos los tipos C y Objective-C fundamentales, incluido *Int* para los números enteros; *Double* y *Float* para los valores de punto flotante; *Bool* para valores booleanos; y *String* para datos textuales. También ofrece versiones potentes de los dos tipos de colecciones principales, *Array y Dictionary.*

Introduce tipos opcionales, que se encargan de la ausencia de un valor. Estos tipos indican o bien “no es un valor, y es igual a x” o “no es un valor en absoluto”.

12.3.1.1 Valores simples

Se usa *let* para declarar una constante y *var* para declarar una variable. El valor de una constante no necesariamente debe ser conocido al momento de compilar, pero solo se le debe asignar valor una vez. Las constantes son para valores que se asignan una vez pero se usan en varias ocasiones.

```
var numeroVariable = 10
numeroVariable = 20
let numeroConstante = 20
```

Una constante o variable debe tener el mismo tipo del valor que se le asigna. No siempre es necesario escribir el tipo explícitamente. Proveer un valor cuando se crea una constante o variable deja al compilador inferir el tipo del dato. En el ejemplo anterior, numeroVariable es un *Int* porque su valor inicial es *Int*.

Cuando el valor inicial no provee suficiente información o no hay valor inicial, se especifica el tipo escribiéndolo después del nombre y separándolo con dos puntos.

```
let numeroDouble: Double = 20
```

Los valores nunca son convertidos de forma implícita de un tipo a otro. Si se necesita convertir un valor de un tipo a otro, se realiza una conversión explícita del tipo deseado.

```
let cadena = "El numero de líneas es "
let numero = 20
let cadenanumero = cadena + String(numero)
```

Estos valores se pueden incluir escribiendo el valor entre paréntesis, y escribir " \ " antes del paréntesis.

```
let numero = 20
let cadenanumero = "El numero de líneas es \(numero)"
```

12.3.1.2 Tipos complejos

Se pueden crear *arrays* y *diccionarios* usando " [ ] " y acceder a su valor escribiendo su índice.

```
var colores = ["azul","blanco","negro", "rojo"]
colores[1] = "amarillo"
var capitales = [
  "España":"Madrid",
  "Francia":"París"
   ]
capitales["Italia"] = "Roma"
```

También se pueden crear *arrays* y *diccionarios* vacíos.

```
let arrayvacio = [String]()
let diccionariovacio = Dictionary<String, Float>()
```

12.3.2 Control de flujo

Se utiliza *if* y *switch* para hacer condicionales, y *for-in*, *for*, *while*, y *do-while* para hacer bucles.

Los paréntesis alrededor de la condición y el bucle son opcionales. Las llaves alrededor del cuerpo son

obligatorias.

```
let puntuaciones = [75,43,103,87,12]
var equipo = 0
for puntuacion in puntuaciones {
    if puntuacion > 50 {
        equipo += 3
    }else{
        equipo += 1
    }

}
```

12.3.2.1 Estructura if

En un *if*, el condicional debe ser una expresión booleana, esto significa que *if score {...}* es un error, no una comparación implícita con cero.

Se puede usar *if* y *let* juntos para trabajar con valores que faltan. Estos valores están representados como opcionales. Un valor opcional contiene un valor o contiene *nil* para indicar que falta un valor. Se escribe "?" después del tipo de valor para mostrar como el valor es opcional.

```
var nombreOpcional: String? = "Juan Perez"
var saludo = "Hola!"
if let nombre = nombreOpcional {
    saludo = "Hola, \(nombre)"

}
```

Si el valor opcional es *nil*, la condición es *false* y el código entre llaves se omite. De lo contrario, el valor opcional es asignado a la constante después de let, lo que hace que el valor esté disponible en ese bloque de código.

12.3.2.2 Estructura switch

Los *switch* soportan cualquier tipo de dato y una gran variedad de operadores de comparación, no están limitados a *Int*.

```
let tiempo = "lluvia"
switch tiempo {
  case "soleado":
      let comentarioTiempo = "Hace un día esplendido."
  case "nublado", "lluvia":
      let comentarioTiempo = "Puede que llueva durante el día.."
  case "tormenta":
      let comentarioTiempo = "Mejor no salir a la calle."
  default:
      let comentarioTiempo = "Parece que hace buen día."

}
```

Después de ejecutar el código en la sentencia que identifica el programa, sale de la declaración de *switch*. La ejecución no continua con el siguiente caso, así que no hay necesidad de colocar el *break* después de cada case.

