UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Aplicación gráfica para dispositivos iOS para creación/edición de patches del lenguaje de programación orientado a flujo denominado 'Pure Data'

Proyecto Final de Carrera

Ingeniero Técnico en Informática de Sistemas

Autor: Pablo Díaz Barrón

Director: Joaquín Gracia Morán

Codirector: Juan Carlos Ruiz García

13 de diciembre de 2012

Documento creado con LaTeX empleando el paquete `hyperref`, que permite navegar por el documento PDF pulsando sobre el índice, referencias a figuras y páginas, marcas de pie de página, palabras y siglas del glosario, enlaces de internet, etc.

Resumen

Pure Data es un entorno de programación gráfico a tiempo real para procesamiento de audio (*vanilla*), video y gráficos (versión *extended*). Es a tiempo real porque no requiere ni compilar ni ejecutar el programa: la *ejecución / edición / modificación / programación* puede realizarse simultáneamente.

Este lenguaje es una rama de la familia de lenguajes de programación de *patches* conocida como *MAX* (ver `cycling74.com`) desarrollado originalmente por *Miller Puckette*.

Es un software gratuito multiplataforma: *linux*, *windows*, *mac*. También puede descargarse el código fuente. El proyecto *libpd* consiste en una librería para poder acceder al *core* de *Pure Data* desde dispositivos móviles. Sin embargo, aún no existe una versión de *Pure Data* para dispositivos móviles con una GUI[1] que permita crear y editar *patches* (sólo es posible cargar y reproducir los *patches* y poco más), por lo que sería necesario modificar el *core* de *Pure Data* para adecuarlo a este propósito.

Esta ha sido la motivación para realizar el proyecto: utilizar pantallas táctiles, ya que éstas son muy adecuadas para controlar este tipo de lenguajes de programación gráfica. Debido a la alta interactividad que requieren, el uso de dispositivos como el ratón pueden llegar a resultar incómodos si se trabaja intensivamente (se hace un uso intensivo de clics de ratón).

Un *patch* es el equivalente al código de programación en los lenguajes de programación normales, salvo que en este caso la programación se realiza creando objetos representados visualmente por cajas con puertos de entrada/salida, e interconectando éstos mediante cables. Se puede ver un ejemplo en la figura 1.

Los *patches* de *Pure Data* están formados básicamente por cuatro elementos:

Objetos normales: tienen forma cuadrada, con el nombre del objeto y puede que algún argumento. Además disponen de *puertos* de *entrada/salida* para interconectarlos. Permiten realizar funciones de programación.

Objetos de entrada/salida de datos: permiten introducir valores y/o visualizar resultados: *message*, *number*, *symbol*, *comment*, *array*...

Objetos de control (GUI): son objetos que permiten interactuar con el *patch*, como por ejemplo mediante un botón, deslizadores para cambiar valores, interruptores... Sirven para crear una GUI de control asociada a un *patch*.

Cables de conexión: permiten interconectar los elementos anteriores para crear el flujo de datos del programa. Hay dos tipos de cables: de *señal* y de *control*. Los cables de *señal* transmiten la información del audio, y los cables de *control* transmiten datos (numéricos, mensajes...) normalmente para controlar e influir sobre las *señales* de audio.

[1] *Graphical User Interface*, Interfaz gráfica de usuario

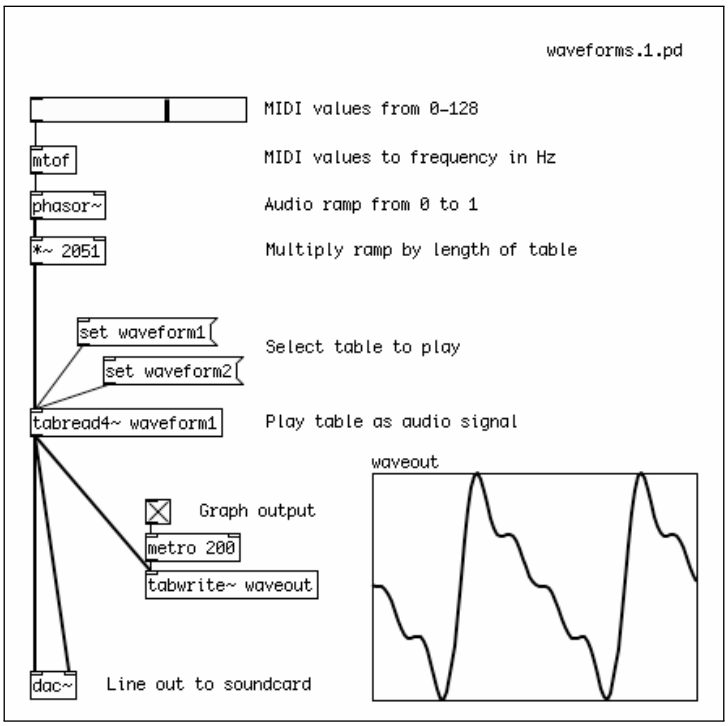

Figura 1: Ejemplo de un *patch* típico donde pueden apreciarse los diferentes tipos de elementos que lo componen.

Dentro de los objetos normales hay que mencionar que existen los denominados *subpatches* y *abstractions*, que son objetos que contienen a su vez subobjetos y sirven para organizar mejor las vistas, para usar un *patch* externo en otro *subpatch* y para crear una interfaz de usuario libre de cables y de objetos normales. En la figura 2 podemos ver un ejemplo.

La aplicación para dispositivos *iOS* estará basada en la versión *vanilla*, por lo que sólo tendrá capacidad de procesar audio. Permite crear, editar, reproducir y compartir *patches*. La interfaz deberá estar muy optimizada para obtener la mejor ergonomía de uso. Se revisarán y mejorarán las características de la versión de escritorio.

Al abrirse el programa se mostrará una lista con los *patches* del usuario almacenados localmente. El usuario puede seleccionar uno de los *patches* existentes para su edición o bien crear uno nuevo. Al seleccionar/crear un *patch*, se abrirá la ventana de edición, desde la que pueden crearse objetos y realizar sus interconexiones. Esta vista de edición estará muy optimizada para su uso con pantallas táctiles.

Palabras clave: *patch*, *Pure Data*, GUI, *vanilla,* iOS

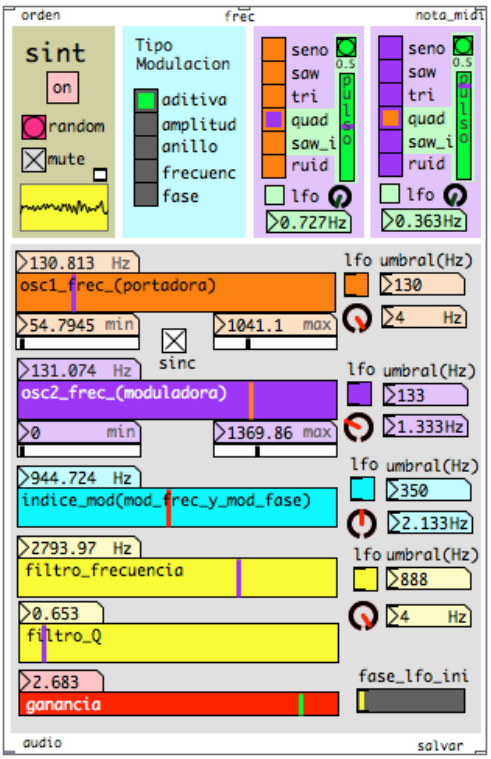

Figura 2: Ejemplo de *subpatch* con una GUI definida. Obsérvense los puertos de entrada y salida en la parte superior e inferior del recuadro.

Índice general

Parte I

Tecnologías de Apple

1. Acerca de *Apple*

Figura 1.1: El logotipo de *Apple*.

Es bien conocida la historia y trayectoria de la que seguramente sea en este momento la empresa más famosa del mundo, por lo que no voy a contar su historia, sino que voy a resaltar el hecho que para mi ha sido uno de los más importantes de su éxito.

Apple ofrece productos informáticos integrados donde tanto el hardware como el software han sido diseñados por ellos mismos. Aún creando productos cerrados con sistemas operativos cerrados, ha sabido conjugar muy bien el hardware y el software que crea, optimizándolo mucho para que la experiencia del usuario sea la mejor posible. Eso es de vital importancia en un entorno táctil como son sus dispositivos *iOS*. El hecho de carecer de sensaciones táctiles como las que se tienen al pulsar las teclas de un teclado junto con la necesidad de crear aparatos móviles que estén siempre disponibles y listos para que los usemos, normalmente por breves espacios de tiempo, sin preocuparnos de si responden lento, mal o simplemente se cuelgan y no responden, hace que esa fluidez que *Apple* proporciona a sus productos sea una de las claves importantes de su éxito.

Básicamente los productos informáticos de *Apple* se clasifican en dos tipos de dispositivos:

- Ordenadores personales, cuyo sistema operativo es *Mac OSX*.
- Dispositivos móviles táctiles, que funcionan con el sistema operativo denominado *iOS*.

Todos los dispositivos móviles tienen botones de control similares (ver figura 1.2):

Botón Home Es el botón redondo situado debajo de la pantalla. Principalmente sirve para volver al menú de iconos de las *apps*, aunque puede configurarse para realizar otras funciones según el número de pulsaciones que se realicen.

Botón de reposo Sirve para encender/apagar el dispositivo y para ponerlo en reposo o desbloquearlo.

Interruptor de silencio Pone el dispositivo en modo silencio.

Controles de volumen Sirven para regular el volumen del audio del dispositivo.

Figura 1.2: Esquema de los controles de un dispositivo *iOS*.

2. El entorno *Cocoa*

Cocoa es un entorno de aplicación para los sistemas operativos de *Apple Mac OSX* e *iOS*, llamándose en este último caso *Cocoa Touch*, similar a otros entornos como *Java* o *.NET* de *Microsoft*. *Cocoa* es un conjunto de *frameworks* orientados a objetos que proporcionan un entorno de ejecución para aplicaciones que funcionan bajo *Mac OSX* e *iOS*. Muchos de los *frameworks* son comunes en ambos sistemas operativos, lo que facilita mucho portar aplicaciones de un sistema a otro.

La mayoría de aplicaciones que se incluyen en *Mac OSX* e *iOS*, como *Mail* o *Safari*, son aplicaciones *Cocoa*. Las aplicaciones *Cocoa* presentan una interfaz de usuario muy integrada con el resto de componentes del sistema operativo, ya que *Cocoa* automatiza muchos de los aspectos de una aplicación para que cumplan con las HIG[1] propias de *Apple*. *Apple* presta un cuidado especial a sus HIG e incita a usarlas nombrándolas constantemente por toda su documentación. También posee un IDE[2] denominado *XCode*, que permite desarrollar aplicaciones tanto para *Mac OSX* como para *iOS*. El lenguaje de programación que emplea de forma preeminente *Cocoa* se denomina *Objective-C*, una capa sobre el lenguaje *C* que le añade funcionalidad orientada a objetos, aunque también pueden usarse otros lenguajes de programación con *Cocoa*.

El término *Cocoa* sirve para referirse de forma general al conjunto que engloba las tecnologías de software de *Apple*. Aunque en *Mac OSX* pueden emplearse otros entornos de aplicación como *Carbon* o *Java*, en *iOS* únicamente está disponible el entorno *Cocoa Touch*, que está compuesto por los *frameworks* `UIKit` —de alto nivel— y `Foundation` —de bajo nivel—. Aunque en el desarrollo de una aplicación se empleen tecnologías o *frameworks* de terceras partes —p.ej. *Open GL* o *Cocos2D*—, la base de esas tecnología es *Cocoa Touch*. Por ejemplo, el contenido mostrado por una aplicación desarrollada en *Open GL*, en última instancia es mostrada en un objeto de la clase `UIView`, perteneciente al *framework* `UIKit` de *Cocoa Touch*.

Para conocer más acerca de *Cocoa* ver [**2**].

[1] Human Interface Guidelines, Directrices de interfaz humana

[2] *Integrated Development Environment*, Entorno de desarrollo integrado

3. El sistema *iOS*

iOS es el sistema operativo incorporado en los dispositivos móviles táctiles de *Apple iPhone*, *iPod touch* e *iPad*, aunque también se ha ampliado para adaptarse a su reproductor multimedia de salón denominado *Apple TV*. Está basado en el *núcleo* de *Mac OSX*, y por tanto, es un sistema *Unix*, aunque está optimizado para interactuar mediante gestos táctiles o de movimiento, por lo que comúnmente se describe como un sistema operativo guiado por eventos: cuando el usuario interactúa con el dispositivo, ya sea de forma táctil, por movimientos, por ubicación según el GPS, etc., cada uno de esos dispositivos hardware produce una serie de datos en bruto que se transfieren a los *frameworks* del sistema. Estos *frameworks* empaquetan los datos y los envían como *eventos* a las aplicaciones que han de procesarlos. Gestiona el hardware del dispositivo y proporciona las tecnologías requeridas para implementar aplicaciones nativas. Incorpora diversas *apps* de sistema como *Teléfono*, *Mail* y *Safari* que proporcionan al usuario servicios del sistema estándar. En el nivel más alto, *iOS* actúa como intermediario entre el hardware subyacente y las *apps* que aparecen en pantalla. Normalmente las *apps* no se comunican directamente con el hardware subyacente, sino que lo hacen a través de interfaces de sistema bien definidas que protege a las *apps* de los cambios en el hardware, quedando estructurada la arquitectura de *iOS* en las capas o niveles de abstracción mostradas en la figura 3.1:

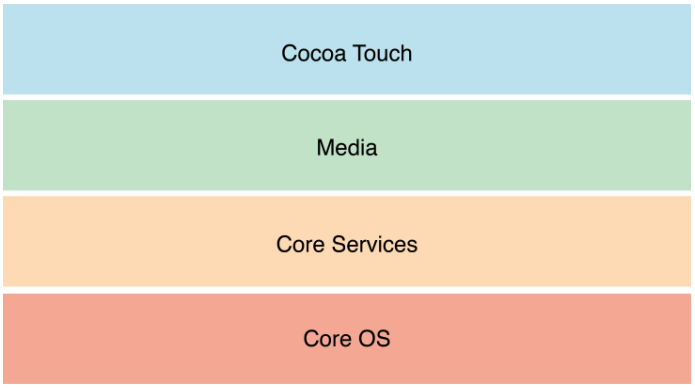

Figura 3.1: Las capas de *iOS*.

***Core OS (Nucleo del SO)*:** Este nivel contiene el núcleo del sistema operativo, el sistema de ficheros, la infraestructura de red, los mecanismos de seguridad, la gestión de la energía y diversos controladores (*drivers*) de dispositivo.

***Core Services (Servicios del Núcleo)*:** Los *frameworks* de este nivel proporcionan servicios del núcleo tales como manipulación de *strings*, gestión de estructuras de datos, gestión de red, utilidades de URLs, gestión de contactos y las preferencias de usuario. También proporciona servicios según las características de hardware del dispositivo, como el GPS, la brújula, acelerómetro y giroscopio.

***Media*:** Los *frameworks* y servicios de esta capa dependen de la capa *Core Services* y proporciona servicios para gráficos y multimedia a la capa *Cocoa Touch* que tiene por encima.

***Cocoa Touch*:** Los *frameworks* de esta capa soportan directamente las aplicaciones basadas en *iOS*.

Para más información acerca del sistema *iOS* ver [**3**].

4. Los *frameworks* de *Apple*

La mejor manera de comprender el concepto de *framework* es compararlo con el concepto de librería de programación clásica. Una librería no es más que un conjunto de métodos o funciones que proporcionan código comúnmente usado (como las librerías matemáticas o las que manejan caracteres de texto) que pueden incorporarse en programas. Para usar una librería, basta incorporar al programa una referencia a ella y usar sus funciones en los lugares del código que lo requieran. Sin embargo un *framework* no es sólo una librería o conjunto de librerías. Un *framework* es un esquema definido para el desarrollo de una aplicación, proporcionándole una estructura global que modela las relaciones generales de las entidades del dominio, imponiendo un diseño al programa y haciendo que sea más sencillo trabajar con tecnologías complejas. Para ello proporciona un conjunto de herramientas como librerías, programas, recursos, etc. La diferencia fundamental entre ambos conceptos es que para usar una librería se introduce la librería en tu código, mientras que para usar un *framework* lo que se hace es introducir tu código en el *framework* para personalizarlo.

Los *frameworks* de *Apple* están escritos principalmente en *Objective-C*, aunque los *frameworks* de más bajo nivel también están en *C*, y consisten en un directorio que incluye una librería compartida, los archivos de cabecera `.h` que proporcionan acceso al código almacenado en la librería, y otros recursos como archivos de imagen y sonido. La librería compartida define funciones y métodos que las *apps* pueden usar. *Mac OSX* e *iOS* comparten muchos *frameworks*, lo que facilita portar aplicaciones de un sistema a otro.

Cada *framework* pertenece a una de las capas del sistema *iOS* (ver figura 4.1). Cada capa está construida sobre las capas que tiene por debajo. Se recomienda usar siempre que sea posible los *frameworks* de más alto nivel en lugar de los *frameworks* de bajo nivel. Los *frameworks* de más alto nivel proporcionan abstracciones orientadas a objetos de las estructuras de bajo nivel. Sin embargo, si los de alto nivel no proporcionan la solución que buscamos, siempre podemos bajar a *frameworks* de bajo nivel.

Las librerías de clases más importantes de *Cocoa* están empaquetadas en los dos *frameworks* principales que usan las *apps* de *iOS* (de hecho se añaden por defecto al crear un nuevo proyecto):

`UIKit:` Es un *framework* de alto nivel que proporciona los objetos que las aplicaciones muestran en su interfaz de usuario (**`UIView`**, **`UIButton`**, **`UITextField`**...),

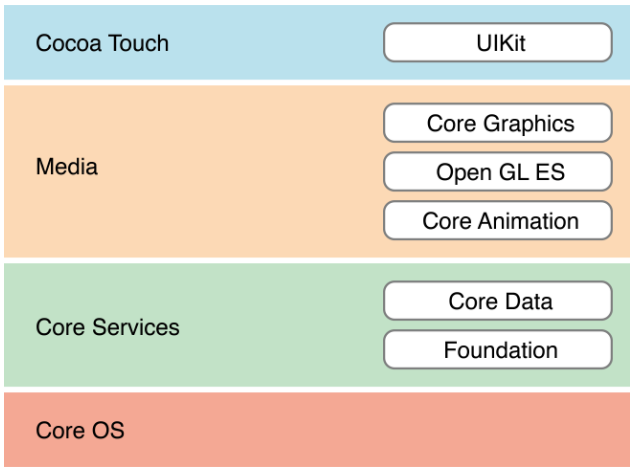

Figura 4.1: Algunos *frameworks* en sus correspondientes capas del sistema *iOS*.

además de definir la estructura del comportamiento de la aplicación, incluyendo la gestión de eventos y el redibujado de los gráficos.

`Foundation:` Es un *framework* de bajo nivel que define el comportamiento básico de los objetos, establece mecanismos para su gestión, y proporciona objetos para tipos de datos primitivos (**`NSString`**, **`NSInteger`**...) y para estructuras de datos básicas (**`NSArray`**, **`NSSet`**, **`NSDictionary`**...). Proporciona la clase básica de la que heredan las clases, denominada **`NSObject`**. Todas las clases poseen como superclase raíz a **`NSObject`**, y es la clase que debe usar el desarrollador para crear objetos básicos.

5. La documentación de *Apple*

La documentación de *Apple* específica de *iOS* está disponible en internet en `developer.apple.com/library/ios`, o bien desde *XCode* en el panel *Organizer*. Además en el panel *Help Inspector* descrito en la sección 8.3 se muestra información del elemento seleccionado.

La forma de trabajar que se recomienda a la hora de incorporar una nueva tecnología a una aplicación es primero consultar la documentación general de dicha tecnología que se suministra en forma de guías (están disponibles desde la página principal de la documentación de *Apple*, en el apartado *Guides* de la sección *Resource Types* en el panel lateral de la izquierda), y luego leer la documentación específica de las clases y métodos concretos del *framework* o *frameworks* que implementan dicha tecnología.

Sin embargo esas guías describen la tecnología de forma muy general, sin dar demasiados detalles de la implementación, por lo que se echa en falta más documentación en forma de tutoriales guiados por pasos. Por suerte la red está plagada de libros y tutoriales generalistas muy buenos, como por ejemplo los de `raywenderlich.com` o los de `techotopia.com`. Sin embargo, la documentación de *Apple* contiene numerosas aplicaciones de muestra que pueden abrirse en *XCode*, ejecutarse y analizar su código.

Para dudas muy concretas me ha brindado una inestimable ayuda la página `stackoverflow.com`, una comunidad muy activa de desarrolladores de todo tipo donde se realizan consultas y se proponen soluciones muy específicas. La inmensa mayoría de dudas y problemas que surgen a la hora de desarrollar aplicaciones en cualquier lenguaje seguramente esté en esa página, donde además se proponen muchas soluciones a un mismo problema.

Para consultar el listado de los documentos básicos de desarrollo de *iOS* ver la sección de Bibliografía.

6. Herramientas empleadas

La mejor manera de desarrollar aplicaciones para *iOS* y para *Mac OSX* es empleando las propias herramientas de software y dispositivos de *Apple*. En este proyecto se han empleado las siguientes herramientas:

Hardware:

Dispositivo *iOS* de prueba: iPhone 4 16Gb con *iOS* 5.0

Equipo de desarrollo: *Apple* MacbookPro 13" mediados 2010, 2.4 GHz Code 2 Duo, 8Gb RAM

Software:

Entorno de desarrollo: *XCode* 4 última versión, aplicando actualizaciones durante el desarrollo.

Sistema operativo de desarrollo: OSX 10.8 Mountain Lion actualizado durante el desarrollo.

7. Los patrones de diseño de software

Los patrones de diseño son un concepto abstracto que trata de resolver problemas de software recurrentes. Al adoptar un patrón de diseño a un problema concreto de nuestro desarrollo se está imponiendo una estructura bien definida de clases y métodos que es bien sabido que resuelve ese problema concreto de la mejor forma posible. Esto no significa que un patrón de diseño sea un *frameworks*, sino que los *frameworks* suelen hacer uso de muchos patrones de diseño en su implementación. Los patrones no son código ni están ligados a un lenguaje de programación de objetos. Pretenden acumular las buenas técnicas de desarrollo a partir de la sabiduría colectiva de los desarrolladores, de igual forma que un arquitecto aplica una serie de técnicas bien conocidas a la hora de realizar cálculos de estructuras.

Su principal motivación es hacer que los programas sean más reutilizables, más legibles y mejor mantenibles.

Apple ha adoptado ampliamente los patrones de diseño en sus *frameworks* y los nombra constantemente en su documentación. Por tanto, muchas veces estaremos obligados a emplear esos patrones y por lo tanto conviene conocer los más comúnmente usados, siendo los dos más importantes el patrón *Modelo Vista Controlador* y el patrón *Delegación*.

Para conocer más detalles acerca de los patrones de diseño de *Cocoa* ver los documentos [**2**] y [**10**].

7.1. El patrón MVC

El patrón *Modelo Vista Controlador* es el patrón más importante en el desarrollo de aplicaciones *Cocoa* ya que los *frameworks* de *Apple* lo usan de forma generalizada. Es un patrón de alto nivel que abarca toda la arquitectura de una aplicación clasificando los objetos según el rol general que desempeñan en la aplicación. Además es un patrón compuesto por otros más elementales.

El patrón MVC[1] clasifica los objetos en tres grandes grupos: objetos *modelo*, objetos *vista* y objetos *controlador*. El patrón MVC define los roles que desempeñan esos tipos de objetos en la aplicación y cómo se comunican. Cada uno de los tres tipos de objetos está separado del resto por límites abstractos, realizándose la

[1] *Model View Controller*, Modelo Vista Controlador

comunicación entre los objetos a través de dichos límites.

Los programas orientados a objetos pueden beneficiarse de varias formas si adaptan el patrón MVC. La mayoría de objetos en esos programas suelen ser más reusables y sus interfaces tienden a estar mejor definidas. Además esos programas son más adaptables a cambios en los requisitos, es decir, son más fácilmente extensibles que los programas que no están basados en el patrón MVC.

Apple emplea este patrón en muchas de sus tecnologías y arquitecturas, por lo que el desarrollador se verá obligado a emplear este patrón en muchos aspectos del desarrollo de sus aplicaciones.

La figura 12.1 de la página 39 muestra el esquema del patrón aplicado a las *apps iOS*.

Los roles que desempeñan los objetos de cada tipo son:

- **Objetos *Modelo***: contienen los datos de la aplicación y la lógica que manipula esos datos. Normalmente no tienen conexiones explícitas con la interfaz de usuario empleada para presentar y editar los datos, sino que son los objetos *Controlador* los encargados de unir los datos con la interfaz. Se recomienda que residan en objetos *Modelo* los datos que formen parte de la capa de persistencia de la aplicación (es decir, el mecanismo por el que los datos persisten más allá de su tiempo de ejecución, por ejemplo almacenándolos en ficheros, en bases de datos, en la nube, etc.), aunque se permite cierta flexibilidad.

- **Objetos *Vista***: se encargan de presentar la información al usuario. Tienden a ser reusables y configurables, y proporcionan consistencia entre aplicaciones. En *Cocoa Touch*, el *framework* que contiene las clases que generan objetos vista es el denominado `UIKit`. Las vistas deben garantizar que muestran correctamente los datos del modelo, por lo que necesitan mecanismos para ser informadas de los cambios realizados en los datos de los objetos modelo.

- **Objetos *Controlador***: se encargan de enlazar el Modelo con la Vista, actuando como intermediario entre la comunicación que se realiza entre los objetos de la capa de Modelo y los objetos de la capa de Vista. Normalmente son los responsables de garantizar que las vistas tengan acceso a los datos que necesiten mostrar de los objetos Modelo y actúan como conducto a través del que las vistas obtienen información de los cambios en el modelo. A este grupo pertenecen los denominados *View Controllers*, elementos clave en el desarrollo de *apps* (para más detalles ver la sección 10 en la página 32).

7.2. El patrón *Delegación*

Un *delegado* es un objeto que actúa en nombre de, o en coordinación con, otro objeto cuando este último objeto encuentra un evento en un programa. Es decir, un objeto, en lugar de realizar pos sí mismo cierta tarea, *delega* dicha responsabilidad en otro objeto denominado *delegado*. La ventaja es que reduce el acoplamiento entre el objeto original y el *delegado*. *Apple* emplea el patrón *Delegación* masivamente en sus

frameworks. Permite crear objetos con un comportamiento genérico, como puede ser un campo de texto (`UITextField`) o una tabla (`UITableView`), a los que, haciendo uso del mecanismo de *delegación*, se les incorpora el comportamiento específico de una aplicación concreta. Por ejemplo, en el proceso de edición de un `UITextField`, la lógica de validación de los datos no puede ser general, sino que dependerá de cada aplicación concreta, o al seleccionar una celda en una tabla, lo que suceda dependerá de los requisitos de la aplicación.

En *Objective-C* el mecanismo de *delegación* está implementado de la siguiente manera: El objeto que *delega* posee una propiedad denominada `delegate` que referencia al objeto que implementa la lógica concreta. Además, debe definir lo que se denomina protocolo formal —un mecanismo similar a las *interfaces* de *Java*— donde únicamente se declaran los métodos que han de emplearse, pero la implementación es responsabilidad de otro objeto, como por ejemplo un *delegado*. Los objetos *delegados* (puede haber muchos) adoptan el *protocolo* definido anteriormente e implementan los métodos del *protocolo* con el código concreto de una aplicación dada. En un momento dado en la ejecución del programa, al objeto que *delega* se le asigna el *delegado* adecuado para la tarea concreta que se está ejecutando.

8. El IDE *XCode*

XCode es el *IDE* de *Apple* diseñado para desarrollar aplicaciones para *iOS* y *Mac OSX*. Funciona en el sistema operativo *Mac OSX* y puede descargarse gratuitamente. La versión a fecha de hoy es la **4.5.2**.

Antes de la versión **4**, *XCode* consistía en una serie de ventanas independientes donde realizar las diversas tareas de desarrollo. A partir de la versión **4** todas esas ventanas se han consolidado en una única ventana de aplicación compuesta por una barra de herramientas arriba, un editor en el centro, y una serie de *zonas* que lo rodean (ver figura 8.1). El editor posee funcionalidad de autocompletado —mostrando sugerencias a medida que se escribe— y puede configurarse para mostrar dos archivos a la vez (modo *Assistant*). Cada zona posee en la parte superior una barra para configurarla o para cambiar de panel.

Puede consultarse la guía de usuario de *XCode* en el documento [**6**].

Nota: tanto el IDE *XCode* como la documentación oficial de *Apple* actualmente sólo están disponibles en idioma inglés.

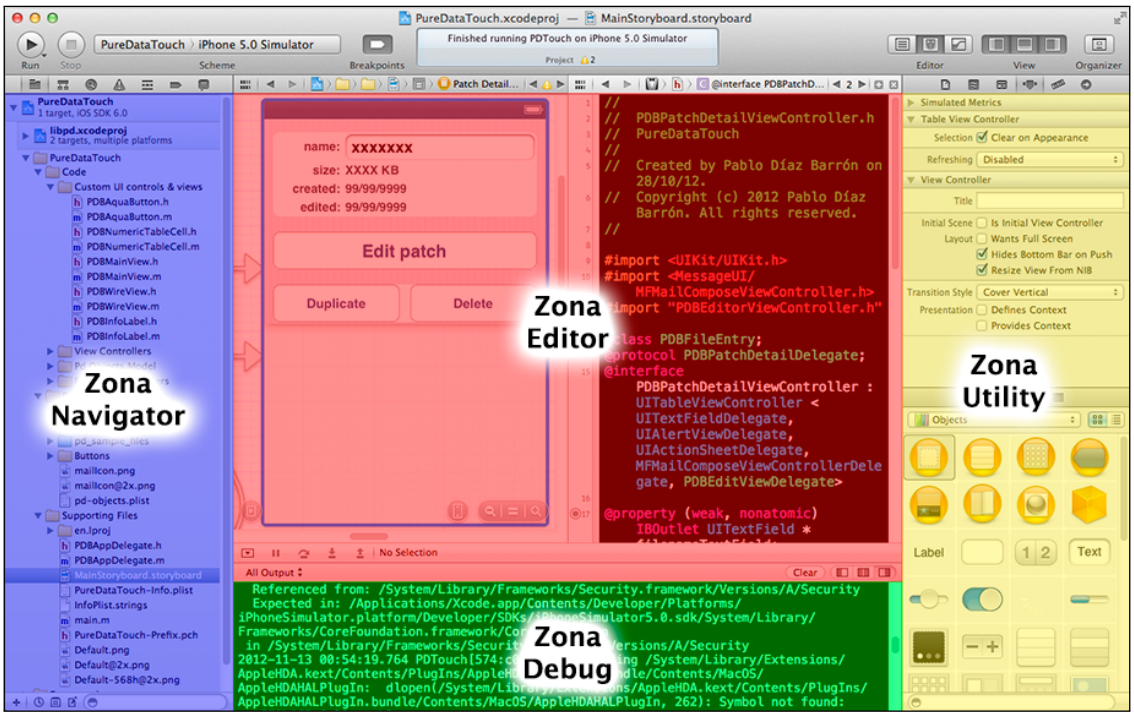

Figura 8.1: La ventana principal de *XCode* con las distintas vistas y paneles.

A continuación se describen los aspectos y paneles más importantes de *XCode*.

8.1. *Interface Builder* y *Storyboard*

Apple incluye en su IDE una aplicación para el diseño de interfaces gráficas denominado *Interface Builder*. Con esta aplicación es posible construir una aplicación con una GUI completamente funcional y navegable sin tener que escribir ni una sola línea de código, únicamente diseñando de forma gráfica. Obviamente esa aplicación no realizará nada útil hasta que no sea programada con código, pero ayuda a diseñar la interfaz general de la aplicación abstrayéndose del código. Un cambio importante introducido en el IDE *XCode* versión **4** es que el *Interface Builder* se ha integrado en la propia interfaz de *XCode*.

Otra novedad con respecto al *Interface Builder* es que si se desarrollan aplicaciones para *iOS* versión **5** o superior, puede emplearse lo que se denomina *Storyboard* (ver figura 8.2). Antes con el *Interface Builder* había que crear cada una de las vistas de la *app* que se estaba desarrollando de forma individual, guardando cada una de ellas en archivos `.nib` o `.xib` independientes, y luego enlazarlas todas ellas mediante código de programación. Sin embargo, el *Storyboard* es un único archivo que permite crear en una única ventana no sólo todas las vistas que componen la *app*, sino también las relaciones que hay entre ellas, es decir, el orden de navegación de la aplicación. Para ello, junto con el *Storyboard* se introduce el concepto de *segue* (Transición) que es un objeto que contiene la transición entre vistas y que en el *Storyboard* se muestra en forma de flecha que va desde la vista de origen a la vista de destino.

Si configuramos la vista del editor para que muestre dos archivos (modo *Assistant*), podremos ver los archivos de implementación de las vistas. En la parte superior de la segunda vista de edición que se abre, si se selecciona *Automatic*, se cargará automáticamente el archivo de código correspondiente a medida que se van seleccionando las distintas vistas contenidas en el *Storyboard*.

La principal ventaja de esta vista doble es que pueden crearse conexiones en el archivo de código de objetos visuales creados en el *Storyboard*. De esta forma se puede acceder a ellos programando y configurar sus comportamientos concretos con respecto a una aplicación concreta. Como se ha comentado en la sección dedicada a los patrones de diseño, un mecanismo ampliamente usado por *Apple* para implementar este comportamiento específico en objetos generales es mediante el uso del patrón *delegación*.

Para establecer relaciones entre las distintas vistas del *Storyboard* y para crear *outlets* hay que mantener pulsada la tecla *Ctrl* mientras se arrastra el elemento deseado al lugar de destino. Al soltar el ratón aparecerá un cuadro de diálogo con opciones para configurar la acción concreta que se desea realizar.

A la izquierda del *Storyboard* hay un panel denominado *Document Outline* que muestra en forma de tabla todas las vistas que contiene, así como las subvistas, controles y relaciones que las componen. Puede ocultarse/mostrarse el panel pulsando el círculo con la flecha situado abajo a la izquierda del *Storyboard*. La acción de

arrastrar con el ratón y la tecla *Ctrl* pulsada también puede realizarse desde este panel. De hecho suele ser más cómodo, sobre todo cuando la vista contiene varios subcontroles anidados y no está claro qué control se ha seleccionado.

Para conocer más detalles acerca del diseño de GUIs en *Cocoa* consultar el capítulo *Edit User Interfaces* de la guía de usuario de *XCode* disponible en [6].

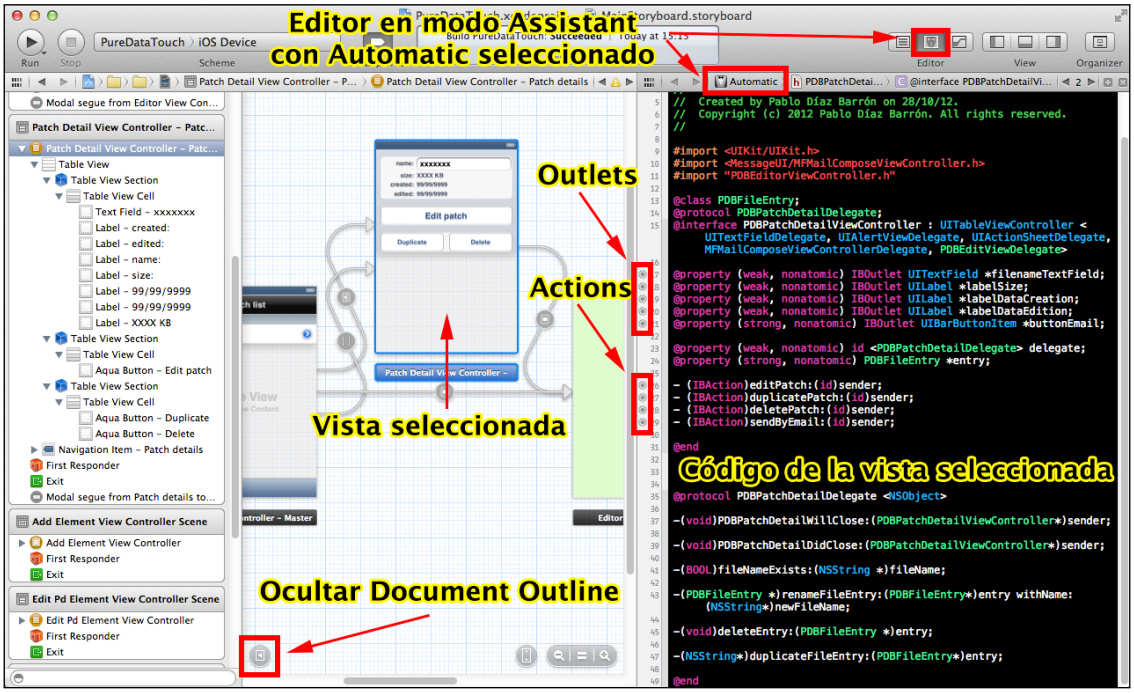

Figura 8.2: El *Storyboard* con el *Document Outline* a la izquierda en modo *Assistant*.

8.2. Zona Navigator

A la izquierda de la interfaz de *XCode* se encuentra la zona *Navigator* con una barra de herramientas en la parte superior para cambiar de panel. Los paneles más importantes son (ver fig. 8.3, pág. 22):

Project Navigator:
Muestra todos los archivos del proyecto. En función del archivo seleccionado la vista de edición cambiará. Por ejemplo, si se selecciona un archivo de código fuente, la vista de edición mostrará dicho código, si se selecciona un archivo de *Storyboard*, la vista de edición mostrará el *Interface Builder*, si se selecciona un archivo de imagen, se mostrará la imagen, etc.

El primer elemento de la jerarquía con el icono azul es el archivo del proyecto donde se configuran aspectos como el tipo de dispositivo *iOS* donde funcionará la *app*, el SDK[1] que usará, las orientaciones que soporta el dispositivo, los archivos de

[1] *Software Development Kit*, Kit de desarrollo de software

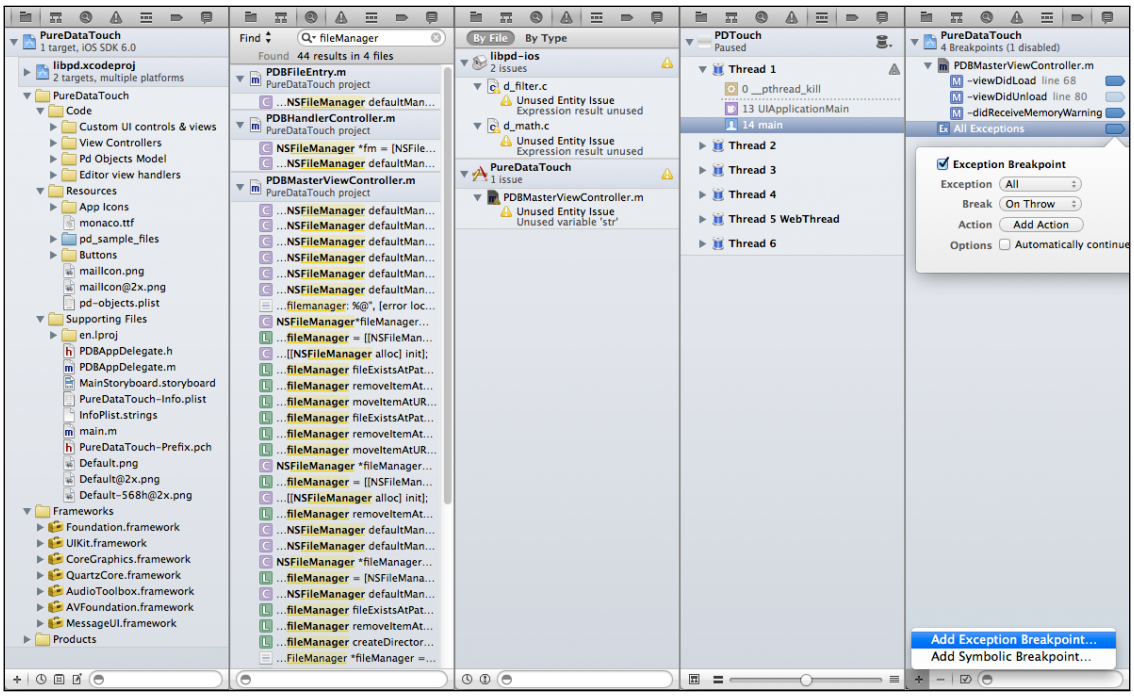

Figura 8.3: Los paneles de la zona *Navigator*. De izquierda a derecha: *Project Navigator*, *Search Navigator*, *Issue Navigator*, *Debug Navigator* y *Breakpoint Navigator*.

código fuente que se compilarán, permite añadir *frameworks* y posee muchas más opciones para configurar el compilador (ver fig. 8.4, pág. 23).

Search Navigator:
Permite realizar búsquedas en todo el proyecto.

Issue Navigator:
En este panel se muestran todos los avisos del compilador: errores, *warnings*, etc. Al seleccionar un problema, la vista del editor cargará el archivo conflictivo y mostrará la línea que ha producido el aviso.

Debug Navigator:
Muestra avisos de depuración. Especialmente importante cuando la aplicación falla por algún error.

Breakpoint Navigator:
Muestra los puntos de ruptura que se hayan añadido a la aplicación. Para añadir un punto de ruptura a una línea de código, en la vista de edición de código hay que hacer un clic de ratón a la izquierda de la línea, en el borde fuera de la vista de código.

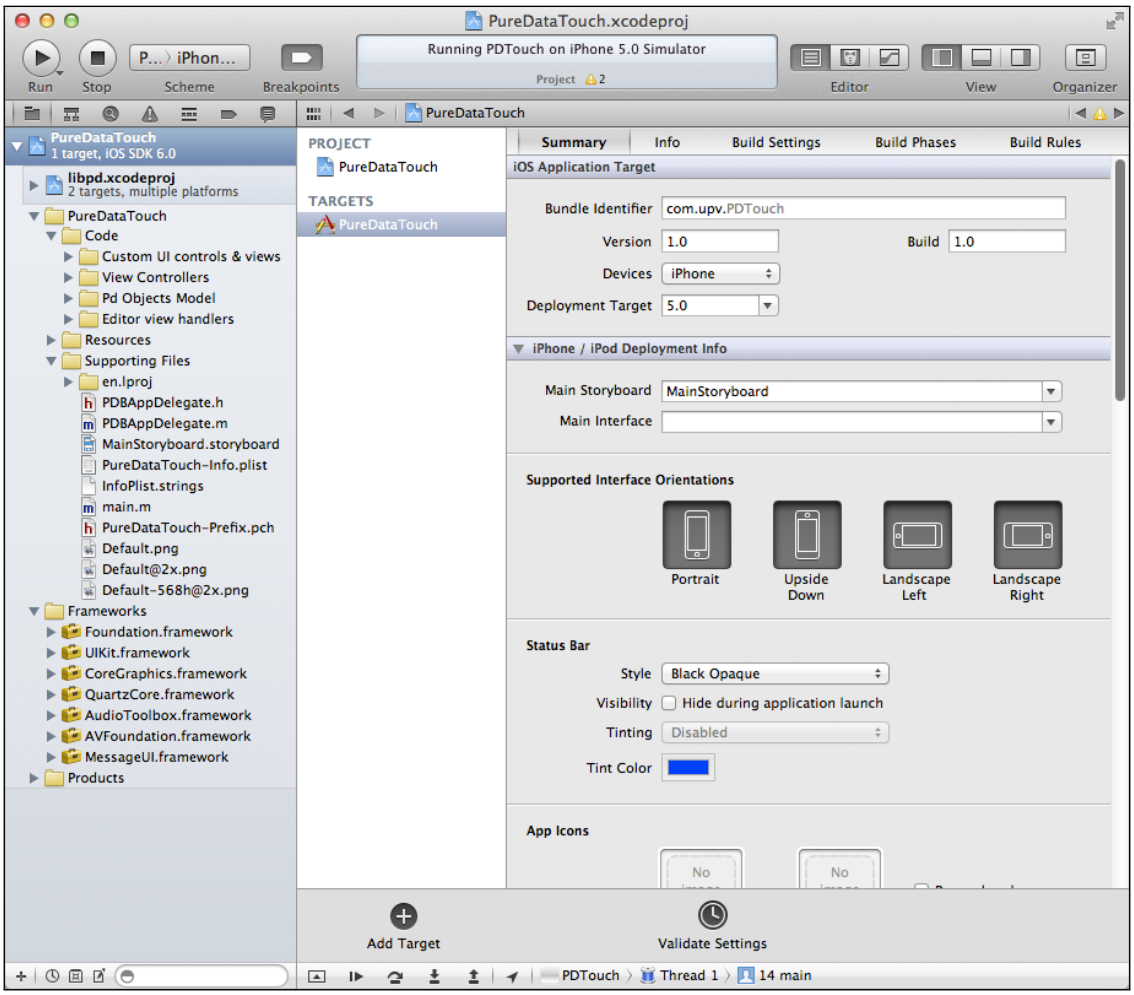

Figura 8.4: El archivo de configuración del proyecto seleccionado.

8.3. Zona Utility

Esta zona es especialmente importante con los *Storyboards* porque permite añadir y configurar objetos. En la parte inferior se muestra una tabla con los objetos *Cocoa* que pueden incorporarse al proyecto arrastrándolos al *Storyboard*. En la parte superior se muestra una serie de paneles en función del elemento que se está editando. La mayoría de paneles sólo se muestran si se está editando un *Storyboard*, donde los paneles más importantes son:

Quick Help: es especialmente útil en la vista de edición de código porque muestra un resumen de la documentación del elemento seleccionado (propiedad, método, clase, etc.) y posee enlaces a la documentación más detallada, que se abrirá en la ventana *Organizer* descrita en la sección 8.5.

Identity Inspector: La sección más importante de este panel es *Custom Class*, arriba del todo. En el campo *Class* puede seleccionarse el archivo de implementación concreto de nuestras subclases personalizadas de los elementos del

Interface Builder. Aquí es donde se especifica la clase *View Controller* concreta asignada a cada vista del *Interface Builder*.

Attributes Inspector: como su nombre indica, muestra los atributos del elemento actualmente seleccionado en el *Interface Builder* para poder configurarlos.

Size Inspector: permite editar los aspectos gráficos del elemento seleccionado como por ejemplo su posición en la pantalla, dimensiones, etc. También permite ajustar el elemento para que se autoredimensione cuando la interfaz cambia (por ejemplo, para adaptarla a los cambios de orientación del dispositivo).

Connections Inspector: muestra la relación de conexiones del elemento seleccionado y permite editarlas. Si se selecciona un *View Controller*, se muestran también todas las conexiones de los objetos que contiene.

8.4. Zona Debug y depuración

Muestra información de depuración y la consola de salida. Para enviar mensajes a la consola se utiliza el comando `NSLog`.

8.5. Ventana *Organizer*

Las secciones más importantes de esta ventana son las siguientes:

Devices Muestra información detallada acerca del dispositivo *iOS* real conectado al ordenador. Se recomienda consultar esta sección cuando ha habido algún error al conectar un dispositivo.

Projects Muestra los proyectos que han sido abiertos o creados con *XCode*. Cabe destacar la opción *Derived Data* que contiene la ubicación de archivos especiales que se generan en tiempo de ejecución (índices, reportes, información de salida, etc.), ya que esos archivos no residen en la carpeta donde se encuentran los archivos del proyecto.

Documentation Muestra la documentación de *Apple* y permite realizar búsquedas y añadir favoritos. Si la documentación no está instalada de forma local, se accederá a ella de forma remota a través de Internet, lo que puede resultar lento. Por ello es recomendable instalar la documentación localmente en el ordenador. Se instala accediendo a *XCode* > *Preferences* > *Downloads* > *Documentation* desde el menú principal de *XCode*. También puede accederse a la documentación del elemento seleccionado en *XCode* desde el panel *Quick Help* en la zona *Utility*, tal y como se describe en la sección 8.3.

8.6. Compilación

Para compilar el proyecto y mostrar los *warnings* y errores en el panel *Issue Navigator* hay que pulsar la combinación de teclas *Cmd+B* o seleccionar el comando *Product > Build* en el menú principal. Este es el modo de compilación básica. Sin embargo, es recomendable emplear el modo *Analyze* (combinación de teclas *mayus+cmd+B*). Además de compilar, busca *bugs*[2] potenciales en la lógica, gestión de la memoria de los objetos, y en las APIs[3] de *Apple* que pueden producirse por no seguir las políticas requeridas por los *frameworks* usados en la aplicación.