12.3.2.3 Estructura for-in

Se usa *for-in* para iterar sobre elementos de un diccionario.

```
let numerosInteresantes = [
    "Primos": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Cuadrados": [1, 4, 9, 16, 25],
]
var masLargo = 0
for (tipo, numeros) in numerosInteresantes {
    for numero in numeros {
        if numero > masLargo {
            masLargo = numero
        }
    }
}
```

Puede tener un índice en un bucle mediante la utilización de "…" para hacer un rango que incluya ambos valores o escribiéndolo explícitamente, con inicialización, condición e incremento.

Estos dos bucles hacen lo mismo:

```
var primerBucle = 0
for i in 0...2 {
    primerBucle += i
}
```

12.3.2.4 Estructura while

Se usa *while* para repetir un bloque de código hasta que una condición cambie, la condición puede estar al final para asegurar que el bucle corra completo al menos una vez.

```
var n = 2
while n < 100 {
    n = n * 2
}
```

12.3.3 Funciones y closures

12.3.3.1 Funciones

Usa *func* para declarar una función. Llama una función poniendo su nombre con una lista de argumentos entre paréntesis. Usa -> para separar el nombre de los parámetros y los tipos que retorna la función.

```
func saludo(nombre: String, dia: String) -> String {
    return "Hola \(nombre), hoy es \(dia)."
}
saludo("Roberto", dia: "Martes")
```

Usa una tupla para devolver los valores de la función.

```
func getPrecioGas() -> (Double, Double, Double) {
    return (3.59, 3.69, 3.79)
}
getPrecioGas()
```

Las funciones pueden recibir un número variable de argumentos, organizándolos en un *array*.

```
func sumaDe(numeros: Int...) -> Int {
    var suma = 0
    for numero in numeros {
        suma += numero
    }
    return suma
}
sumaDe()                // = 0
sumaDe(42, 597, 12)     // = 651
```

12.3.3.2 Closures

Los closures son un tipo de función especial. Se puede usar como el argumento de métodos y funciones, el valor devuelto por métodos y funciones o se puede asignar a una variable que se usara como función.

Estas sentencias se ejecutan en el ámbito donde se creó y pueden acceder a una copia de las variables, funciones, etc. de ese ámbito.

Se pueden escribir *closures* sin un nombre rodeado de llaves ({ }). Se usa *in* para separar los argumentos y el tipo de retorno del cuerpo.

```
numeros.map({
    (numero: Int) -> Int in
    let resultado = 3 * numero
    return resultado
})
```

12.3.4 Objetos y clases

Se utiliza *class* seguido del nombre de la clase para declarar una clase. La declaración de una propiedad en una clase se escribe de la misma forma que una variable o constante, con la diferencia de estar en el contenido o

contexto de la clase. Los métodos y las funciones se escriben de la misma forma.

```
class Forma {
    var numeroDeLados = 0
    func descripcionSimple() -> String {
        return "Una forma con \(numeroDeLados) lados."
    }
}
```

Se crea una instancia de la clase poniéndole paréntesis después del nombre. Usa la sintaxis de punto para acceder a los métodos y propiedades de la instancia.

```
var forma = Forma()
forma.numeroDeLados = 7
var descripcionForma = forma.descripcionSimple()
```

A esta versión de la clase *Forma* le falta algo importante, un inicializador para configurar la instancia cuando es creada. Se usa *init* para crear uno.

```
class NombreForma {
    var numeroDeLados: Int = 0
    var nombre: String

    init(nombre: String) {
        self.nombre = nombre
    }
    func descripcionSimple() -> String {
        return "Una forma con \(numeroDeLados) lados."
    }
}
```

Se usa *self* para diferenciar entre el *nombre* de la propiedad y el *nombre* del argumento. Los argumentos se le pasan al inicializador como en una función cuando creas una instancia de la clase. Cada propiedad necesita que se le asigne un valor, ya sea en la declaración o en el inicializador.

Los métodos de la subclase que sobrescribe la implementación de la superclase están marcados con *override*. Sobrescribir un método por accidente, sin poner *override*, es detectado por el compilador como un error. El compilador también detecta cuando se usa la palabra *override* sin sobrescribir un método de su superclase.

12.3.5 Enumeración y estructuras

12.3.5.1 Enumeración

Usa *enum* para crear una enumeración. Como pasa con clases y los otros tipos, las enumeraciones pueden tener métodos asociados a ellas.

```
enum Valor: Int {
    case As = 1
    case Dos, Tres, Cuatro, Cinco, Seis, Siete, Ocho, Nueve, Diez
```

```
    case Sota, Reina, Rey
    func descripcionSimple() -> String {
        switch self {
        case .As:
            return "as"
        case .Sota:
            return "sota"
        case .Reina:
            return "reina"
        case .Rey:
            return "rey"
        default:
            return String(self.rawValue)
        }
    }
}
let As = Valor.As
let asRawValue = As.rawValue
```

En el ejemplo, el tipo de valor de la enumeración es *Int*, así que solo hay que especificar el primer valor simple. El resto de los valores es asignado por orden. Puedes usar *String* o *Float* como tipo de valor en una enumeración.