8.7. Ejecución

Para ejecutar una aplicación desde *XCode* primero hay que seleccionar el dispositivo deseado en la barra de herramientas y luego pulsar el botón de *Play* (el círculo con la flecha negra) o la combinación de teclas *cmd+R*.

XCode dispone de un dispositivo virtual que se denomina *iOS Simulator* en forma de aplicación externa. Se abre automáticamente si se ha seleccionado como dispositivo de prueba en la barra de herramientas. Como su nombre indica, esta aplicación simula los dispositivos táctiles *iOS* de *Apple* como iPhone o iPad, clasificados en función de sus dimensiones y resolución de pantalla (menú principal de *IOS Simulator > Hardware > Dispositivo*): *iPad*, *iPad (Retina)*, *iPhone (3.5")* que representa a los *iPhone 3GS* y anteriores, *iPhone (3.5" Retina)* que representa al *iPhone 4* y *4s*, *iPhone (4" Retina)* que representa al *iPhone 5*.

Como es un simulador, no emula el hardware de los dispositivos *iOS* reales, sino que ejecuta un *iOS* adaptado a la arquitectura *X86* de los ordenadores personales. Esto hace que sea una aplicación muy fluida, siendo muy cómodo trabajar con ella. Sin embargo esa fluidez no se corresponde con el rendimiento real de la aplicación en el dispositivo físico, ya que normalmente se ejecuta más fluido en el simulador. Además el simulador no permite realizar todos los gestos (no se pueden usar más de dos dedos; se simulan dos dedos manteniendo pulsada la tecla *Alt*; los *dedos* se mueven manteniendo pulsada simultáneamente la tecla *Mayus*), por lo que es recomendable probar la aplicación en el dispositivo o dispositivos finales para los que ha sido diseñada.

Apple no permite desarrollar en dispositivos físicos reales a menos que se cree una cuenta de desarrollador de *Apple* y se registre el dispositivo o los dispositivos físicos pagando una cuota anual. Sin embargo existen alternativas que permiten usar dispositivos reales sin tener que darse de alta como desarrollador, donde la más cómoda es una aplicación gratuita que puede descargarse en `jailcoder.com`.

Una vez se está ejecutando el programa, independientemente de la naturaleza virtual o real del dispositivo, se mostrará la información pertinente en los distintos

[2]Error, fallo o defecto en un programa informático normalmente debido a errores humanos producidos durante el desarrollo del programa.

[3]*Application Programming Interface*, Interfaz de programación de aplicacioness

paneles de *XCode*, como por ejemplo la salida de consola en la zona *Debug*.

Para detener la ejecución puede pulsarse el botón de Stop en la barra de tareas o bien cerrar la aplicación *iOS Simulator*, aunque se recomienda dejarla abierta porque tarda cierto tiempo en abrirse cada vez.

8.8. Depuración

No siempre es fácil depurar los errores en tiempo de ejecución. Por defecto el código mostrado con los fallos es código en lenguaje ensamblador, difícil de leer. Por ello recientemente han incorporado una funcionalidad que permite indicar qué línea del código sin compilar es la que ha producido el fallo. Para conocer cómo se añade esa funcionalidad, ver el panel *Breakpoint Navigator* en la fig. 8.3, pág. 22.

9. El lenguaje *Objective-C*

9.1. Descripción

Objective-C es el lenguaje de programación empleado por *Apple* en sus sistemas. Consiste de una capa muy fina situada por encima de *C*, compuesta por un conjunto de extensiones que implementan un modelo de objetos basado en *Smalltalk*, uno de los primeros lenguajes de programación orientado a objetos. Es un superconjunto estricto de *ANSI C* y por tanto soporta la misma sintaxis básica que *C*. Se trabaja como en *C*, con ficheros de cabecera `.h` para definir la especificación y ficheros de implementación `.m`.

Una de las características de este lenguaje es que proporciona un tipado y enlazado *dinámicos*, por lo que posterga la mayoría de las responsabilidades hasta que no se está en tiempo de ejecución. Esto implica que no se conoce realmente qué objeto ejecutará un método concreto hasta que no se ejecute la aplicación. Debido a esto aparece el concepto de *mensaje* en la forma de comunicación entre los objetos, término ampliamente empleado en la documentación de *Apple*.

En otros lenguajes de programación orientada a objetos, como *C++*, el nombre de un método está vinculado a una clase en tiempo de compilación, pero en *Objective-C*, el nombre de un mensaje sigue siendo un nombre hasta que llega el momento preciso en tiempo de ejecución donde el objeto receptor ya ejecuta realmente un método compilado. Cuando un objeto *A* le comunica a otro objeto *B* que debe realizar cierta tarea, es decir, el objeto *B* debe ejecutar un método propio, el objeto *A* se convierte en el emisor (*sender*) de un mensaje —el nombre del método— que envía a un receptor (*receiver*), de forma que el receptor al recibir el mensaje, busca el método correspondiente y lo ejecuta.

En *Objective-C* no existe el concepto de *Espacio de Nombres*, por lo que hay que tener en cuenta las posibles colisiones con los nombres de las clases, ya que esos nombres son visibles en todo el proyecto. El convenio que se sigue es anteponer al nombre de las clases y protocolos un prefijo formado por dos o tres letras mayúsculas. Por ejemplo, las clases del *framework* `UIKit` comienzan por el prefijo `UI`, las del *framework* `Foundation` por el prefijo `NS`, las del *Core Graphics* con el prefijo `CG`, etc. Por tanto, es recomendable elegir unas siglas para el nombre de nuestras clases que no esté reservada. Como norma general puede usarse cualquier combinación de tres letras. Por ejemplo en este proyecto se han empleado las siglas `PDB` correspondientes al nombre y apellidos del autor.

9.2. Cadenas de texto

En *Objective-C*, al ser una superclase de *C*, pueden emplearse las cadenas de caracteres normales tipo **`char*`**. Sin embargo *Objective-C* posee un tipo de dato más adecuado para esta tarea: la clase **`NSString`**. Por ejemplo, para crear un nuevo texto se procede del siguiente modo:

```
NSString* saludo = @"¡hola a todos!";
```

Nótese que a la hora de escribir cadenas de texto directametne, éstas deben ir entre comillas dobles y precedidas por el símbolo de la arroba (**`@`**).

Con respecto al asterisco que sigue a la palabra **`NSString`**, se debe a que *Objective-C* hereda la notación de punteros de *C*, pero no es necesario conocerla en *Objective-C*. Simplemente hay que recordar colocar el asterisco tras el nombre de una clase (si se nos olvida ponerlo nos avisará, como siempre, el compilador de *XCode*).

Para mostrar cadenas de texto en la consola de *XCode* se emplea la función **`NSLog()`**:

```
NSLog(\"El saludo es: %@\", saludo);
```

La notación **`%@`** sirve para referenciar a objetos, en este caso, a un **`NSString`**.

9.3. Clases

En los lenguajes orientados a objetos una clase es la construcción básica para encapsular datos con las acciones que operan sobre los datos. Un objeto es una instancia en tiempo de ejecución de una clase que contiene su propia copia de las variables de instancia declaradas por esa clase y punteros a los métodos de la clase.

En *Objective-C* la especificación de las clases se divide en dos partes: la interfaz, que se suele definir en un archivo de cabecera **`.h`**, contiene la declaración de la clase: variables de instancia y métodos; y la implementación, que se suele definir en un archivo **`.m`**, contiene el código de los métodos. La figura 9.1 muestra un ejemplo de implementación de una clase en el lenguaje *Objective-C*.

El método constructor por defecto de una clase es el método **`init`**, heredado de la superclase **`NSObject`**. Si se incorporan constructores personalizados, el nombre del método debe comenzar con el prefijo **`init`**.

9.4. Métodos

En *Objective-C* las funciones o procedimientos se denominan *métodos* (*methods*). Hay 2 tipos de métodos: métodos de instancia y métodos de clase. Los métodos de instancia, que se declaran anteponiendo el signo menos (**`-`**) al nombre del método, como su nombre indica, sólo pueden usarse cuando se ha creado un objeto, esto es,

PDBPersona.h (archivo de cabecera .h)

```
#import <Foundation/Foundation.h>

@interface PDBPersona : NSObject
{
    NSString *empleo;
}

@property NSString* dni;
@property int edad;
@property NSString* nombre;

-(id)initConDni:(NSString*)dni edad:(int)edad;

+(PDBPersona*)personaConDni:(NSString*)dni edad:(int)edad;

@end
```

nombre de la clase

clase padre

declaración de variables de instancia

declaración de properties

declaración de método de instancia (es el constructor del objeto)

declaración de método de clase

PDBPersona.m (archivo de implementación .m)

```
#import "PDBPersona.h"

@implementation PDBPersona

-(id)initConDni:(NSString*)dni edad:(int)edad{
    self = [super init];
    if (self) {
        _dni = dni;
        _edad = edad;
    }
    return self;
}

+(PDBPersona*)personaConDni:(NSString*)dni edad:(int)edad{
    PDBPersona *nuevaPersona = [[PDBPersona alloc] initConDni:dni edad:edad];
    return nuevaPersona;
}

@end
```

implementación del constructor

Figura 9.1: Ejemplo de una clase en *Objective-C*.

se ha instanciado una clase. Por otra parte, los métodos de clase, que se declaran anteponiendo el signo (**+**) al nombre del método, pueden usarse directamente desde la clase.

La declaración de un método consiste en:

- El identificador de tipo de método: el signo menos (**-**) si es un método de instancia o el signo más (**+**) si es un método de clase.
- El tipo de dato devuelto: puede ser un tipo de datos básico (**`int`**, **`float`**,...), una clase (**`NSString`**, **`UIView`**, etc.) o si no devuelve nada, se define como **`void`**.

- Una o varias palabras que definen el nombre del método, en función del número de parámetros de entrada.
- Los tipos de parámetros de entrada y sus nombres.

Por ejemplo, un método que devuelve un objeto de tipo **PDBPersona** a partir de su edad y dni se declararía de esta forma:

```
+(PDBPersona*)personaDeEdad:(int)edad conDni:(NSString*)dni
```

Para usar los métodos se emplea la notación corchetes (**[ ]**) donde primero se escribe el nombre del objeto y luego el nombre del método con sus parámetros. Los corchetes pueden anidarse, de forma que la llamada a un método puede usarse a su vez como parámetro de otro método.

Este ejemplo muestra cómo crear un objeto a partir de su constructor por defecto. Nótese el anidamiento de los corchetes:

```
PDBPersona *unaPersona = [[PDBPersona alloc] init];
```

Cuando se usa el constructor por defecto, la creación de objetos puede resumirse del siguiente modo:

```
PDBPersona *unaPersona = [PDBPersona new];
```

Este método muestra un ejemplo del empleo de un método de clase que se encarga de construir y devolver un objeto de tipo **PDBPersona** con valores iniciales.

```
PDBPersona *juanjo = [PDBPersona personaDeEdad:25 conDni:@"12345A"];
```

9.5. Properties

A medida que evoluciona el SDK de *iOS* las variables de instancia están en desuso a favor del empleo de las denominadas *Properties*. Este mecanismo de *Objective-C* automáticamente genera una variable de instancia interna (a la que puede accederse anteponiendo el signo subguión (**_**) al nombre de la *property*) y los métodos de acceso (*getter*) y asignación (*setter*) a la variable interna. Los métodos de tipo (*getter*) (*setter*) son muy comunes en la programación orientada a objeto porque garantizan la encapsulación de los datos (uno de los principios de la programación orientada a objetos), ya que el acceso a ellos no se hace directamente accediendo a las variables internas, sino mediante métodos específicos. Las *properties* de *Objective-C* simplifican enormemente la tarea de implementar esos tipos de métodos, puesto que al declararlas, se generan automáticamente de forma interna, aunque pueden redefinirse para modificar la implementación por defecto.

Por ejemplo, al realizar la declaración de esta *property*:

```
@property int edad;
```

de forma interna se generan automáticamente de forma implícita los siguientes métodos, sin ninguna intervención por parte del desarrollador —aunque éste también puede definirlos explícitamente para personalizar el comportamiento—:

```
-(int)edad{
  return _edad;
}

-(void)setEdad:(int)edad{
  _edad = edad;
}
```

Para usar las *properties*, puede accederse a ellas mediante la notación punto o bien mediante el *getter* o el *setter* (**unaPersona** es una instancia de la clase **PDBPersona**):

"**unaPersona.edad = 32;**" es equivalente a: "**[unaPersona setEdad:32];**"
"**int edad = unaPersona.edad;**" es equivalente a: "**[unaPersona edad];**"

9.6. Protocolos

Los protocolos son análogos a las interfaces de *JAVA*: declaran métodos que pueden ser implementados por cualquier clase que adopte el protocolo. Se usan frecuentemente para definir la interface de los objetos delegados.

Para conocer más detalles acerca de *Objective-C* se recomienda leer los documentos de la sección Documentación de *Objective-C* en la Bibliografía.

El documento [**9**] describe los aspectos fundamentales de *Objective-C*.

El documento [**14**] describe las convenciones adoptadas en la nomenclatura de clases, métodos, protocolos, etc. de *Objective-C*.

10. Vistas y *View Controllers*

Una vista es un objeto de una clase que hereda de la clase `UIView`. Es la mínima unidad visual sobre la que se basan el resto de vistas y controles de *iOS*. Todos los elementos visuales se dibujan en última instancia en un objeto `UIView`. Además, puede añadirse subvistas a las vistas y crear jerarquías complejas.

Además de mostrar contenido visual, las vistas también se encargan de analizar los gestos táctiles que realiza el usuario sobre ellas. La clase `UIView` posee métodos para analizar los datos en bruto de los eventos táctiles del usuario. Sin embargo, puede resultar muy complejo el análisis de esos datos en bruto, por lo que se recomienda emplear los denominados *reconocedores de gestos*, que son objetos de alto nivel que simplifican enormemente la gestión de los eventos táctiles.

Globalmente la estructura de vistas de *iOS* está basada en los denominados *View Controllers*, objetos que pertenecen a la capa de *Controlador* dentro del patrón de diseño *Modelo - Vista - Controlador*[1], y por lo tanto actúa como intermediario entre los datos del modelo y las vistas que presentan dichos datos. Un *controlador de vista* gestiona una porción concreta de la GUI. Cada una de las vistas que se muestran en la pantalla del dispositivo normalmente están controladas por un *View Controller*.

Internamente posee una vista raíz que puede mostrarse y con la que se puede interactuar, y que normalmente es la vista raíz de una jerarquía de vistas más compleja (botones, deslizadores, campos de texto, subvistas, etc.). El *controlador de vista* actúa como coordinador de esa jerarquía de vistas, gestionando el intercambio de datos entre las vistas y los objetos de la capa de modelo.

También gestiona otros aspectos como la memoria o los cambios de orientación del dispositivo.

Hay un tipo especial de *View Controllers* cuya función es presentar a otros *View Controllers*: el *Navigation Controller* y el *Tab Bar Controller*.

El *Navigation Controller* gestiona una pila de *View Controllers* proporcionando una interfaz para navegar por la jerarquía de contenidos. Automáticamente añade en la parte superior una barra de navegación con un botón a la izquierda para volver a un *View Controller* previo.

El *Tab Bar Controller* muestra en la parte inferior de la pantalla una barra de pestañas para seleccionar un *View Controller* concreto. Es la típica vista de aplicaciones como *App Store*, *Music*, *iTunes*, *Whatsapp*, etc.

Pueden encontrarse más detalles acerca de los *View Controllers* en los documentos [**15**] y [**16**].

[1]Para más información acerca del patrón MVC ver la sección 7.1

11. Reconocedores de Gestos

Las *apps iOS* se controlan principalmente mediante eventos generados cuando el usuario toca los controles presentes en la interfaz de usuario de la *app*: botones, celdas de tablas, etc. Las clases de `UIKit` proporcionan un comportamiento por defecto para la gestión de eventos en la mayoría de esos objetos. Sin embargo, si la *app* tiene requisitos específicos, puede que esos comportamientos predefinidos no sean suficiente, por lo que se ha de crear una gestión de eventos específica, principalmente cuando se usan vistas personalizadas. La *app* deberá analizar el flujo de datos en bruto de los objetos táctiles para determinar la intención del usuario.

Sin embargo, el código para examinar datos en bruto de eventos táctiles y conseguir detectar varios gestos es normalmente muy complejo.

Por ello, en *iOS* versión 3.2 se presentaron los *reconocedores de gestos*, objetos que heredan directamente de la clase `UIGestureRecognizer`. Estos objetos simplifican enormemente la tarea de reconocer gestos. Incluyen seis reconocedores predefinidos de los gestos más comunes. Se instalan en los objetos de tipo `UIView` y pueden configurarse para reconocer cierto número de dedos simultáneamente o cierto número de toques.

Se distinguen dos tipos de reconocedores de gestos:

reconocedores de gestos discretos por ejemplo, realizar una pulsación sobre la pantalla. Este tipo de reconocedores sólo envían una vez información del evento cuando es detectado el gesto.

reconocedores de gestos contínuos que una vez detectan el gesto, envían constantemente información de eventos al método configurado para gestionarlo —por ejemplo, el gesto de pulsar sobre la pantalla y arrastrar el dedo para desplazar el contenido —.

Esta es la lista de los seis reconocedores de gestos que incluye por defecto `UIKit`:

`UITapGestureRecognizer` De tipo **discreto**. Reconoce el gesto de pulsar (en la documentación se denomina *tapping*). Puede configurarse para reconocer cualquier número de pulsaciones.

`UIPinchGestureRecognizer` De tipo **contínuo**. Reconoce el gesto de *pellizcar* (en la documentación denominado *pinching*), tanto al juntar como separar los dedos. Es el típico gesto que se emplea para aumentar o reducir la ampliación de las imágenes en la aplicación de *Fotos* incluida por defecto en *iOS*.

`UIPanGestureRecognizer` De tipo **contínuo**. Reconoce el gesto de *arrastrar* el dedo por la pantalla, por ejemplo para poder desplazar contenido que no cabe en la pantalla, como la lista de contactos telefónicos del iPhone.

`UISwipeGestureRecognizer` De tipo **discreto**. El gesto *swipe* envía un único evento cuando detecta que el dedo se desliza sobre la pantalla en cierta dirección.

`UIRotationGestureRecognizer` De tipo **contínuo**. Detecta cuando dos dedos se mueven en direcciones opuestas, permitiendo, por ejemplo, girar una imagen.

`UILongPressGestureRecognizer` De tipo **contínuo**. Detecta el gesto de pulsación larga. Consiste en mantener pulsado el dedo sobre la pantalla hasta que el gesto reconoce la acción, también conocida como *pulsar y mantener*. Como es contínuo, una vez se reconoce el gesto, el dedo o dedos pueden moverse por la pantalla y los movimientos se enviarán como eventos al reconocedor, de forma similar al funcionamiento del reconocedor **`UIPanGestureRecognizer`**.

En el código de ejemplo de la figura 11.1 se muestra cómo crear, configurar e instalar reconocedores de gestos. Se supone que es un método implementado en un *View Controller*, por eso posee la propiedad **`self.view`**.

```
-(void)crearVistaConGesto
{
    // Crear una vista nueva como ejemplo
    UIView *nuevaVista = [[UIView alloc]
                          initWithFrame:CGRectMake(0.0, 0.0, 320.0, 480.0)];

    // Crear el reconocedor de gesto para que reconozca una doble pulsación
    // realizada con un dedo
    // Al crearlo se le indica el objeto que gestionará el reconocedor (Target)
    // y el método que se llamará cuando se reconozca el gesto (action)
    UITapGestureRecognizer *doubleTapGesture = [[UITapGestureRecognizer alloc]
                                                initWithTarget:self
                                                action:@selector(handleDoubleTapGesture:)];

    // Configurar el reconocedor para que reconozca una pulsación doble
    [doubleTapGesture setNumberOfTapsRequired:2];

    // Configurar el reconocedor para que reconozca sólo un dedo
    [doubleTapGesture setNumberOfTouchesRequired:1];

    // Instalar el reconocedor en la nueva vista
    [nuevaVista addGestureRecognizer:doubleTapGesture];

    // Incluir la nueva vista como subvista
    [self.view addSubview:nuevaVista];
}
```

Figura 11.1: Método de ejemplo que crea una nueva subvista con un reconocedor de gestos instalado

En el método asignado como **`action:`** es donde se realiza la gestión del gesto una vez es detectado como tal. Ese método recibe como argumento el objeto reconocedor de gesto del que se puede extraer diversa información, como las coordenadas de la pulsación en la pantalla.

Durante el proceso de reconocimiento de gestos contínuos, el gesto va pasando por una serie de estados que se gestionan en el método designado como **`action:`**. La información de los estados se extrae del objeto reconocedor que recibe el método como argumento.

El código de la figura 11.2 muestra un ejemplo de implementación del método manejador asignado como **`Action:`** a un reconocedor de gestos contínuo con los estados por los que se va pasando durante el proceso de reconocimiento del gesto.

```
-(void)handleLongPressGesure:(UITapGestureRecognizer*)gesture
{
    switch (gesture.state) {
        case UIGestureRecognizerStateBegan:
        {
            // El usuario ha mantenido pulsada la vista un cierto tiempo
            // y el reconocedor ha reconocido el gesto como suyo
            // Realizar aquí la implementación de las tareas iniciales al
            // reconocerse el gesto, por ejemplo seleccionar un objeto para
            // moverlo indicándolo cambiando su color.
            break;
        }
        case UIGestureRecognizerStateChanged:
        {
            // El usuario sigue pulsado la vista y ha movido el dedo
            // Mientras el usuario esté moviendo el dedo, se ejecuta
            // contínuamente el código aquí implementado, por ejemplo,
            // código para desplazar un objeto por la pantalla
            break;
        }
        case UIGestureRecognizerStateEnded:
        {
            // El usuario ha levantado el dedo, lo que produce la finalización
            // normal del gesto. Aquí se pueden realizar tareas como
            // deseleccionar el objeto indicándolo restaurando su color original.
            break;
        }
        case UIGestureRecognizerStateCancelled:
        {
            // El gesto ha sido interrumpido debido a que se ha recibido
            // un evento del sistema como una llamada telefónica entrante,
            // un aviso de batería baja, etc.
            break;
        }
        default:
            break;
    }
}
```

Figura 11.2: Ejemplo de implementación de un método manejador de eventos contínuos.

Puesto que el reconocimiento de gestos es la interfaz principal del usuario con el dispositivo, se han empleado intensivamente los reconocedores de gestos en la *app* desarrollada en este proyecto. Cuando se instalan simultáneamente varios reconocedores de gestos hay que tener en cuenta que puede haber conflictos entre ellos a la hora de reconocer un gesto como suyo, por lo que habrá que tenerlo en cuenta y corregirlo empleando una combinación de reconocedores de gestos que sea menos conflictiva. Además, los reconocedores pueden configurarse para esperar a que el gesto se complete para discernir si debe reconocerlo como suyo. Como ejemplo supongamos que se implementa simultáneamente un reconocedor de gestos que de-

tecta una única pulsación y otro reconocedor que detecta una pulsación doble. Si no se configuran adecuadamente, entonces el reconocedor del gesto de pulsación doble jamás llegará a activarse, puesto que cualquier pulsación será reconocida como un único gesto por el reconocedor de gestos único. Esto se soluciona diciéndole al reconocedor de pulsación única que se espere a reconocer el gesto como suyo hasta que el otro reconocedor falle, es decir, no reconozca el gesto de doble pulsación. La implementación sería simplemente así:

```
[singleTapGesture requireGestureRecognizerToFail:doubleTapGesture];
```

Este método le dice al **`singleTapGesture`** que se espere un poco hasta reconocer el gesto, por lo que añade cierto retardo al reconocimiento del gesto. Esto normalmente no afecta de forma notable a la experiencia del usuario.

Para conocer más detalles acerca de la tecnología de gestión de eventos en *iOS*, ver el documento [**22**].

12. Las *apps iOS*

12.1. Tipos de *apps*

Hay dos tipos de *apps* que pueden crearse para *iOS*: de tipo *nativo* o de tipo *web*. Las *nativas* emplean el lenguaje *Objective-C* y se instalan físicamente en el dispositivo, estando siempre disponibles para su uso, incluso en modo *offline* (sin conexión a internet), mientras que las de tipo *web* se crean usando una combinación de código de programación de páginas web *HTML*, hojas de estilo en *cascada* (*CSS*) y código *Javascript*; se acceden a través del navegador de internet *Safari*, por lo que es necesario disponer de conexión a internet para poder usarlas.