Usa la propiedad *rawValue* para obtener un valor de la enumeración e *init(rawValue:)* para convertir entre valores simples y valores de la enumeración.

```
if let carta = Valor(rawValue: 3) {
    let descripcionTres = carta.descripcionSimple()
}
```

12.3.5.2 Estructuras

Usa *struct* para crear una estructura. Las estructuras se comportan de forma similar a las clases, incluyendo mértodos e inicializadores. Una de las diferencias más importantes entre clases y estructuras, es que la estructura siempre se copia cuando se pasa de un lado a otro en el código, pero las clases se pasan por referencia.

```
struct Carta {
    var valor: Valor
    var palo: Palo
    func descripcionSimple() -> String {
        return "El \(valor.descripcionSimple()) de \(palo.descripcionSimple())"
    }
}
let tresDeEspadas = Carta(valor: .Tres, palo: .Espadas)
let tresDeEspadasDescripcion = tresDeEspadas.descripcionSimple()
```

12.3.6 Protocolos y extensiones

12.3.6.1 Protocolos

Se usa protocol para declarar un protocolo.

```
protocol EjemploProtocolo {
    var descripcionSimple: String { get }
    mutating func ajustar()
}
```

Las clases, enumeraciones y las estructuras pueden adoptar protocolos.

```
class ClaseSimple: EjemploProtocolo {
    var descripcionSimple: String = "Una clase muy simple."
    var otraPropiedad: Int = 69105
    func ajustar() {
        descripcionSimple += "  Ahora ajustado al 100%."
    }
}
var a = ClaseSimple()
a.ajustar()
let aDescripcion = a.descripcionSimple

struct EstructuraSimple: EjemploProtocolo {
    var descripcionSimple: String = "Una estructura simple"
    mutating func ajustar() {
        descripcionSimple += " (ajustado)"
    }
}
var b = EstructuraSimple()
b.ajustar()
let bDescripcion = b.descripcionSimple
```

Se observa el uso de la palabra *mutating* en la declaración de *EstructuraSimple* para marcar un método que modifica la estructura. La declaración de *ClaseSimple* no necesita ningún método marcado como *mutating*, debido a que los métodos de una clase pueden siempre modificar la clase.

12.3.6.2 Extensiones

Se usa *extension* para añadir funcionalidad a un tipo existente, como nuevos métodos y propiedades calculadas. Se puede usar una extensión para añadir protocolos a un tipo que fue declarado en otra ocasión, o que se importase desde una librería.

```
extension Int: EjemploProtocolo {
    var descripcionSimple: String {
        return "El numero \(self)"
    }
    mutating func ajustar() {
        self += 42
```

```
    }
}
7.descripcionSimple
```

12.3.7 Genéricos

Se escribe un nombre dentro de las llaves angulares para crear una función genérica.

```
func repetir<ItemType>(item: ItemType, veces: Int) -> [ItemType] {
    var resultado = [ItemType]()
    for _ in 0...veces {
        resultado.append(item)
    }
    return resultado
}
repetir("toc",veces: 4)
```

Puedes hacer formas genéricas de funciones o métodos, también de clases, enumeraciones y estructuras.

```
// Reimplementar el tipo genérico opcional de Swift
enum ValorOpcional<T> {
    case Ninguno
    case Alguno(T)
}
var posibleInt: ValorOpcional<Int> = .Ninguno
posibleInt = .Alguno(100)
```

12.3.8 Opcionales

Se utilizan opcionales en situaciones en las que el valor puede estar ausente.

Los opcionales de Swift permiten indicar la ausencia de un valor de cualquier tipo en absoluto, sin la necesidad de constantes especiales.

Se establece una variable opcional a un estado sin valor, asignándole el valor *nil*, que es la ausencia de un valor de un cierto tipo:

```
var codigoRespuestaServidor: Int? = 404
// codigoRespuestaServidor contiene un valor real Int de 404:
codigoRespuestaServidor = nil
// codigoRespuestaServidor ahora no contiene ningún valor
```

Si se define una constante o variable opcional sin proporcionar un valor por defecto, la constante o variable se ajusta automáticamente a *nil.*