Las aplicaciones *nativas* son más completas y su interfaz gráfica se ejecuta de forma más fluida, en conclusión, la experiencia del usuario es mejor. La *app* diseñada en este trabajo es de tipo *nativo*. Respecto de las aplicaciones de tipo *nativo*, por motivos de seguridad, *iOS* restringe el ámbito de cada aplicación a un único directorio específico de la aplicación en el sistema de archivos, función conocida como *sandbox* (*cajón de arena*). Ninguna aplicación puede ver más allá de su *sandbox*. No tienen acceso a los ficheros del sistema o de otras aplicaciones. Este mecanismo no evita ataques informáticos a una aplicación que puedan vulnerar su seguridad, sino que evita que posibles fallos de seguridad de una aplicación puedan propagarse más allá de su *sandbox* y afectar a otras aplicaciones o al propio sistema operativo. Toda la información referente a la aplicación, como la configuración de usuario u otros archivos, residen en su *sandbox*.

12.2. Ayudas visuales

Para que la aplicación tenga retroalimentación con el usuario, *Apple* recomienda incorporar pequeñas ayudas visuales —por ejemplo animaciones— a cada una de las acciones que el usuario realiza en el dispositivo. De este modo, aunque la acción realizada por el usuario necesite cierto tiempo para completarse, por ejemplo cargarse una página web, el usuario obtiene una primera respuesta por parte del sistema indicándole que ha recibido la acción, dándole la sensación de que el sistema funciona y responde. En un sistema inerte que no proporciona retroalimentación rápida al realizar acciones se crea confusión en el usuario hasta que se produce realmente la acción final, puesto que al ver que la interfaz no reacciona, no le queda claro si ha realizado correctamente la acción o tiene que volver a repetirla.

Además también pueden tener una función informativa, por ejemplo produciéndose una animación distinta en función de la respuesta del sistema a la acción realizada por el usuario. Por ejemplo, al borrar un archivo de una tabla, podemos crear una animación que hace desaparecer la celda correspondiente al archivos si se ha borrado correctamente, y otra animación distinta, por ejemplo, que *agite* la celda, para indicar que dicho archivo no puede eliminarse. De hecho muchos de los controles de *Apple* proporcionan por defecto y de forma automática muchos tipos de animaciones como respuesta a acciones realizadas por el usuario o bien cuando se cambian ciertas propiedades del control, por ejemplo el color de fondo.

12.3. Estructura de las *apps*

El *framework* `UIKit` proporciona la infraestructura de las *apps* a la que se incorporan las clases y objetos personalizados que definen una *app* específica. Cuando se crea un nuevo proyecto en *XCode* 4 automáticamente se genera un *esqueleto* a partir del cual puede el desarrollador incorporar su código, que incluye, entre otros, el archivo de diseño de la interfaz gráfica —el *Storyboard*— y la clase que implementa el delegado de la *app* que se describe más adelante, clase encargada de gestionar la *app* cuando cambia de estado de ejecución. También existe una serie de plantillas según el tipo de *app* que pretende desarrollarse, como una basada en *Navigation Controllers* (plantilla *Master-Detail Application*) o basada en una única aplicación (plantilla *Single View Application*). Todos estos proyectos recién creados ya son de por sí programas funcionales que pueden ejecutarse y navegar entre sus vistas (por ejemplo en caso de usar la mencionada plantilla *Master-Detail Application*).

Las *apps* tienen un ciclo de vida donde van transitando por estados a medida que se abren, se cierran, se cambia de *app*, etc. Cuando una aplicación está activa en primer plano, se encuentra en el estado denominado ciclo de eventos, que consiste en mantenerse a la espera de eventos, procesarlos cuando se reciben, y volver a mantenerse a la espera cuando se finaliza el procesado.

Los objetos principales que proporciona `UIKit` son los encargados de gestionar el ciclo de eventos y las principales interacciones con *iOS*. Combinando técnicas como subclases, delegación y demás se modifica el comportamiento por defecto definido por `UIKit` para implementar una *app* específica.

12.4. Objetos fundamentales de las *apps*

El objeto principal de la estructura básica que proporciona `UIKit` es el denominado `UIApplication`, que recibe eventos del sistema y los envía a las partes del código creado por el desarrollador que gestionan esos eventos.

Para comprender mejor cómo los objetos `UIKit` interactúan con el código creado por el desarrollador conviene conocer la estructura global que clasifica los objetos de una *app*, basada en el patrón de diseño Modelo-Vista-Controlador. Como puede verse en la figura 12.1 este patrón separa los objetos de las aplicaciones en tres

categorías. Este patrón hace independiente la capa de modelo de la capa de vista, lo que resulta de especial utilidad a la hora de desarrollar las denominadas *apps universales*, que son *apps* que pueden funcionar tanto en iPhone como en iPad, en cada caso con una interfaz de usuario adaptada a las dimensiones de la pantalla del dispositivo concreto.

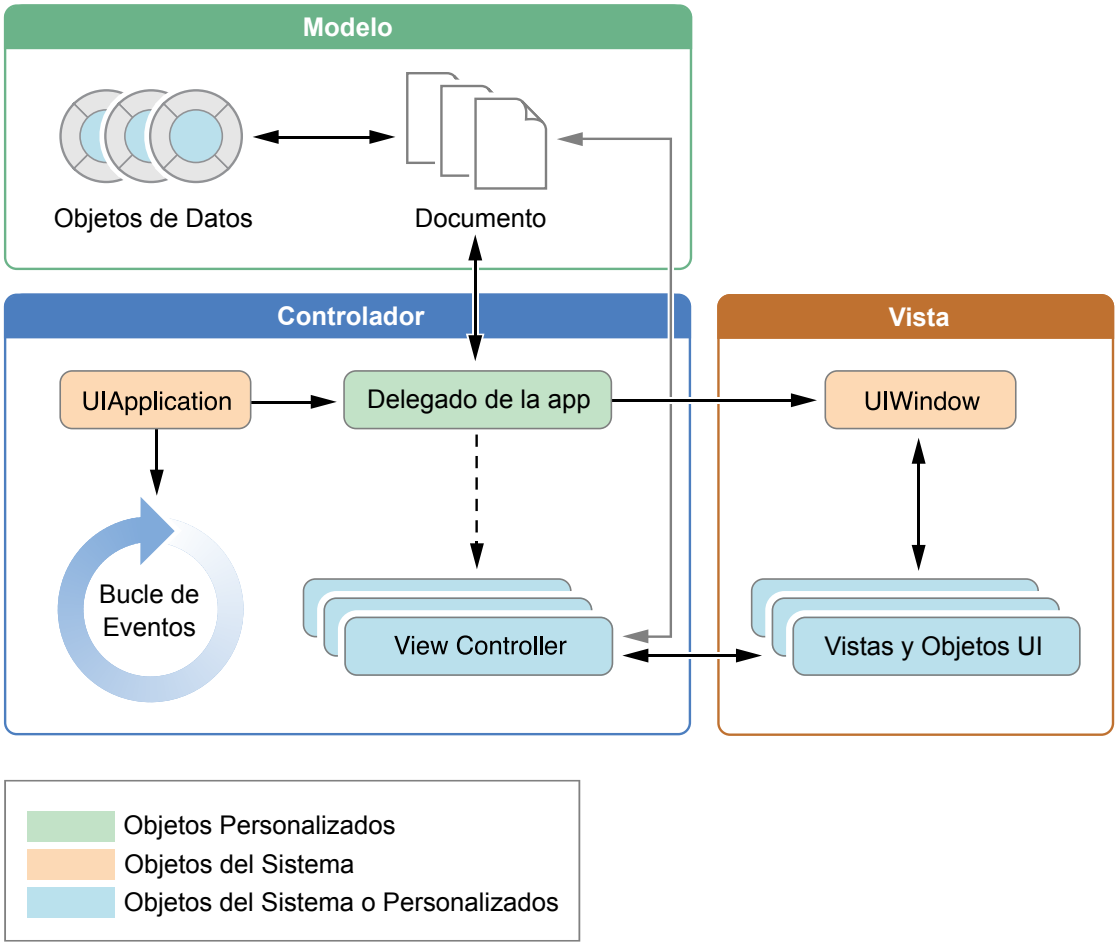

Figura 12.1: Los objetos fundamentales de una *app iOS*.

Los papeles que desempeñan los objetos fundamentales de la figura 12.1 en una *app iOS* son:

objeto `UIApplication` Este objeto controlador gestiona el ciclo de eventos de la *app* y coordina otros comportamientos de la *app* de alto nivel. El desarrollador nunca trabaja directamente con este objeto, sino que se comunica con él con el objeto delegado de la *app*.

objeto delegado de la *app* Es un objeto personalizado creado en el momento del lanzamiento de la aplicación. La principal función de este objeto es gestionar las transiciones entre estados de la *app*.

Documentos y objetos de datos de la capa de Modelo Los objetos de la capa de modelo son específicos en cada *app* y almacenan su contenido.

Las *apps* también pueden usar los denominados *objetos documento*, que son subclases personalizadas de la clase `UIDocument`, para gestionar los datos de la capa de Modelo. Los objetos documentos es una tecnología de `UIKit` que permite de forma bastante automatizada cargar y guardar archivos de documentos, así como realizar operaciones de deshacer/rehacer sobre el documento abierto o marcarlo como que tiene cambios sin guardar. Esta clase está pensada para liberar al usuario de la tarea de guardar documentos, ya que los documentos de este tipo se guardan automáticamente cada cierto tiempo, cuando se detectan cambios en el documento (al realizar operaciones de deshacer/rehacer o al marcarlo como que tiene cambios pendientes) o bien al cerrarlo. En este proyecto se ha empleado este tipo de documentos para gestionar los *patches* de *Pure Data* en la aplicación a partir de los ficheros de datos de *Pure Data*.

La figura 12.2 muestra la relación entre los archivos, los documentos y los objetos de datos del modelo. Cada documento gestiona un único archivo (o un paquete de archivos, que es un directorio especial gestionado como si fuera un documento —en este proyecto se trabaja directamente con archivos únicos—) y crea la representación en memoria de los datos contenidos en ese archivo. Cada *app* posee una carpeta de lectura y escritura ubicada en su *sandbox* para almacenar sus documentos a la que accede el desarrollador programando para leer archivos, copiarlos, guardarlos, etc.

Para conocer más acerca de la tecnología de documentos que proporciona la clase `UIDocument` ver [**27**].

Objetos View Controller Estos objetos son los encargados de gestionar el contenido visual que se muestra en pantalla. Controla una única vista y toda su jerarquía de subvistas. Cuando se presenta un *View Controller*, éste hace visible su vista principal instalándola en la ventana de la *app*.

objeto `UIWindow` El objeto `UIWindow` se encarga de coordinar la presentación de una o más vistas en la pantalla. La mayoría de *apps* sólo tienen una ventana que muestra contenido en la ventana principal, pero se pueden crear *apps* capaces de mostrar contenido también en pantallas externas. Para cambiar el contenido de una *app* se emplean los *View Controllers*, nunca se reemplazan las propias ventanas. Este objeto sólo tiene relevancia para el desarrollador si tiene previsto programar la aplicación para mostrar contenido en pantallas externas al dispositivo. Este objeto además trabaja mano a mano con el objeto `UIApplication` para reenviar eventos a las vistas y *View Controllers* personalizados.

Vistas, controles y objetos capa Las vistas y los controles proporcionan la representación visual del contenido de una *app*. Una *vista* es un objeto de la clase `UIView` que dibuja contenido en una zona rectangular designada y responde a los eventos que se realizan dentro de ella: tocar, arrastrar, pellizcar, etc.

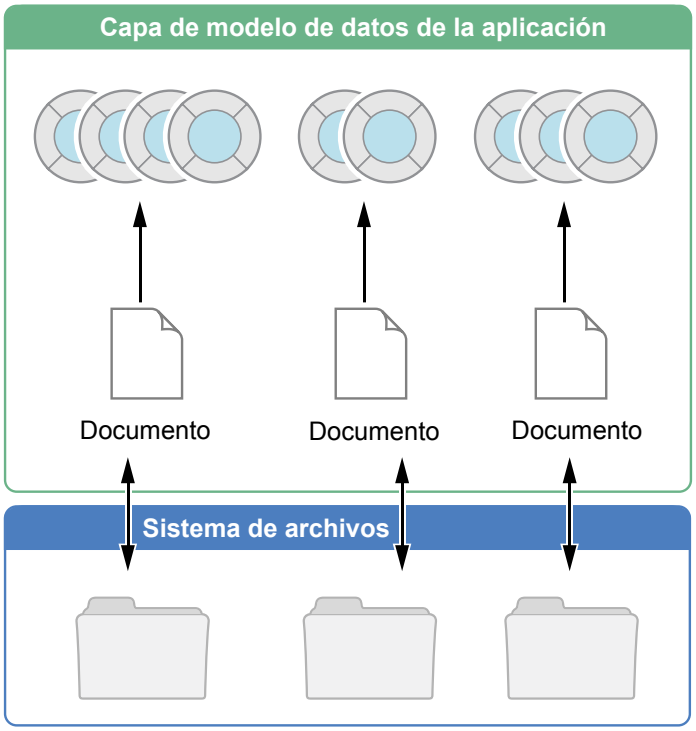

Figura 12.2: Relación entre archivos, documentos y objetos.

Los controles son unos tipos de vistas especializadas incluidos en `UIKit` que proporcionan implementaciones a objetos visuales comunes: botones, campos de texto, interruptores, deslizadores, etc. Todas las vistas tienen por debajo los denominados objetos capa (*layer*), que pertenecen al *framework* de más bajo nivel denominado *Core Animation*. Pueden usarse estos objetos de bajo nivel para implementar animaciones complejas y efectos personalizados que no pueden encontrarse entre las animaciones por defecto de los objetos `UIKit` estándar.

12.5. Estados de las *apps*

En las *apps iOS* es crucial saber si una *app* se está ejecutando en primer plano o en segundo plano. Como los recursos del sistema son más limitados en un dispositivo *iOS* que en un ordenador personal, las *apps* se comportan de forma distinta en segundo plano y en primer plano. Además el sistema operativo limita lo que pueden hacer las *apps* en segundo plano para ahorrar batería y evitar que las aplicaciones en primer plano se ralenticen. El sistema operativo notifica a las aplicaciones cuando cambia entre primer plano y segundo plano. Esas notificaciones permiten a los desarrolladores adaptar el comportamiento de la *app* en función del estado.

Cuando una *app* está en primer plano, el sistema le envía los eventos táctiles que realiza el usuario en la pantalla para que los procese. `UIKit` se encarga de las tareas

más complicadas de este proceso de envío de eventos a los objetos personalizados de una *app*. El desarrollador simplemente tiene que sobrescribir ciertos métodos en los objetos apropiados para procesar los eventos, permitiéndole centrarse en la lógica de la *app* que está desarrollando, sin preocuparse por realizar tareas de bajo nivel de control de eventos táctiles.

Los controles de `UIKit` suben aún más el nivel de abstracción gestionando automáticamente los eventos sin intervención del desarrollador y notificándole sólo cuando se produce la acción realmente importante, por ejemplo cuando el valor de un campo de texto cambia. La figura 12.3 muestra los estados y las transiciones entre estados de las *apps iOS*.

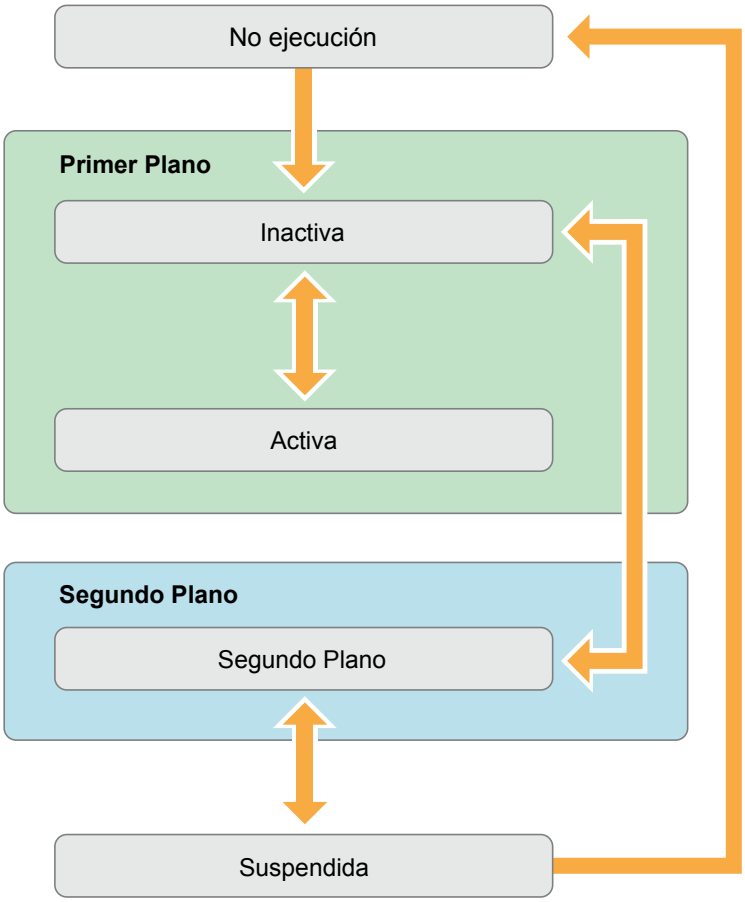

Figura 12.3: Transiciones entre los estados de una app

Cualquier *app* debe gestionar como mínimo dos aspectos:

- Responder apropiadamente a las transiciones de estados de la *app* que se produzcan. Si no, pueden producirse pérdida de datos en la aplicación o inconsistencias en la experiencia del usuario.
- Ajustar adecuadamente el comportamiento de la *app* cuando pasa a ejecutarse en segundo plano. Una *app* que pasa a segundo plano es candidata a ser cerrada potencialmente por el sistema. Por lo tanto es necesario garantizar que todos

los datos han sido guardados cuando está en segundo plano. La tecnología de documentos `UIDocument` automatiza muchas de estas tareas, guardando automáticamente los documentos abiertos si la *app* pasa a segundo plano.

El sistema mueve las *apps* de un estado a otro según ciertas acciones producidas en el sistema. Por ejemplo cuando la memoria RAM está demasiado llena, cuando se pulsa el botón Home, si acaba de entrar una llamada telefónica, etc.

En un momento dado, una *app* está en uno de los siguientes estados:

No ejecutándose la *app* todavía no se ha lanzado o estaba ejecutándose pero ha sido finalizada por el sistema.

Inactiva La *app* está ejecutándose en primer plano pero no está recibiendo eventos del usuario. Sin embargo, puede estar ejecutando otro código. Las *apps* normalmente permanecen brevemente en este estado y luego pasan a un estado distinto.

Activa La *app* está ejecutándose en primer plano y está recibiendo eventos del usuario. Es el modo normal de las aplicaciones en primer plano.

En segundo plano La *app* está en segundo plano ejecutando código. La mayoría de las *apps* entran brevemente en este estado antes de pasar al estado *suspendido*. Este es el momento para realizar operaciones de guardado, puesto que del estado *suspendido* puede pasarse al estado *No ejecutándose*, lo que libera la *app* de la memoria del sistema.

Suspedida La *app* está en segundo plano pero no está ejecutando código. Este es el estado normal de reposo al que entran las *apps* cuando se cambia a otra *app*. La *app* se mantiene en memoria por si vuelve a pasar a primer plano.

La mayoría de transiciones entre estados tienen un método correspondiente en el delegado de la *app* que se ejecuta cuando se realiza una transición concreta. Esos métodos permiten al desarrollador gestionar la lógica específica de la *app* cuando se producen las transiciones entre estados. Cuando se crea un nuevo proyecto *iOS* en *XCode* a partir de una plantilla, por ejemplo, *Single View Application*, automáticamente se genera la clase del delegado de la *app* con los métodos que gestionan las transiciones, cada uno comentado con la explicación de cuándo utilizarlo.

12.6. El proceso de inicio de ejecución

Cuando se lanza una *app*, pasa del estado de no ejecución al estado activa o en segundo plano, pasando brevemente por el estado inactivo. Como parte del proceso de inicio, el sistema crea un proceso y un hilo principal para la *app* y como sucede en *C*, llama a la función `main` de la *app* en ese hilo principal. La función `main` por defecto se incorpora automáticamente al crear un proyecto en *XCode*. Esta función realiza poco trabajo: en cuanto se ejecuta transfiere el control al *framework* `UIKit`,

quien se encarga de la mayoría de las tareas de inicialización y preparación de una *app* para su ejecución.

Las *apps* pueden ejecutarse de diversas maneras:

En primer plano: este es el modo normal de funcionamiento, en el que el usuario explícitamente pulsa el icono específico de la *app*

En segundo plano: la aplicación se ejecuta en segundo plano de forma implícita al realizar cierta tarea externa. Un ejemplo típico es la *app* de *Música*: puede lanzarse en segundo plano para que reproduzca la música de diversas maneras: con el dispositivo bloqueado, si se pulsa dos veces en el botón Home, aparecen los controles de música, entre ellos el típico de *play*; si se conectan auriculares externos, al pulsar el botón central del mando incorporado; en la vista de inicio de *iOS* (la vista que muestra los iconos de las aplicaciones), si se pulsa dos veces el botón Home, aparece en la parte inferior de la pantalla la barra de las *apps* abiertas. Si se desliza esta barra hacia la derecha, aparecen los controles de música, incluido el *Play*.

Para importar un documento: las *apps* pueden configurarse para abrir cierto tipo de documentos de forma externa, por ejemplo al recibir un documento adjunto en un email. En este caso, se ha configurado la *app Pure Data Touch* para reconocer archivos con extensión `*.pd`, que son los ficheros de datos de *Pure Data*. Cuando una *app* se configura de esta forma, el sistema automáticamente la añade a la lista de *apps* que son capaces de abrir ese archivo específico.

En este caso se ha configurado la *app* para que se abra en primer plano y para abrir documentos, pero no para ser ejecutada en segundo plano.

La figura 12.4 muestra la secuencia de eventos que suceden cuando se lanza una *app* en primer plano, incluyendo los métodos del delegado de la *app* que se ejecutan. Durante esta secuencia es cuando se inicializan los elementos globales de la aplicación, como por ejemplo estructuras de datos o tareas comunes. En el caso de esta aplicación la tarea principal que se realiza en el proceso de inicialización es la preparación del audio de la librería libpd.

Para conocer más detalles acerca de los estados y ciclos de ejecución de las *apps* consultar el documento [**1**].

12.7. Importación y exportación de documentos

Como la idea de este programa es poder editar *patches* de *Pure Data* creados en su versión de PC, es imprescindible incorporar a la *app* mecanismos que permitan tanto la importación de documentos externos como la exportación de documentos creados en la *app*.

Una de las tecnologías introducidas últimamente por *Apple* para estos menesteres es el sistema de almacenamiento de archivos en la nube denominado *iCloud*. La idea

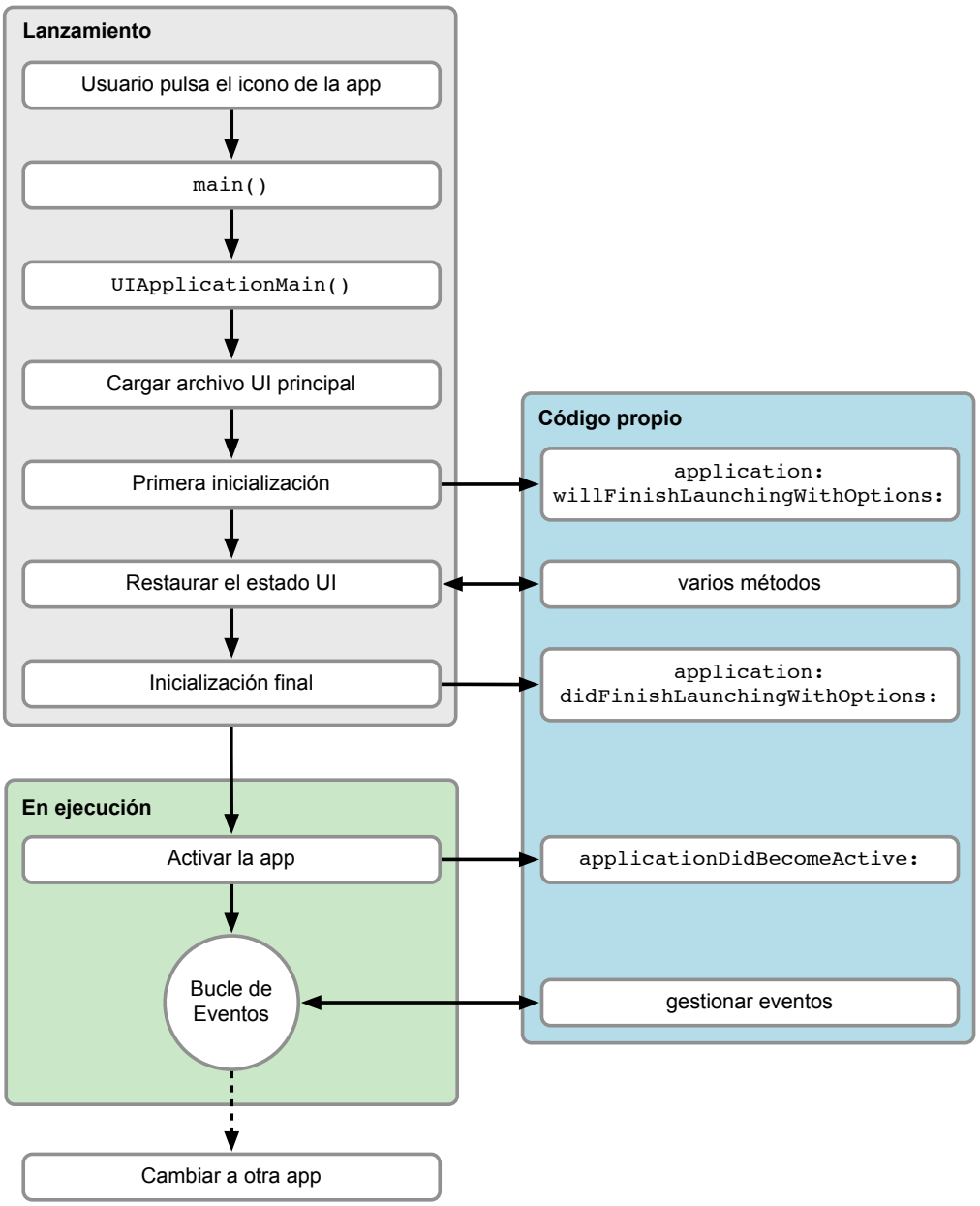

Figura 12.4: Lanzando una *app* en primer plano.

de esta tecnología es abstraer al máximo al usuario de la ubicación de los archivos, estando siempre disponibles en las aplicaciones de todos los dispositivos de *Apple*: teléfonos, tabletas, ordenadores...

Sin embargo, esta tecnología requiere desarrollar aplicaciones que específicamente empleen esa tecnología en cada uno de los dispositivos de *Apple*. Como *Pure Data* es un programa abierto y multiplataforma, para aprovechar *iCloud* habría que rediseñar la aplicación en su versión *Mac OSX* para adaptarla, lo que requeriría una ardua tarea.

Por ello en esta *app* concreta se ha optado por implementar los mecanismos de

exportación de documentos *Pure Data* empleando el correo electrónico: los documentos de *Pure Data* son archivos de texto plano que suelen ocupar unas pocas decenas de *kilobytes*, por lo que enviar y recibirlos por email no supone ningún problema.

En esta *app* pueden enviarse documentos por correo electrónico de dos formas:

- En la vista de la lista de *patches*, al pulsar el botón editar pueden seleccionarse múltiples archivos y enviarlos todos ellos como archivos adjuntos en un único email.

- En la vista de detalle del *patch*, en la parte superior hay un botón para enviar el archivo por correo electrónico.

El mecanismo de envío de correos electrónicos de *iOS* es de muy alto nivel y su uso es muy sencillo, tal y como se explica en [**41**].

Para importar documentos a una *app* se emplea la tecnología denominada UTI[1], que son identificadores de datos de archivos especiales. Cuando una *app* desea importar un tipo de documento, debe crear y registrar una UTI específica para el tipo de archivo que gestiona esos documentos. Al realizar este proceso, el sistema automáticamente incluye la *app* en una lista de *apps* que son capaces de abrir ese archivo. Por ejemplo, la *app* de este proyecto está configurada para importar archivos de *Pure Data* cuya extensión es `*.pd`, de manera que si se recibe un archivo *Pure Data* por correo electrónico, al pulsar sobre el archivo adjunto se mostrará una lista con las *apps* que son capaces de abrirlo, entre las que aparecerá esta *app*

En el documento [**29**] de *Apple* se describe esta tecnología, aunque para ver un ejemplo más práctico recomiendo consultar el siguiente tutorial:
`http://www.raywenderlich.com/1980/how-to-import-and-export-app-data-via-email-in-your-ios-app`

12.7.1. Proceso de importación en ejecución

Cuando se lanza una *app* porque se le ha solicitado que abra un documento, ésta recibe una URL con la ubicación del archivo. Normalmente se encuentra en la carpeta *Inbox* dentro de la carpeta *documents* de la *app*. Es responsabilidad de cada *app* copiar ese archivo a la propia carpeta *documents*, ya que tiene permisos de lectura y escritura. Este procedimiento se realiza en la clase del delegado de la *app*.

Cuando se solicita a una *app* que importe un documento se distinguen dos procedimientos en función del estado de ejecución de la *app*.

Si la *app* estaba en estado de *no ejecución*, entonces tiene lugar la secuencia de eventos de la figura 12.5. A la derecha se muestran los métodos que son llamados en el delegado de la *app*. La figura muestra el proceso de inicialización de una *app* cuando se ejecuta debido a que se requiere para abrir un documento. En el caso de la *app* desarrollada en este proyecto, la tarea que realiza en esta secuencia consiste en importar el documento externo a su carpeta *documents* y mostrar el archivo en la vista de detalle del *patch*.

[1] Uniform Type Identifier

Si la *app* estaba ejecutándose en segundo plano, entonces se produce la secuencia de eventos de la figura 12.6.

En el caso de esta *app*, si estaba en estado de ejecución, importa la URL de manera normal, como puede verse en el diagrama general de la *app* en la figura 13.1 de la página 53, pero si estaba en segundo plano, puede que se hubiera quedado con un archivo abierto porque se estaba editando, por lo que antes de importar el archivo externo, si se da este caso, primero se cierra y guarda el documento abierto.

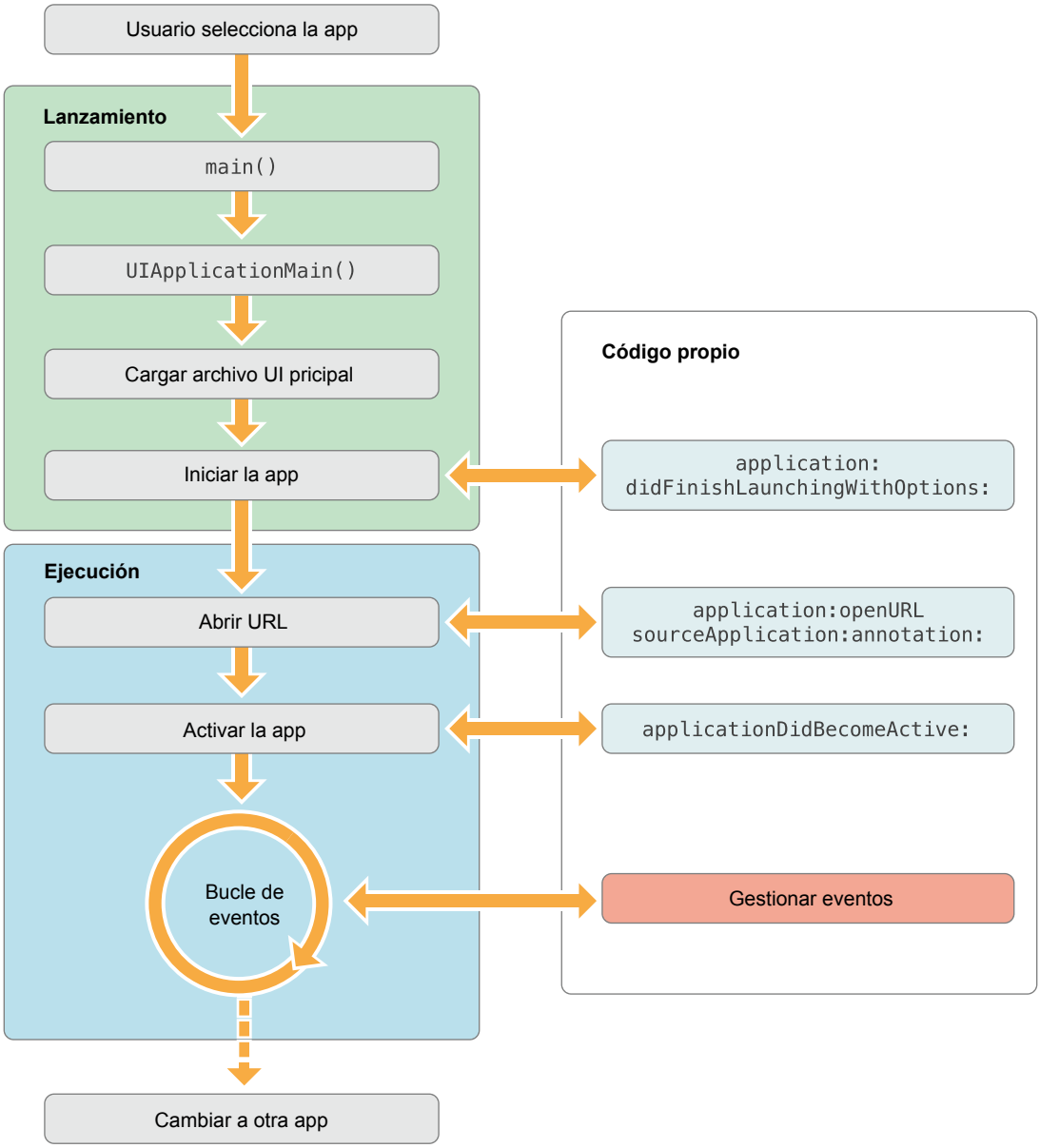

Figura 12.5: Importación de documentos desde el estado de no ejecución.

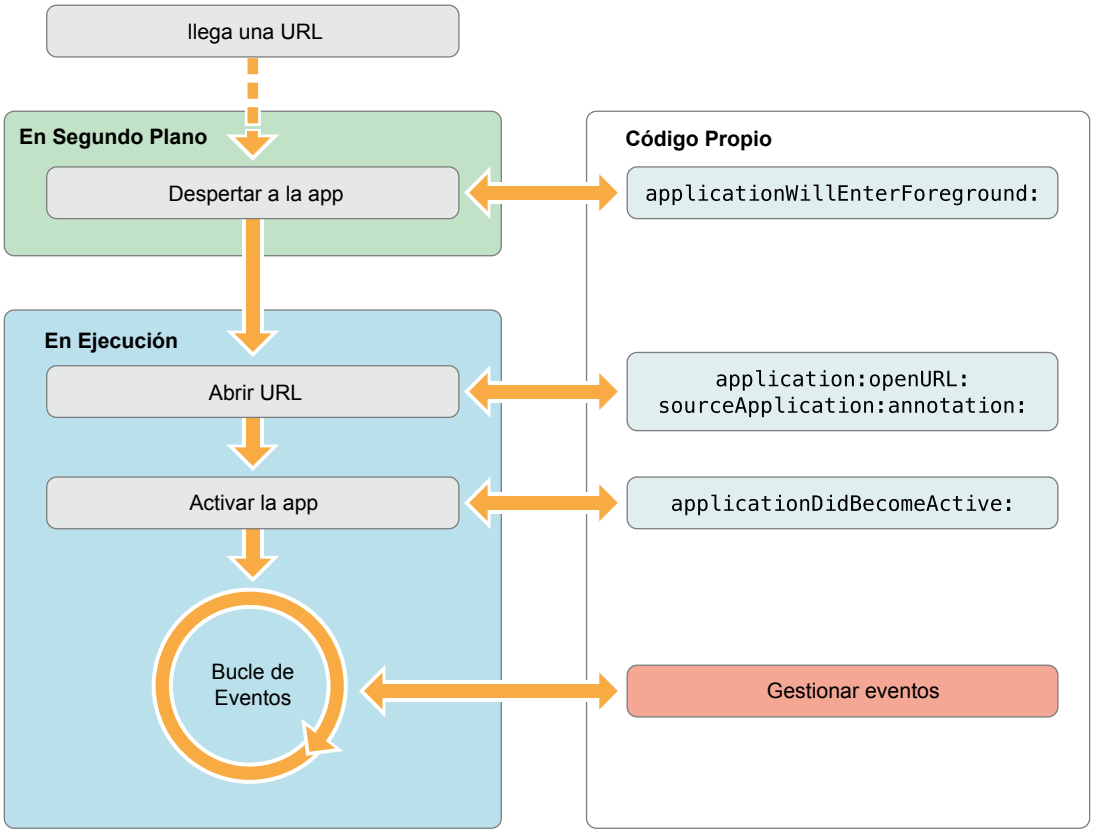

Figura 12.6: Importación de documentos desde el estado en segundo plano.

Parte II

La app Pure Data Touch

13. Descripción general de la *app*

La figura 13.1 presenta el diagrama de flujo general de funcionamiento de la *app*. Muestra las relaciones que hay entre todas las vistas y la secuencia de eventos que se producen al pulsar los controles de la *app*, como botones, celdas de tablas, etc.

A continuación se describe el funcionamiento general de cada una de las vistas y el contenido mostrado en cada una.

13.1. Vista Lista de *Patches*

Esta es la vista que se muestra inicialmente al abrir la aplicación (ver fig. 13.2, pág. 54). Es un *View Controller* embebido en un *Navigation Controller* y muestra una lista en forma de tabla de todos los *patches* almacenados localmente en aplicación. En cada celda se muestra el nombre, tamaño y fecha de modificación por *patch*. Al pulsar sobre la celda se abre la vista del editor de *patch* y al pulsar sobre el botón azul situado a la derecha de cada celda, conocido como *botón accesorio* (*accesory button*), se abrirá la vista de detalle del *patch* seleccionado. En la parte inferior de la pantalla se muestra una barra de herramientas, la cual posee botones para ordenar los *patches* por nombre, por tamaño de fichero o por fecha de modificación. En la parte superior de la pantalla se muestra la barra de navegación del *Navigation Controller*, la cual contiene dos botones con la siguiente funcionalidad:

- El botón de la derecha permite crear un nuevo *patch*. Al pulsarlo sucede lo siguiente:

  1. Se crea un nuevo archivo con un nombre genérico numerado en caso de conflicto (si ya existe un archivo con ese nombre, se añade un sufijo al nombre compuesto por un número).
  2. Automáticamente se abre el archivo en la vista del editor.

- El botón de la izquierda permite activar el modo edición de la tabla, mecanismo interno implementado en las tablas *Cocoa*. La vista en modo edición funciona del siguiente modo:

  - El botón de la izquierda ahora sirve para salir del modo edición y el botón de la derecha desaparece.

- Al pulsar sobre las celdas se seleccionan / deseleccionan indicándose con un marcador rojo situado a la izquierda de cada celda. El *botón accesorio* desaparece en este modo.
- La barra de herramientas de la parte inferior ahora muestra dos botones:
  - El botón de la izquierda con el sobre permite enviar por email todos los *patches* seleccionados. La figura 13.3 de la página 55 muestra el email que se genera automáticamente.
  - El botón de la derecha sirve para eliminar *patches*, mostrándose un aviso de confirmación antes del borrado (ver fig. 13.4, pág. 56). Si no hay ningún *patch* seleccionado, elimina todos los archivos. A medida que se van seleccionando *patches*, el texto del botón muestra un contador indicando el número total de *patches* seleccionados.

Al regresar a la lista de *patches* desde otra vista, ya sea la vista de detalles o la vista de edición, como ayuda visual se mostrará brevemente seleccionada la celda correspondiente al *patch* que se estaba consultando o editando. Así el usuario sabrá cuál es exactamente la celda que abrió la vista de la que acaba de volver. Esto es especialmente importante al crear un nuevo archivo usando el botón de crear arriba a la derecha, puesto que el nombre se genera automáticamente y no es visible para el usuario hasta que vuelve a esta vista. Sin esta ayuda el usuario debería discernir por fecha de modificación para poder encontrar el *patch* que acaba de crear.

13.2. Vista Detalles del *patch*

Muestra información acerca del *patch* y permite renombrarlo, duplicarlo, borrarlo o editarlo en la vista de edición (ver fig. 13.5, pág. 57). También se incluye un botón con un sobre que permite enviarlo por email (ver fig. 13.3, pág. 55). Si al renombrar el archivo existe un conflicto de nombres, se informará al usuario, permitiéndole elegir entre sobreescribir el otro archivo o cancelar el renombrado (ver fig. 13.6, pág. 58).

Al pulsar el botón de Duplicar el *patch* se duplicará y el nombre de la copia estará formado por el nombre del *patch* original seguido por la palabra *'copia'*. Si el nombre de la copia ya está en uso se añadirá además un sufijo numérico para crear un nombre único sin conflictos. Al pulsar el botón de editar el *patch*, se abrirá la vista de edición. Al pulsar el botón de borrar se pedirá confirmación al usuario.

La figura 13.7 de la página 59 muestra a la izquierda el mensaje informativo que aparece al duplicar un *patch*, y a la derecha el mensaje de confirmación de borrado del *patch*.

13.3. Vista Editar *Patches*

Esta vista es realmente el corazón de la aplicación y la más compleja de todas, por lo que se le ha dedicado una sección. Permite editar los *patches* de *Pure Data* de

forma similar a como se hace en la aplicación *Pure Data* para ordenadores personales, pero en este caso de forma táctil.

Se ha intentado fidelizar al máximo el aspecto, la forma y sobre todo las dimensiones de los objetos *Pure Data* con respecto a su versión de PC. De esta forma los *patches* creados o editados con esta *app* pueden abrirse sin sorpresas en la versión de PC, manteniendo la posición, composición y tamaño de los elementos para que la experiencia del usuario sea consistente en ambas plataformas.

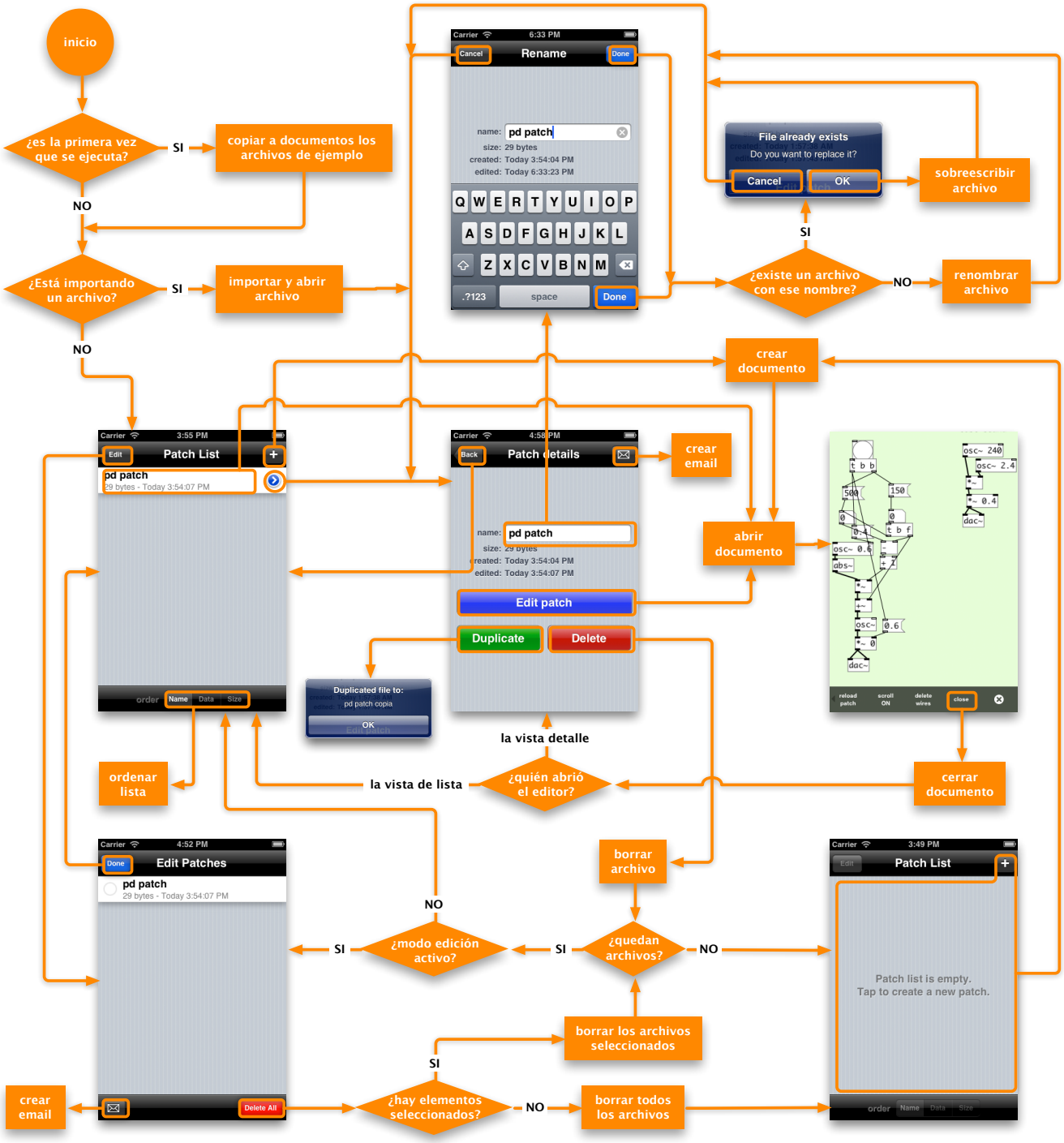

Figura 13.1: Diagrama de flujo general de la aplicación.
Las operaciones de borrado se suponen validadas por el usuario.

Figura 13.2: Vista Lista de *patches*. Arriba: vista de la lista de *patches* en modo normal y en modo edición; abajo: vista de la lista en modo normal apaisado.

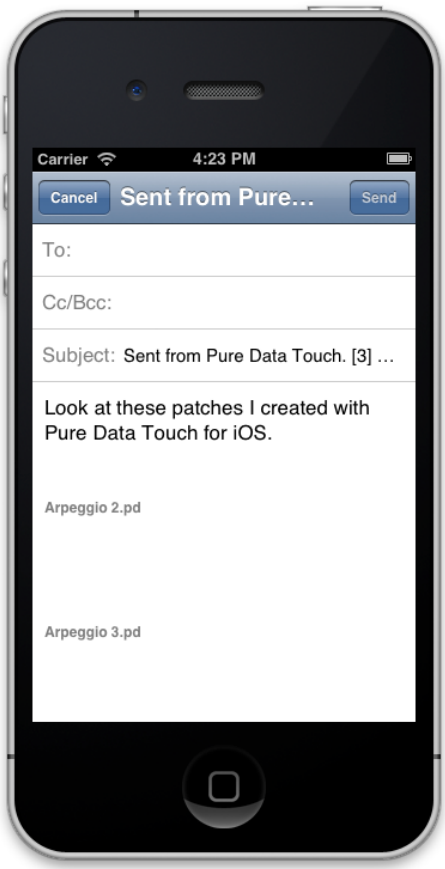

Figura 13.3: Vista del email que se genera automáticamente al enviar *patches* tanto desde la Vista de Lista de *patches* como desde la Vista Detalles del *patch*.

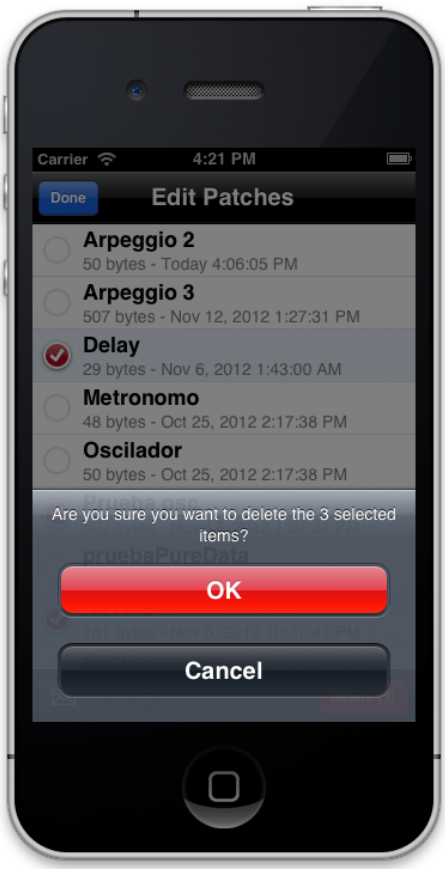

Figura 13.4: Vista Lista de *patches*. Detalle del mensaje de confirmación mostrado al borrar *patches*.

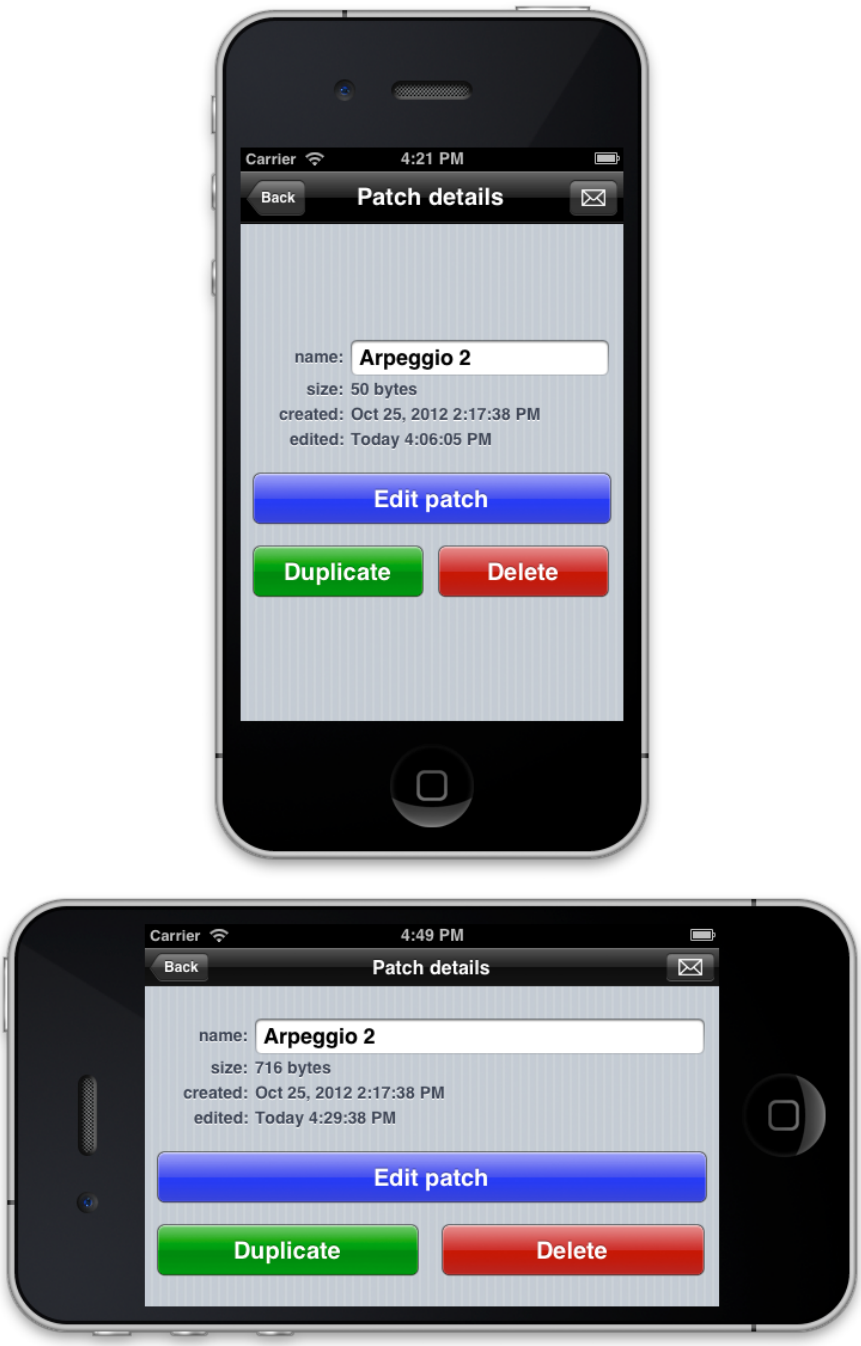

Figura 13.5: Vista Detalles del *patch*.

Figura 13.6: Vista Detalles del *patch*. Detalle de renombrado del *patch*. Si el nuevo nombre ya está usado el sistema muestra un aviso.

Figura 13.7: Vista Detalles del *patch*. Detalle del aviso informativo de duplicación y del mensaje de confirmación de borrado del *patch*.

14. Funcionamiento de la vista Editar Patches

En esta vista es donde se abren los documentos de *Pure Data* a partir de archivos con extensión `*.pd`, conocida en la jerga de *Pure Data* como *patch* (ver fig. 14.1, pág. 65). En el programa original de PC, las dimensiones de los *patches* no tienen límites, por lo que es posible colocar objetos muy alejados entre sí. En el caso de esta *app* se ha preferido acotar estas dimensiones para que el usuario no llegue a sentirse perdido, puesto que las dimensiones de las pantallas de estos dispositivos móviles son mucho más reducidas que las de un monitor de PC. Además de esta forma se evita que el usuario genere demasiados objetos en un único *patch*, incitándole a usar la técnica de *abstractions*, con lo que se consigue evitar saturar la memoria RAM del dispositivo, ya que es un recurso más escaso que en un PC convencional. Cuando se carga un archivo de *Pure Data* cuyos objetos están fuera de los límites de la vista de esta *app*, por ejemplo si el *patch* se ha creado en la versión de PC, la *app* automáticamente ajusta la posición de los objetos para que estén siempre dentro de la zona delimitada. La zona delimitada consiste en un cuadrado de 2048x2048 pixeles.

Esta vista de edición está incrustada en un `UIScrollView`, que es una vista de `UIKit` especializada en la visualización de contenido demasiado grande para mostrarse al completo en la pantalla. En este caso se ha configurado para que realice dos tareas principales:

- Permite desplazar el contenido arrastrando el dedo por la pantalla en cualquier dirección.

- Permite hacer zoom (ampliar/reducir la escala de los elementos mostrados) sobre la vista. Esto se consigue mediante el denominado *gesto pellizcar*, que consiste en emplear dos dedos sobre la pantalla acercándolos como si se *pellizcara* la pantalla o alejándolos. Si durante este gesto se acercan los dedos, la vista se amplía; si se alejan, la escala de la vista se reduce.

El zoom tiene unos límites mínimo y máximo para evitar inconsistencias visuales. El máximo nivel de zoom permite mostrar los elementos muy grandes para trabajar con ellos cómodamente. El mínimo nivel de zoom ajusta todo el contenido (los 2048x2048 pixeles) en la pantalla para visualizarlo de forma global.

La vista de edición de *patches* se controla de dos formas:

- Mediante una serie de gestos que permiten navegar por el contenido, crear objetos, moverlos, seleccionarlos...
- Mediante los botones de la barra de herramientas.

14.1. La barra de herramientas

Por defecto la barra de herramientas está oculta. Abajo a la derecha de la pantalla hay un punto que si se pulsa hace aparecer la barra de herramientas. Para volver a ocultar la barra de herramientas hay que pulsar sobre el botón circular con la equis situado a la derecha.

Cuando el dispositivo está en modo vertical, no se pueden mostrar simultáneamente todos los botones de la barra de herramientas. Debido a esto se han incluido los botones como subvistas de un `UIScrollView`, que permite deslizar la barra de herramientas para mostrar el resto de botones. Como ayuda visual aparece una flecha en la barra de herramientas indicando que en ese sentido hay botones ocultos.

Cuando el dispositivo está en modo horizontal, las dimensiones de la barra de herramientas sí que permiten mostrar todos los botones simultáneamente, por lo que el mecanismo de scroll se desactiva (ver fig. 14.1, pág. 65).

El funcionamiento de los botones es el siguiente:

Edit ON/OFF: Activa o desactiva el modo de edición. En *Pure Data* hay dos modos de funcionamiento: el modo de edición, que permite añadir, mover, borrar y conectar objetos, y el modo de reproducción, donde se desactivan las opciones de edición pero se permite interactuar con los objetos de tipo control gráfico, como los bang, deslizadores, numbers, etc.

DSP ON/OFF: Las siglas DSP significan *Digital Signal Processing* (Procesamiento Digital de la Señal), que es el procesado interno de la señal que realiza *Pure Data* para generar el sonido. Por tanto, este botón permite activar o desactivar el audio del *patch*.

reload patch: Debido a que libpd no fue pensado para editar *patches* de forma explícita, sino más bien para usarse como motor de audio en aplicaciones, no permite editar *patches* y a tiempo real reflejar en el audio los cambios realizados. Para solventar esta carencia, este botón recarga el *patch* en el motor de *Pure Data* para reflejar los últimos cambios realizados en el *patch*.

scroll ON/OFF: Como se han incorporado simultáneamente varios reconocedores de gestos, puede llegar a estorbar el gesto incorporado en el `UIScrollView` que permite desplazar el contenido, por ejemplo al mover un rectángulo de selección. Este botón permite activar o desactivar el scroll para evitar que afecte al resto de gestos.

delete wires: cambia el editor al estado de borrado de cables. Este modo permite eliminar conexiones existentes. La figura 14.8 de la página 72 muestra el

diagrama del funcionamiento general del proceso de borrado de conexiones. En la figura 14.9 de la página 73 se muestran las vistas correspondientes a los estados por los que transita la aplicación durante el proceso de borrado de cables.

close: cierra la vista del editor guardando el documento actual y volviendo a la vista anterior desde la que se le llamó: si se pulsó sobre una celda de la tabla de la vista Lista de *patches* entonces se vuelve a esa vista; si se pulsó sobre el botón *edit patch* de la vista de detalles del *patch*, entonces se regresa a la vista de detalles.

14.2. Gestos incorporados

Además de los gestos que incluye el `UIScrollView` principal que permiten navegar por el contenido desplazando o ampliando la pantalla, se han incluido los siguientes gestos:

gesto para ampliar o reducir la pantalla de forma discreta:
además de poder hacer zoom con dos dedos como se ha comentado, se han incorporado los siguientes gestos adicionales:

- Pulsación simple con dos dedos: si se pulsa la pantalla simultáneamente con dos dedos el contenido se amplía en un factor de 2.
- Pulsación doble con dos dedos: al realizar una doble pulsación con dos dedos la escala del contenido se reduce en un factor de 1/2.

En ambos casos el centro de la transformación del escalado es la posición media entre los dos dedos.

Selección simple de objetos:
basado en un `UITapGestureRecognizer`, permite seleccionar un objeto para mostrar un menú con opciones (editar, borrar...) y desplegar los puertos de conexiones, si tiene. La segunda vista de la figura 14.7 en la página 71, muestra un objeto seleccionado con su menú de opciones y un puerto desplegado.

Si se pulsa sobre el botón de eliminar (*delete*), el objeto será eliminado junto a las posibles conexiones que pudiera tener con otros objetos.

Si se pulsa sobre el botón de editar —sólo disponible en algunos objetos—, se abrirá una vista de edición de objetos, mostrando las opciones del objeto que pueden editarse.

En la figura 14.3 de la página 67 se muestra la vista de edición que aparece al editar un objeto *Pure Data* de tipo *Vslider*. Como sus parámetros son numéricos, al editarlos aparecerá un teclado numérico estándar al que se le han incorporado en la parte superior botones adicionales.

La figura 14.4 de la página 68 muestra la vista de edición que permite modificar los objetos de *Pure Data* que manejan texto, como por ejemplo los objetos *Pure Data* denominados *comment* y *message*.

Si se pulsa sobre un puerto, se inicia el proceso de creación de conexiones, donde el comportamiento del `UITapGesture` se va adaptando en cada uno de los pasos del proceso de creación de conexiones. En la figura 14.6 de la página 70 se muestra la secuencia de eventos que se producen a la hora de crear un cable. Se han incluido además el nombre de los métodos y de las clases empleados en la implementación. En la figura 14.7 de la página 71 se muestran las vistas correspondientes a los estados por los que transita la aplicación durante el proceso de creación de cables.

En el modo de borrado de cables, este gesto se configura para realizar la tarea inversa: seleccionar conexiones para ser eliminadas.

Mover objetos:
está basado en el reconocedor de gestos `UILongPressGesture` configurado para dectectar un sólo dedo, y permite desplazar los objetos por la pantalla para reubicarlos cuando se está en modo de edición activado. La figura 14.10 de la página 74 muestra el diagrama de flujo de las secuencias que se suceden durante el proceso de mover objetos.

Si se mantiene pulsado un objeto, el color de su borde cambia para indicar que puede desplazarse. Si se mueve entonces el dedo, el objeto se mueve por la pantalla.

Debido a las reducidas dimensiones de las pantallas de los dispositivos móviles, se ha incorporado además un complejo mecanismo que desplaza automáticamente el contenido de la pantalla cuando el usuario mueve un objeto hacia los bordes de la pantalla. De esta forma puede reubicar el objeto en una zona de la pantalla no visible sin tener que deseleccionar el objeto, desplazar el contenido de la pantalla un poco, volver a mover el objeto, etc. Esta funcionalidad se denomina *autoscroll*. Entre otras cosas, ajusta la velocidad de desplazamiento en función de la distancia del dedo al margen de la pantalla y detecta si se han alcanzado los límites del documento deteniendo el desplazamiento e impidiendo que el usuario pueda desplazar más la pantalla en esa dirección. Además el autoscroll también se detiene si el usuario mueve el dedo fuera de la zona de los márgenes. La figura 14.5 de la página 69 muestra de forma gráfica las zonas inerte —donde no tiene lugar el autoscroll— y la zona del margen, que mide unos 50 pixeles de ancho.

Seleccionar múltiples objetos:
está basado en el reconocedor de gestos `UILongPressGesture` configurado para dectectar dos dedos. Permite seleccionar múltiples objetos a la vez y así poder desplazarlos conjuntamente por la pantalla. Es la típica selección múltiple implementada mediante un rectángulo cuyas dimensiones varía el usuario para abarcar a aquellos objetos que desea seleccionar. La figura 14.11 de la

página 75 muestra el funcionamiento general de la funcionalidad de selección múltiple.

En este caso, para activarse, el usuario debe mantener brevemente dos dedos pulsados sobre la pantalla. Una vez el reconocedor concreto reconoce el gesto, muestra un rectángulo cuyas dimensiones están determinadas por la posición de los dedos en la pantalla. El usuario puede entonces redimensionar el rectángulo moviendo los dedos para seleccionar los objetos deseados. Una vez el usuario suelta los dedos, si el rectángulo creado contenía algún objeto, entonces sus dimensiones se ajustan al tamaño del grupo.

Si no habían objetos dentro del rectángulo, entonces no se selecciona ninguno y el gesto se cancela, desapareciendo el rectángulo. Mientras está activada la selección múltiple es posible desplazar el rectángulo simplemente arrastrándolo con el dedo, gesto basado en un reconocedor de tipo `UIPanGestureRecognizer`. Este gesto de desplazamiento también está configurado con el mecanismo de autoscroll del gesto de mover objetos, por lo que si se mueve la selección múltiple a un borde de la pantalla, el contenido automáticamente también se desplazará en esa dirección. La figura 14.12 de la página 76 muestra el funcionamiento del proceso de mover el rectángulo de selección.

Durante el estado de selección múltiple es posible añadir o eliminar objetos a la selección simplemente pulsando sobre ellos. Si el objeto estaba seleccionado, entonces pasará a no estarlo, y si es necesario el rectángulo se redimensionará para adaptarse a la nueva cantidad de objetos seleccionados. Si el objeto no estaba seleccionado entonces pasa a estarlo y de nuevo el rectángulo de selección se redimensionará para adaptarse al cambio.

Para cancelar el estado de selección múltiple simplemente hay que pulsar sobre una zona vacía de la pantalla.

Doble pulsación con un dedo:

El comportamiento de este gesto varía en función del estado en el que se encuentre la vista de edición:

- En modo de edición activado este gesto permite crear nuevos objetos, mostrándose una ventana adicional con un listado de los objetos *Pure Data* disponibles (ver fig. 14.2, pág. 66).
- En modo de borrado de cables, este gesto permite cancelar ese estado, volviendo al estado de edición normal.

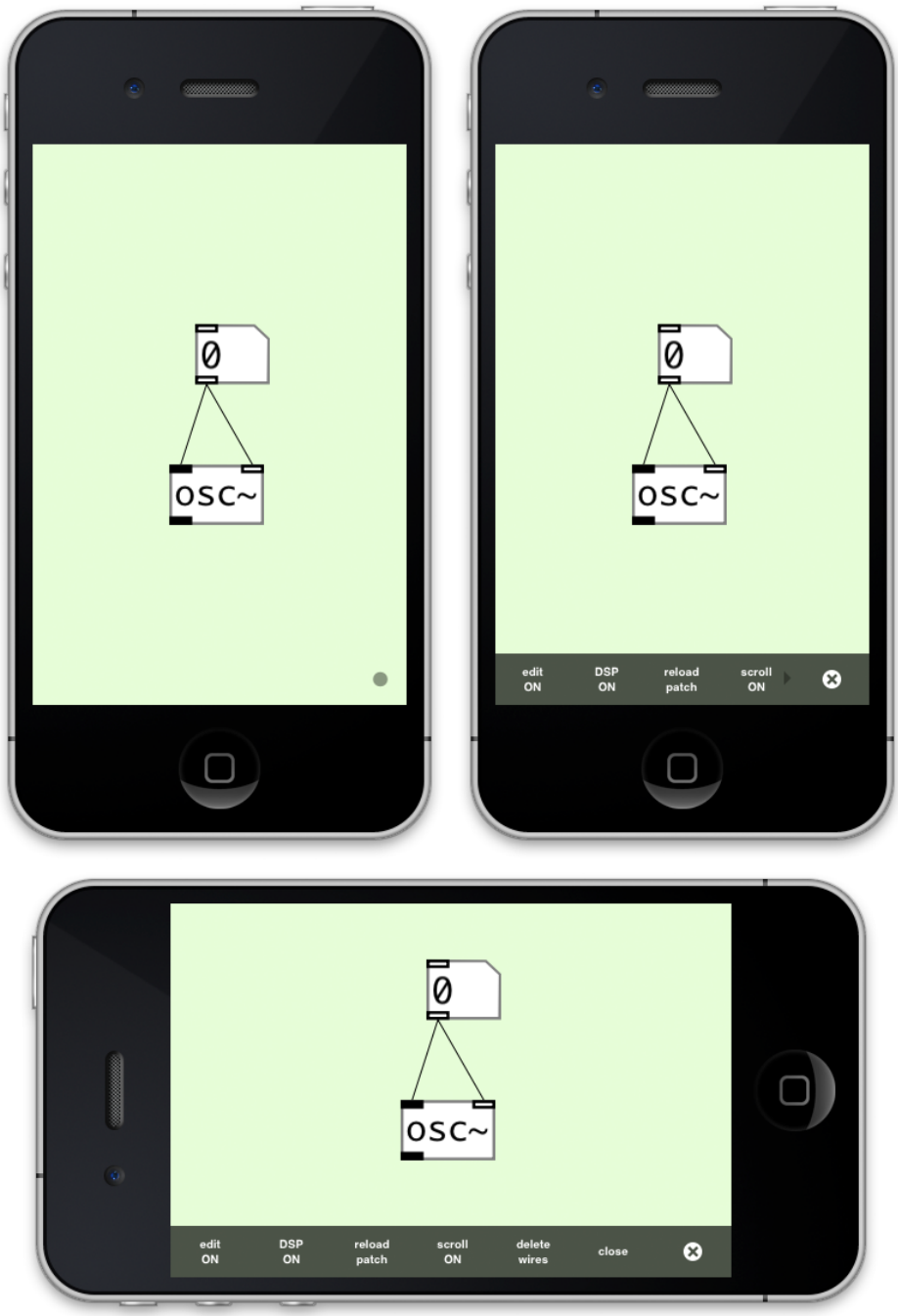

Figura 14.1: Vista Editor de *patches*. Abajo se puede observar cómo la barra de herramientas se adapta a la nueva anchura de la ventana.

Figura 14.2: Vista de creación de objetos. Se accede desde la Vista Edición de *patches*.

Figura 14.3: Vista de edición de objetos. A la derecha se muestra desplegado el teclado numérico. Se accede desde la Vista Edición de *patches*.

Figura 14.4: Vista de edición de las propiedades de texto de objetos *Pure Data*. Se accede desde la Vista Edición de *patches*.

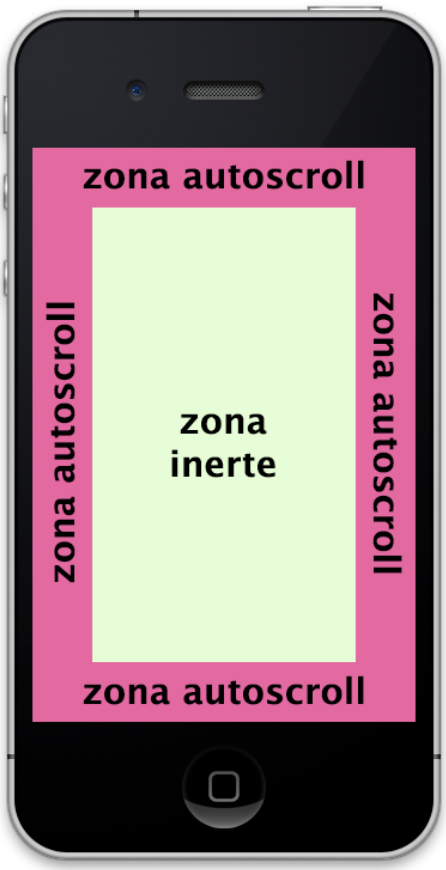

Figura 14.5: Detalle de la zonas inerte y de autoscroll.

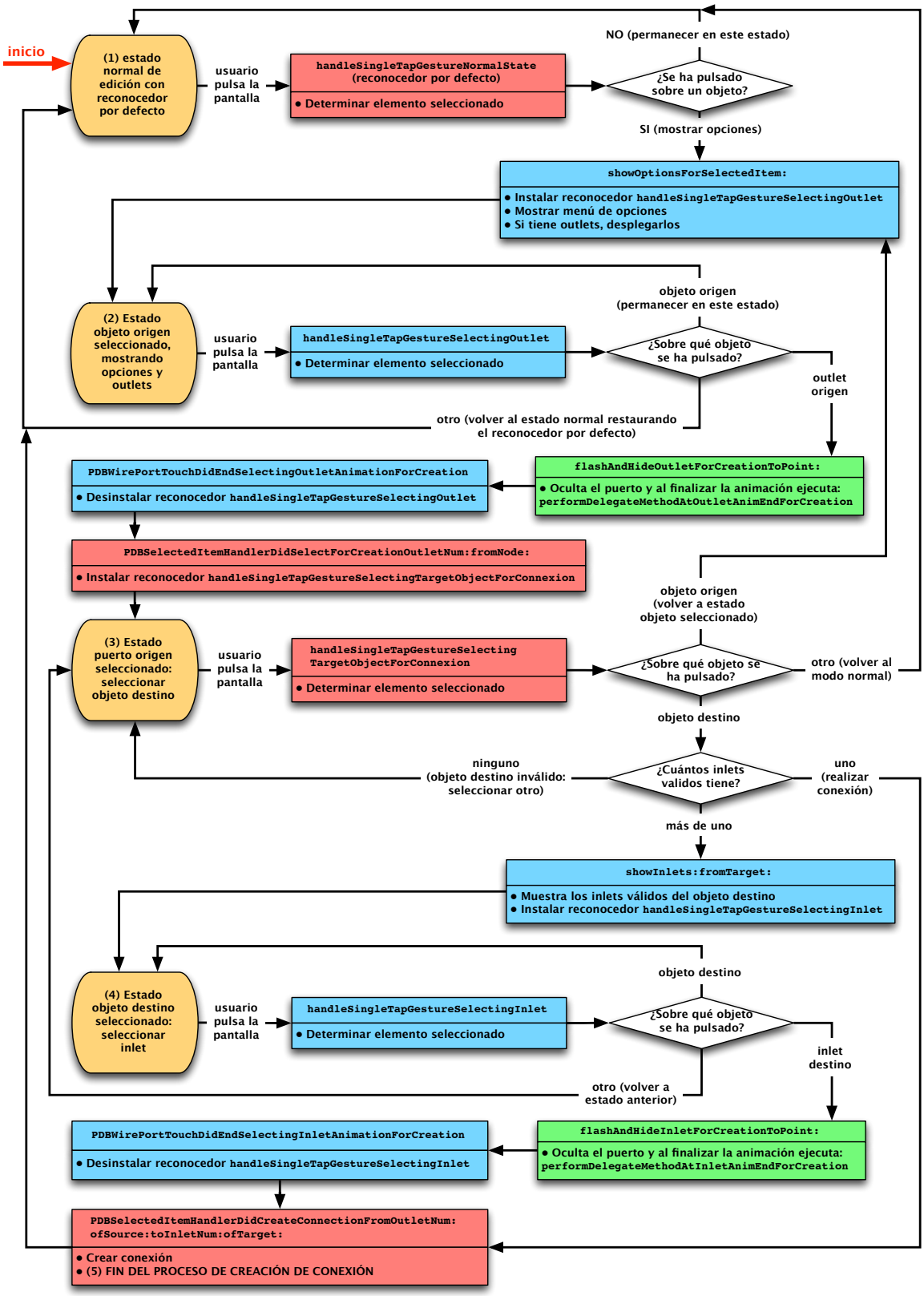

Figura 14.6: Diagrama de flujo del proceso de creación de conexiones.
(ver leyenda en fig. 14.7, pág. 71)

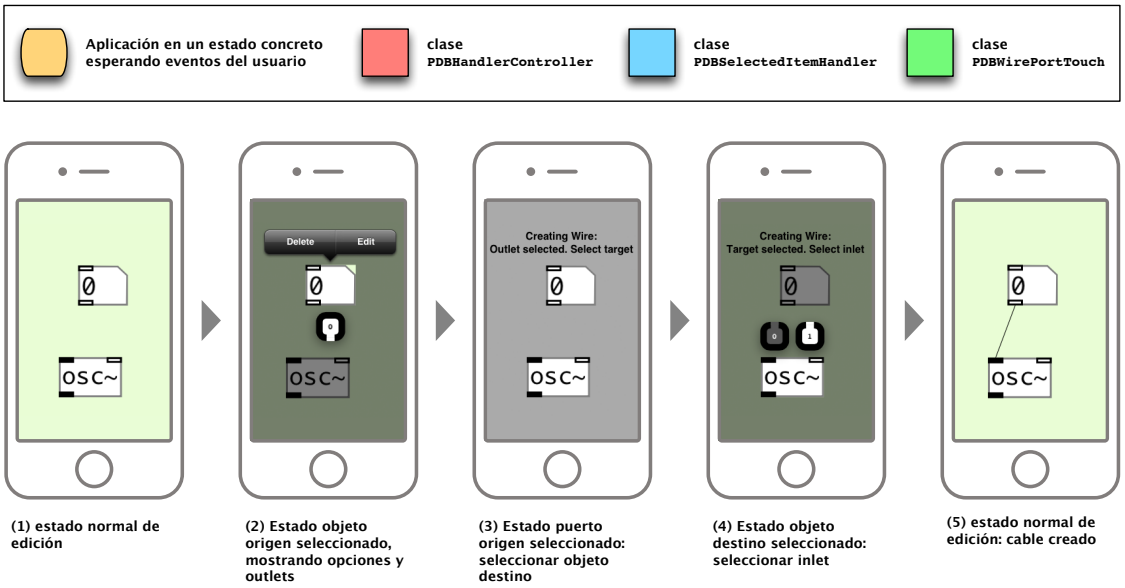

Figura 14.7: Leyenda y detalle de los estados del diagrama de flujo del proceso de creación de conexiones. (ver fig. 14.6, pág. 70)

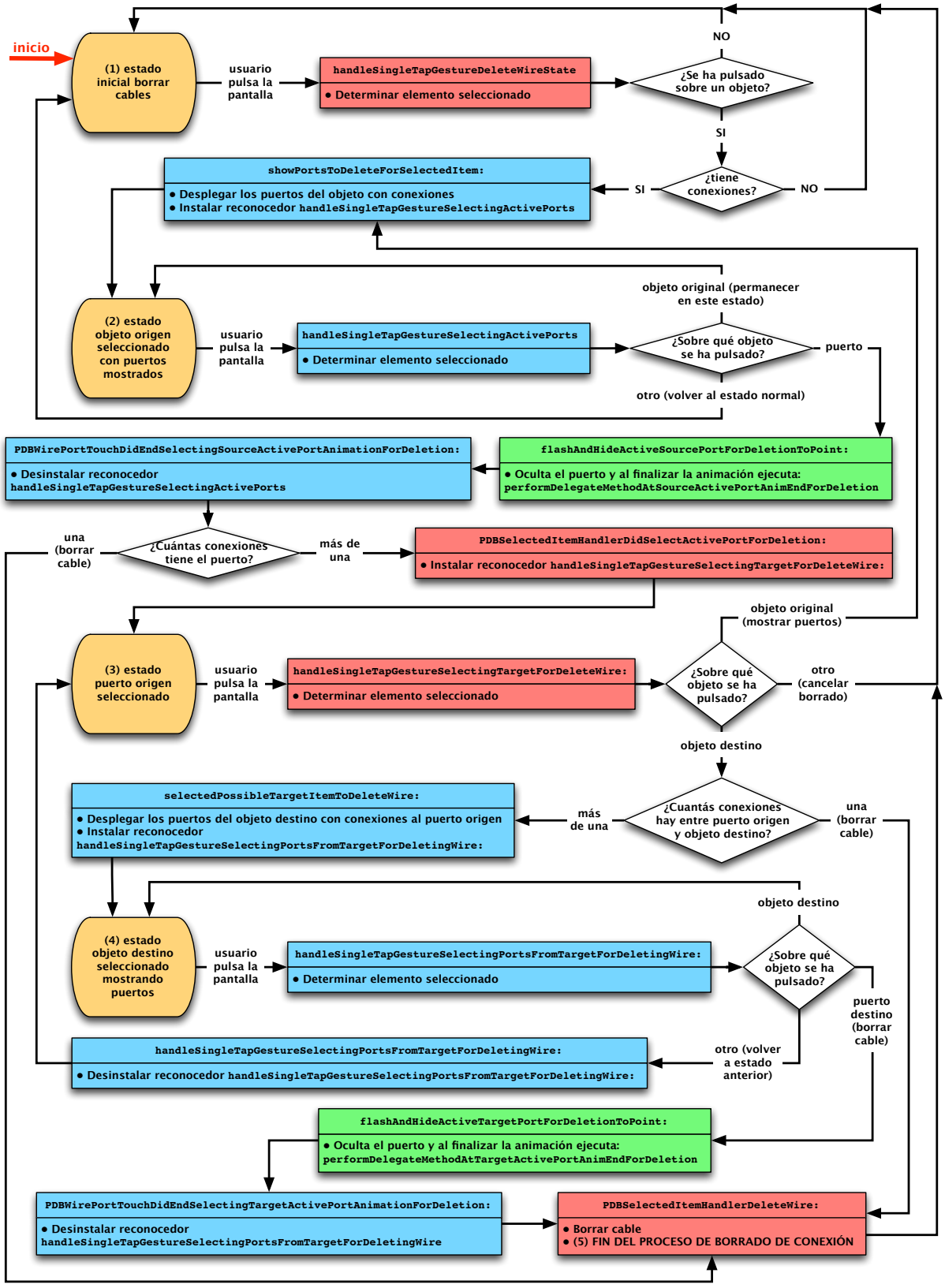

Figura 14.8: Diagrama de flujo del proceso de borrado de conexiones.
(ver leyenda en fig. 14.9, pág. 73)

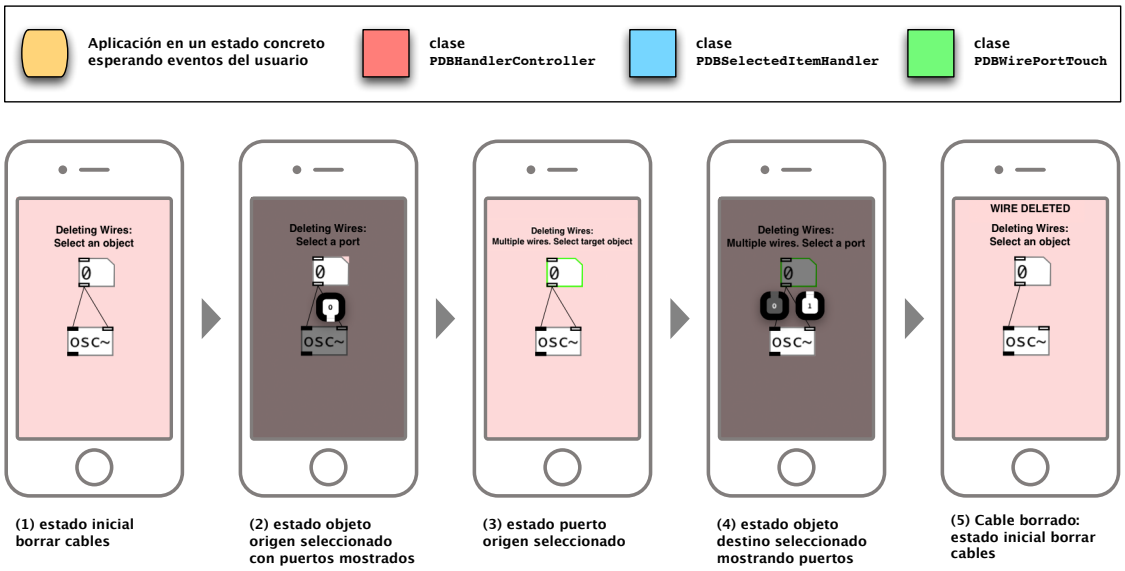

Figura 14.9: Leyenda y detalle de los estados del diagrama de flujo del proceso de borrado de conexiones. (ver fig. 14.8, pág. 72)

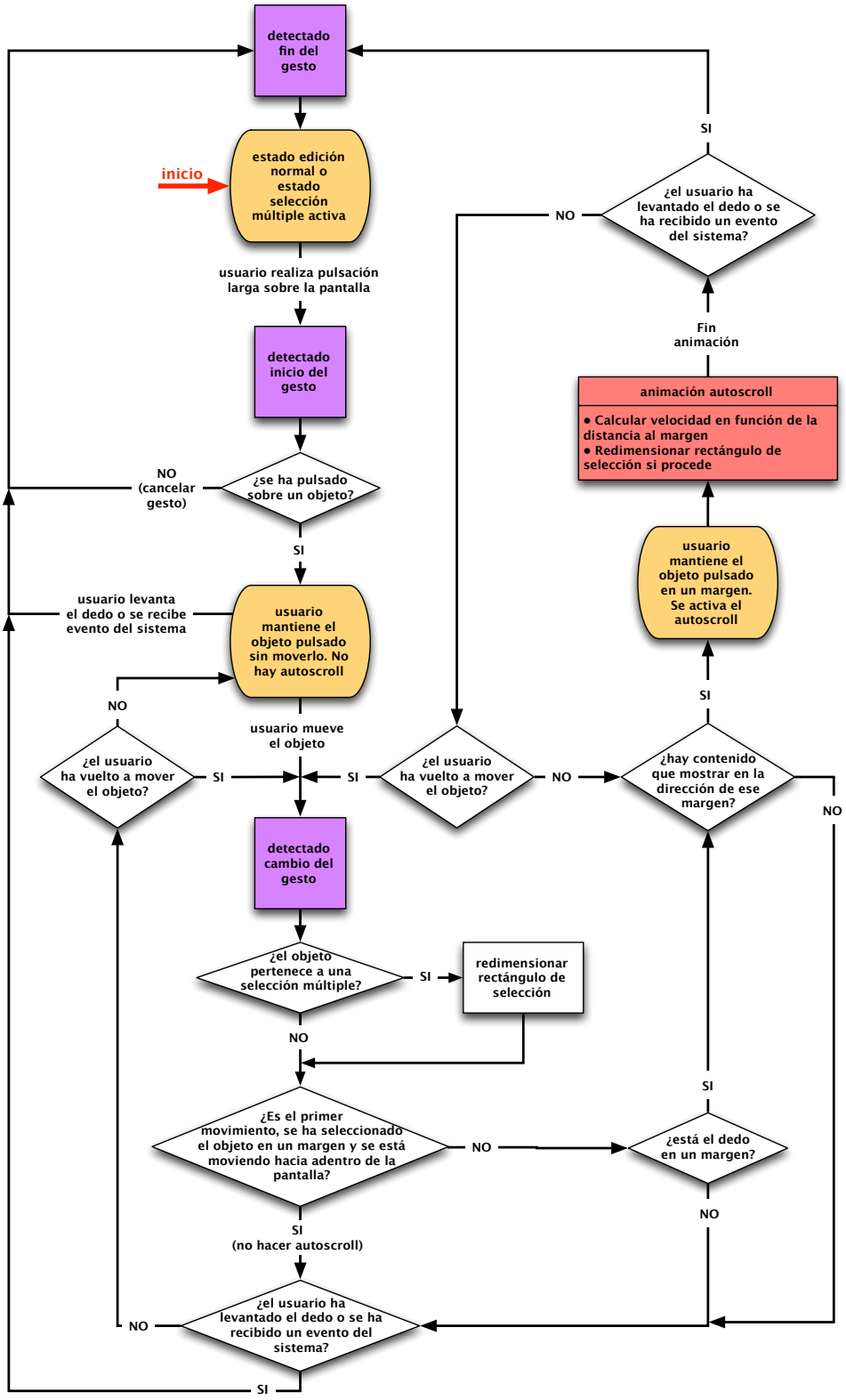

Figura 14.10: Diagrama de fulo del proceso de mover un objeto.

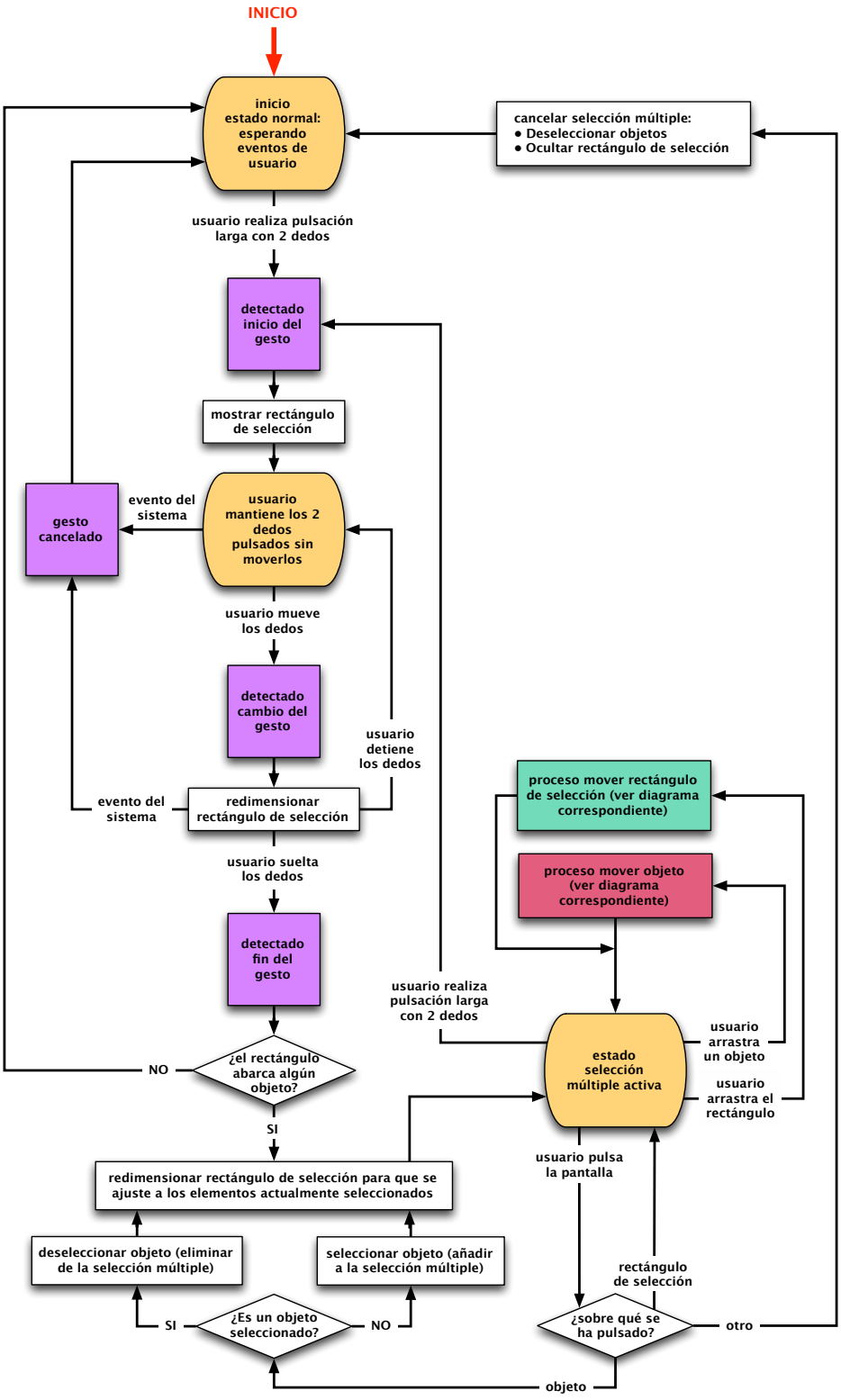

Figura 14.11: Diagrama de flujo del proceso de selección múltiple.

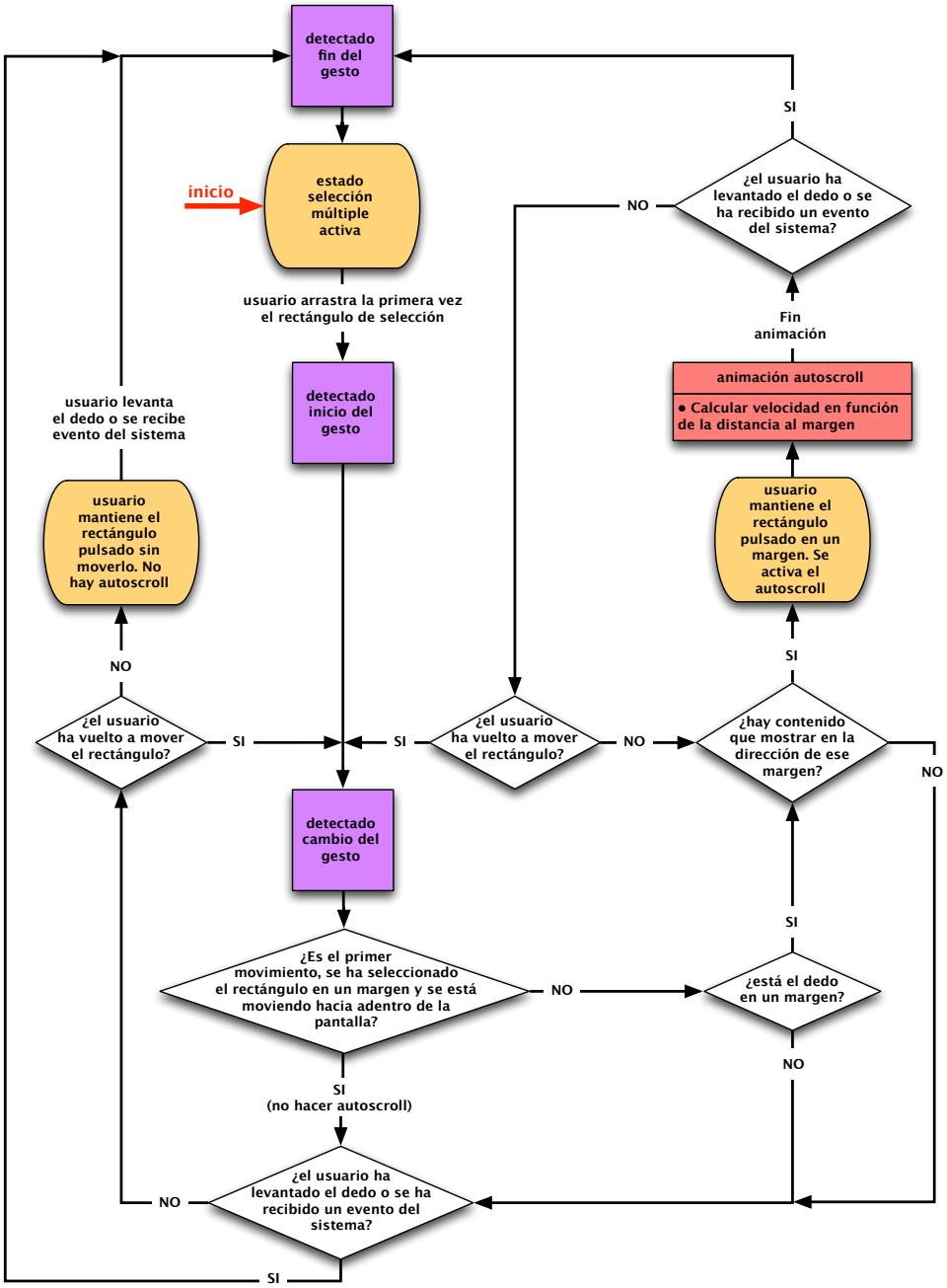

Figura 14.12: Diagrama de flujo del proceso de mover el rectángulo de selección.

15. Documentos Pure Data

15.1. Formato del archivo Pure Data

Pure Data es un lenguaje de programación *orientado a flujo* muy complejo compuesto por una serie de objetos visuales con forma cuadrada que poseen puertos con los que se interconectan los objetos, formando una red por la que fluye y se procesa la información. Aunque este lenguaje se ideó originalmente para el procesamiento de audio, se le han incorporado capacidades gráficas y es capaz además de generar gráficos o controlar imágenes o vídeos. Estos objetos y las conexiones se almacenan en archivos de texto plano con extensión **`*.pd`** consistentes en una serie de registros. Cada registro puede ocupar una o varias líneas, pero todos tienen la misma sintaxis global:

```
#[datos];\r\n
```

donde el carácter `#` indica el inicio del registro, `[datos]` contiene los datos del registro y `;\r\n` indica el final del registro, formado por el carácter punto y coma y los caracteres de retorno de carro.

Los datos del registro consisten en los siguientes campos separados por espacios:

```
#[tipo_general] [tipo_de_elemento] [p1] [p2] [p3] [...];\r\n
```

- `[tipo_general]` es un único carácter con tres posibles valores:

  "N": nueva ventana. Los *patches* pueden contener *subpatches*, que son objetos que contienen un *patch* propio que puede contener a su vez otros objetos. La *app* desarrollada no soporta *subpatches*, por tanto este tipo de registros son ignorados en este caso.

  "X": indica que el registro se refiere a un objeto *Pure Data* o una conexión. Esta aplicación se centra en este tipo de registros.

  "A": el registro es de tipo *data array*. No están soportados en esta *app* y por tanto son ignorados.

- `[tipo_de_elemento]` es el nombre del objeto *Pure Data*. Los objetos se numeran internamente por orden de aparición en el texto del fichero.

- `[p1] [p2] [p3] [...]` son los parámetros requeridos por el objeto *Pure Data*. Difieren de un elemento a otro.

15.2. Cableado

Prácticamente todos los objetos de *Pure Data* pueden conectarse mediante cables. Los objetos poseen puertos de interconexión indicados mediante pequeños rectángulos. Los puertos se clasifican de dos formas:

- Según el tipo de información que transmiten. En *Pure Data* la información que se envía por los cables puede ser de dos tipos:

  Datos de Control: estos datos pueden ser valores numéricos, cadenas de caracteres o listas compuestas por elementos, como números o cadenas de caracteres. Como su nombre indica, sirven para realizar tareas de control. Por ejemplo, para regular el volúmen o para definir la frecuencia en hercios a la que oscilará un objeto de tipo `osc~`. Estos datos son enviados de forma discreta cuando sucede un evento.

  Datos de Señal: contienen las muestras de audio que se transmiten a los objetos. Son realmente los datos que contienen la información de audio. La señal que viaja por estos puertos se envía constantemente en forma de muestras de audio. Para poder escuchar el audio, normalmente el objeto que realiza el último procesamiento de la señal de audio se conecta a objetos de tipo `dac~` encargados de transformar las señales en audio audible (siempre que esté activado el audio en el programa o en el sistema).

- Según si son de entrada o de salida:

  Puertos de entrada En la jerga de *Pure Data* se denominan *inlets* y son los puertos que se encuentran encima del objeto.

  Puertos de salida conocidos en la jerga de *Pure Data* como *outlets*, son los puertos situados debajo de los objetos.

La configuración y el número de puertos varía en cada objeto según sus necesidades. Por ejemplo un objeto de tipo `number` contiene un *inlet* de control —el valor de los datos que recibe por ese puerto se muestran en el objeto— y un *outlet* de *control*, por donde envía el objeto los datos cuando el usuario cambia el valor del objeto; un objeto de tipo `dac~` contiene dos *inlets* de *señal*, uno para reproducir el audio por el canal izquierdo y otro para reproducirlo por el canal derecho —la típica configuración de audio *estéreo*—. Además hay ciertos objetos que poseen puertos variables en función de sus parámetros.

Las conexiones tienen las siguientes particularidades:

- A la hora de realizar una conexión entre dos objetos, primero se selecciona el puerto de salida *outlet* del objeto origen y se finaliza la conexión seleccionando un puerto de entrada *inlet* en el objeto destino.

- Los *outlets* de *control* pueden conectarse a *inlets* tanto de *control* como de *señal*. Sin embargo, los *outlets* de *señal* sólo pueden conectarse a *inlets* de *señal*.

15.3. Tipos de objetos

Los objetos de *Pure Data* consisten en pequeños rectángulos que pueden desplazarse por la pantalla, interconectarse y —según qué tipo— interactuar con ellos para modificar valores.

Los objetos *Pure Data* se clasifican como sigue:

Objetos de tipos de datos básicos Sirven para almacenar tipos de datos básicos y todos poseen la misma configuración de puertos: un puerto de entrada por donde reciben los datos actualizando la información que muestran, y un puerto de salida por donde transmiten el valor almacenado.

number Objeto que permite mostrar valores numéricos y cambiar el valor cuando el usuario pulsa sobre el objeto y arrastra el dedo por la pantalla.

symbol Similar al objeto anterior, salvo que en este caso almacena cadenas de caracteres.

message Este objeto sirve para almacenar listas.

comment Es un objeto especial que no posee puertos y sirve simplemente para escribir comentarios en un *patch*.

Objetos normales Los objetos normales de *Pure Data* son realmente los que procesan o generan la información. Estos objetos consisten en un cuadrado con una configuración de puertos concreta y realizan infinidad de tareas relacionadas con la programación y con el procesamiento de audio. Por ejemplo, el objeto `osc` es un generador de audio que simula un oscilador de onda sinusoidal y puede configurarse y variarse su frecuencia de funcionamiento y la fase de inicio. Para programar hay muchos objetos para simular operaciones condicionales como **`if...else`** o bucles **`for`** o **`while`**, realizar operaciones numéricas (sumar: objeto +, restar: objeto -, multiplicar: objeto *...), etc.

Objetos tipo GUI El usuario puede interactuar con estos objetos para controlar el *patch* y variar su comportamiento en tiempo real. *Pure Data* funciona en dos modos: el modo de edición, que permite crear, mover, conectar... objetos y el modo de reproducción, donde la funcionalidad de edición está desactivada pero se permite interactuar con los objetos —principalmente objetos de tipo GUI— para variar los datos y controlar el *patch*. En esta aplicación se han implementado los dos objetos de tipo GUI más importantes de *Pure Data*:

Bang: Hay un tipo de datos denominado `bang` cuya función es simplemente actuar como disparador de otros objetos. Por ejemplo, al enviar un `bang` a un objeto `number` o a un objeto `message` esto provoca que el objeto receptor extraiga por su puerto de salida el dato que contiene almacenado. Este objeto es simplemente un botón que, al pulsarlo el usuario, envía un dato de tipo `bang` por su puerto de salida.

Vslider, Hslider: Estos objetos son unos deslizadores que el usuario puede controlar —Vslider corresponde a un deslizador vertical y Hslider a uno horizontal—. Pueden editarse parámetros del deslizador como el tamaño y los valores máximo y mínimo que acotan el rango de valores que es capaz de extraer.

Nbx: es una variante del objeto estándar `number` que posee más funcionalidades, pero su funcionamiento es muy similar. En el caso de esta implementación, se tratan de la misma forma tanto el objeto `number` estándar como este objeto `nbx`. En ambos casos internamente se genera un objeto de tipo `nbx`.

15.4. Conexiones y transmisión de datos

Los registros que definen las conexiones tienen el siguiente formato, donde los parámetros son valores numéricos de tipo `integer`:

```
#X connect [obj_origen] [núm_outlet] [obj_destino] [núm_inlet];\r\n
```

`[obj_origen]`: es el objeto origen desde el que se envían los datos. Es un valor numérico que se corresponde con el número de objeto, por orden secuencial, a medida que aparecen en el fichero de texto, comenzando en 0.

`[núm_outlet]`: Los puertos, tanto de entrada como de salida, también están numerados en cada objeto, comenzando en 0. Este es el puerto de salida del objeto origen que envía los datos.

`[obj_destino]`: Es el valor numérico que corresponde con el objeto destino que recibe los datos por la conexión.

`[núm_inlet]`: corresponde al valor numérico del puerto de entrada del objeto destino de la conexión.

En *Pure Data* además es posible transmitir información de forma *inalámbrica* entre objetos usando los objetos normales `send` y `receive`. Esta manera de comunicación emplea nombres de puertos para transmitir. La figura 15.1 muestra dos configuraciones equivalentes para transmitir información entre objetos. Cuando la caja numérica superior varía su valor, en el caso de la izquierda esa información viaja de forma visualmente explícita por los cables; pero en el caso de la derecha la comunicación se realiza de forma inalámbrica: la caja numérica envía los datos al objeto `send`, quien a su vez envía los datos a cualquier puerto cuyo nombre sea `number_rcv`, configurado en este caso como nombre del puerto de entrada del objeto de tipo GUI `nbx`, como puede verse en sus propiedades mostradas en la figura (el objeto `nbx` es una variante de la caja numérica estándar con más opciones).

A continuación se muestra el contenido del fichero `*.pd` correspondiente al *patch* de la figura 15.1. Nótese cómo el nombre de los puertos definidos de forma explicita quedan registrados correspondientemente:

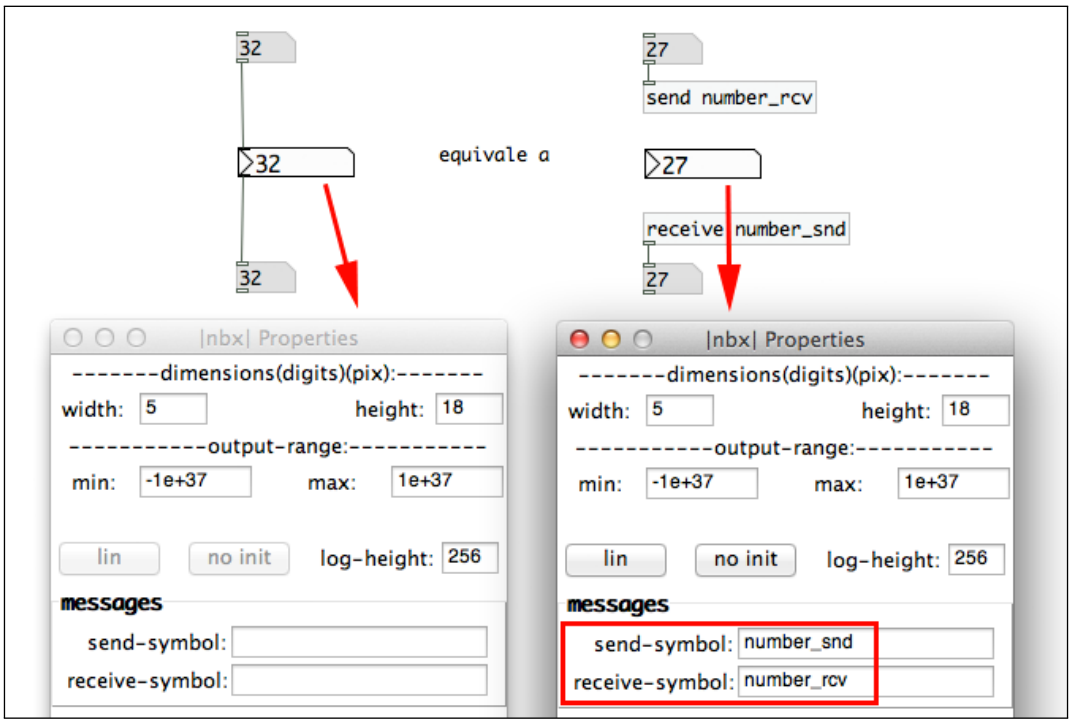

Figura 15.1: Equivalencia entre conexiones cableadas sin emplear nombres de puertos y conexiones inalámbricas mediante nombres de puertos

```
#N canvas 156 176 1330 763 12;
#X floatatom 738 90 5 0 0 0 - - -;
#X floatatom 738 235 5 0 0 0 - - -;
#X obj 739 163 nbx 5 18 -1e+37 1e+37 0 0 number_snd number_rcv empty
0 -8 0 14 -262144 -1 -1 27 256;
#X obj 738 120 send number_rcv;
#X obj 738 203 receive number_snd;
#X floatatom 481 89 5 0 0 0 - - -;
#X floatatom 481 234 5 0 0 0 - - -;
#X obj 482 162 nbx 5 18 -1e+37 1e+37 0 0 empty empty empty 0 -8 0 14
-262144 -1 -1 32 256;
#X text 607 156 equivale a;
#X connect 0 0 3 0;
#X connect 4 0 1 0;
#X connect 5 0 7 0;
#X connect 7 0 6 0;
```

Todos los objetos de tipo GUI (bang, sliders, numbers, etc) permiten definir de forma explícita el nombre de los puertos. En esta *app* se ha aprovechado esta característica de puertos inalámbricos con nombre para poder comunicarse a tiempo real con los objetos del núcleo de *Pure Data*, como se comenta en la sección 16.2.

15.5. Parámetros comunes

Los parámetros de los objetos son comunes según al tipo que pertenezcan. Por ejemplo, todos los objetos poseen parámetros que almacenan sus coordenadas en la pantalla. Los objetos de tipo GUI poseen parámetros para nombrar sus puertos. Esto permite acceder a ellos de forma remota sin usar cables.

El formato de archivo de *Pure Data*, así como los registros y los parámetros se describen en detalle aquí: `http://puredata.info/docs/developer/PdFileFormat`

16. Incorporando la librería libpd

La librería libpd se ha desarrollado principalmente para poder usar *Pure Data* como motor de audio en aplicaciones de dispositivos móviles. Consiste en el núcleo de *Pure Data*, que está desarrollado con el lenguaje *C*, y en una serie de clases adaptadoras escritas en varios lenguajes de programación (*Java, Ojective-C, C++...*) que sirven como API concreta según el sistema en el que se incorpore. En este caso nos centraremos en las clases que adaptan *Pure Data* para poder usarse con *Objective-C*.

16.1. Incluir la librería en el proyecto

La librería libpd puede descargarse gratuitamente archivada en un ZIP en la siguiente página web: `https://github.com/libpd/libpd`

Para incluir la librería libpd a un nuevo proyecto en *XCode* hay que seguir estos pasos, descritos en `http://gitorious.org/pdlib/pages/ObjC`:

- Descargar, descomprimir el archivo ZIP y abrir la carpeta que se genera. Arrastrar el archivo `libd.xcodeproj` desde esa carpeta en el explorador de archivos de *Mac OSX* —el *Finder*— a la zona *Project Navigation* en *XCode*.
- Ir a los ajustes del proyecto, seleccionando en el *Project Navigation* de *XCode* el archivo azul de tu proyecto y luego en la zona de edición seleccionar el *target* principal.
- En la pestaña *Build Phases* desplegar *Target Dependencies*, hacer clic en el botón "+" y añadir libpd en la lista que aparece.
- Estando aún en la pestaña *Build Phases*, desplegar *Link Binary With Libaries*, hacer clic en el botón "+" y añadir los *frameworks* o librerías necesarias. Añadir los *frameworks libpd.a*, *AudioToolbox.framework* y *AVFoundation.framework*.
- En la pestaña *Build Settings*, buscar la entrada *"Header Search Paths"* y añadir la ruta a la carpeta de libpd para que *XCode* pueda encontrar los archivos de cabecera `.h` de libpd.

A partir de ese momento ya se pueden importar y usar las clases *Objective-C* de libpd en el código propio de la *app*.

16.2. Uso de las clases *Objective-C* de libpd

La clase principal es **`PdBase`**, que envuelve la API escrita en *C*, convirtiendo los tipos de datos entre *C* y *Objective-C*, y proporcionando una sincronización básica entre hilos de ejecución. Otras clases importantes son **`PdAudioController`**, que conecta libpd con *Core Audio* —los *frameworks* de programación de sonido de *Cocoa* (ver [**42**])— en el dispositivo *iOS*, y **`PdDispatcher`** que se encarga de gestionar los mensajes desde *Pure Data* al código personalizado del desarrollador: permite comunicarse en tiempo de ejecución con los objetos del núcleo de *Pure Data*.

En el siguiente enlace se describen las clases desarrolladas en *Objective-C* que adaptan *Pure Data* a ese lenguaje: `https://github.com/libpd/libpd/wiki/objc`

Índice de figuras

Bibliografía

A continuación se describen los documentos de *Apple* han sido de imprescindible lectura para la realización de este proyecto.

Documentos básicos

[1] ***Start Developing iOS Apps Today***

- `https://developer.apple.com/library/ios/#referencelibrary/GettingStarted/RoadMapiOS/chapters/Introduction.html`
- Este documento introductorio describe paso a paso los conceptos fundamentales del desarrollo con *Cocoa*. Es el punto de partida conceptual de la documentación, por lo que su lectura es un prerrequisito para el resto de documentos básicos.

[2] ***Cocoa Fundamentals Guide***

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CocoaFundamentals/Introduction/Introduction.html`
- Esta guía describe los conceptos fundamentales del desarrollo con *Cocoa*. Es el punto de partida conceptual del resto de la documentación, por lo que su lectura es un prerrequisito.

[3] ***iOS Technology Overview***

- `https://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html`
- Este documento introductorio describe los aspectos fundamentales de la programación con el SDK de *iOS*.

[4] ***iOS App Programming Guide***

- `https://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html`
- Esta guía describe los fundamentos del desarrollo de *apps iOS* nativas y por tanto es el punto de partida de consulta. Describe la arquitectura fundamental de las *apps*, incluyendo cómo se ajusta el código que escribe el desarrollador con el código proporcionado por *iOS*.

[5] ***iOS Human Interface Guidelines***

- `https://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html`
- Las HIG son una serie de recomendaciones a los desarrolladores para que la experiencia del usuario con la aplicación sea la adecuada. Este documento describe en detalle las HIG de *Apple* para diseñar interfaces de usuario coherentes.

[6] ***Xcode 4 User Guide***

- `https://developer.apple.com/library/ios/#documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/000-About_Xcode/about.html`
- La guía de usuario de *XCode* versión **4**.

[7] ***Tools Workflow Guide for iOS***

- `https://developer.apple.com/library/ios/#documentation/Xcode/Conceptual/ios_development_workflow/00-About_the_iOS_Application_Development_Workflow/introduction.html`
- Esta guía describe el flujo de trabajo requerido para desarrollar y publicar apps *iOS*: registro como desarrollador en *Apple*, configuración de apps, ejecución en el Simulador iOS o en dispositivos reales, etc.

[8] ***Instruments User Guide***

- `https://developer.apple.com/library/ios/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html`
- Esta guía de usuario describe el software *Instruments*, un programa de análisis de rendimiento de aplicaciones incluido en *XCode*. Permite medir aspectos del rendimiento de una aplicación como consumo de memoria, si tiene *fugas* de memoria (objetos no eliminados de la memoria), permite automatizar pruebas de depuración, ejecutar pruebas de *estrés* (llevar la aplicación al límite), etc.

Documentación de *Objective-C*

[9] ***Learning Objective-C: A Primer***

- `https://developer.apple.com/library/ios/#referencelibrary/GettingStarted/Learning_Objective-C_A_Primer/_index.html`
- Breve guía de iniciación al lenguaje de programación *Objective-C* que presenta sus aspectos básicos.

[10] ***Concepts in Objective-C Programming***

- `https://developer.apple.com/library/ios/#documentation/General/Conceptual/CocoaEncyclopedia/Introduction/Introduction.html`
- Una colección de artículos explicando los conceptos principales, los patrones de diseño y los mecanismos de los *frameworks Cocoa* y *Cocoa Touch*.

[11] ***Object-Oriented Programming with Objective-C***

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/OOP_ObjC/Introduction/Introduction.html`
- Describe los conceptos fundamentales de la programación orientada objetos usando *Objective-C*.

[12] ***The Objective-C Programming Language***

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html`
- La documentación oficial de *Objective-C*.

[13] ***Programming with Objective-C***

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html`
- Este documento describe exhaustivamente el lenguaje *Objective-C* a base de ejemplos.

[14] ***Coding Guidelines for Cocoa***

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html`
- Esta guía contiene los convencionalismos de nomenclatura aplicados a las interfaces de programación en *Objective-C*, es decir, describe la forma correcta de nombrar a las *clases*, *métodos*, *protocolos*, etc.

Controladores de Vistas, Vistas y gráficos 2D

[15] ***View Controller Catalog for iOS***

- `https://developer.apple.com/library/ios/#documentation/WindowsViews/Conceptual/ViewControllerCatalog/Introduction.html`
- Describe los distintos tipos de *View Controllers* disponibles en `UIKit`. Es un catálogo que describe unos *View Controllers* especiales que sirven para gestionar otros *View Controllers*: *Navigation Controller*, *Tab Bar Controller*, etc.

[16] ***View Controller Programming Guide for iOS***

- `https://developer.apple.com/library/ios/#featuredarticles/ViewControllerPGforiPhoneOS/Introduction/Introduction.html`
- Describe detalladamente el funcionamiento de los *View Controllers*.

[17] ***View Programming Guide for iOS***

- `https://developer.apple.com/library/ios/#documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/Introduction/Introduction.html`
- Describe de forma detallada el funcionamiento de las vistas de `UIKit`, objetos de la clase *UIView*, que son la base para mostrar contenido en la pantalla del dispositivo.

[18] ***Scroll View Programming Guide for iOS***

- `https://developer.apple.com/library/ios/#documentation/WindowsViews/Conceptual/UIScrollView_pg/Introduction/Introduction.html`
- Las vistas con *scroll* permiten desplazarse por contenido visual demasiado grande para mostrarse al completo en la pantalla, como por ejemplo una tabla o una imagen con el *zoom* ampliado. Este documento describe la tecnología de *scroll* táctil disponible en el *framework* `UIKit`: la clase denominada *UIScrollView*, que es una vista que de forma automatizada permite varios métodos para navegar por el contenido visual, como por ejemplo *arrastrar* el dedo para desplazarse o *pellizcar* con dos dedos parar hacer *zoom*.

[19] ***Table View Programming Guide for iOS***

- `https://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/TableView_iPhone/AboutTableViewsiPhone/AboutTableViewsiPhone.html`
- Describe con todo detalle el funcionamiento de las tablas de `UIKit`, objetos de la clase *UITableView*.

[20] ***Drawing and Printing Guide for iOS***

- `https://developer.apple.com/library/ios/#documentation/2DDrawing/Conceptual/DrawingPrintingiOS/Introduction/Introduction.html`
- Esta guía describe los conceptos básicos del redibujado 2D personalizado de vistas: mapas de bits, gráficos vectoriales, etc.

[21] ***Quartz 2D Programming Guide***

- `https://developer.apple.com/library/ios/#documentation/GraphicsImaging/` `https://developer.apple.com/library/ios/#documentation/GraphicsImaging/Conceptual/drawingwi`
- *Quartz 2D* es un *framework* de bajo nivel también denominado *Core Graphics* (sus clases comienzan por el prefijo *CG*) que proporciona un motor de *render* 2D común en *iOS* y *Mac OSX*. Se recomienda usar siempre que sea posible el *framework* `UIKit`, que proporciona opciones de redibujado de más alto nivel. Sin embargo, es imprescindible para realizar tareas de redibujado no contempladas en `UIKit`.

Implementación de tecnologías *iOS*

[22] ***Event Handling Guide for iOS***

- `https://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/Introduction/Introduction.html`
- Describe la gestión de eventos de usuario como eventos multitoque, eventos de movimiento —detectados por los acelerómetros del dispositivo— y eventos de control multimedia, como por ejemplo al pulsar los botones de los auriculares.

[23] ***Bundle Programming Guide***

- `https://developer.apple.com/library/ios/#documentation/CoreFoundation/Conceptual/CFBundles/Introduction/Introduction.html`

- Las aplicaciones de *iOS* y *Mac OSX* se empaquetan en un único archivo especial que contiene tanto los binarios como los recursos (imágenes, sonidos, etc.). Este documento describe de forma general esta tecnología mostrando la estructura de los *bundles* y las formas de acceder a su contenido. Se recomienda leer además el documento relacionado *Resource Programming Guide*, que contiene información más detallada.

[24] *Resource Programming Guide*

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/LoadingResources/Introduction/Introduction.html`
- Describe detalladamente cómo emplear archivos de recursos como imágenes, texto, sonidos, vídeos, etc. en una aplicación. Se recomienda leer antes la guía *Bundle Programming Guide*, ya que contiene información al respecto más general.

[25] *Preferences and Settings Programming Guide*

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/UserDefaults/Introduction/Introduction.html`
- Describe los mecanismos de *Cocoa* para gestionar las preferencias y los ajustes del usuario.

[26] *Text, Web, and Editing Programming Guide for iOS*

- `https://developer.apple.com/library/ios/#documentation/StringsTextFonts/Conceptual/TextAndWebiPhoneOS/Introduction/Introduction.html`
- Esta guía describe cómo implementar los siguientes conceptos:
  - Gestionar objetos de texto: *UILabel*, *UITextField* y *UITextView*.
  - Mostrar contenido web en una aplicación.
  - Gestionar el teclado: mostrar/ocultar, reorganizar la vista cuando se muestra el teclado, etc.
  - Gestionar operaciones copiar, cortar, pegar.
  - Mostrar y gestionar el menú editar (*edit menu*), un pequeño menú personalizable con las típicas opciones de cortar, copiar, borrar, etc.

[27] *Document-Based App Programming Guide for iOS*

- `https://developer.apple.com/library/ios/#documentation/DataManagement/Conceptual/DocumentBasedAppPGiOS/Introduction/Introduction.html`
- Describe la tecnología de gestión de documentos soportada por el *framework* `UIKit`, donde la clase central encargada de esa tarea es *UIDocument*. Un *documento* es un archivo de datos almacenado de forma persistente en el *sandbox* de la aplicación o en *iCloud*.

[28] *Undo Architecture*

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/UndoArchitecture/Articles/UndoManager.html`
- Describe cómo implementar la tecnología de deshacer/rehacer de *Cocoa*.

[29] ***Uniform Type Identifiers Overview***

- `https://developer.apple.com/library/ios/#documentation/FileManagement/Conceptual/understanding_utis/understand_utis_intro/understand_utis_intro.html`
- La tecnología descrita en este documento permite el intercambio de datos y documentos entre *apps* y otros servicios del sistema. Por ejemplo, cuando se abre un correo con un archivo adjunto, al pulsar sobre el archivo adjunto se mostrará una lista de todas las *apps* que pueden manejar ese tipo de archivo o documento.

[30] ***Internationalization Programming Topics***

- `https://developer.apple.com/library/ios/#documentation/MacOSX/Conceptual/BPInternational/BPInternational.html`
- Describe cómo preparar una aplicación para ser traducida a varios idiomas.

[31] ***Developing for the App Store***

- `https://developer.apple.com/library/ios/#documentation/General/Conceptual/ApplicationDevelopmentOverview/Introduction/Introduction.html`
- Este documento te va guiando por los procesos necesarios del desarrollo de apps para la A*App Store*, la tienda de aplicaciones para *iOS*.

[32] ***Dashcode User Guide***

- `https://developer.apple.com/library/ios/#documentation/AppleApplications/Conceptual/Dashcode_UserGuide/Contents/Resources/en.lproj/Introduction/Introduction.html`
- Esta guía describe cómo desarrollar aplicaciones web que se ejecutan en el navegador web de *iOS* denominado *Mobile Safari*.

Conceptos específicos del desarrollo con *Cocoa*

[33] ***Collections Programming Topics***

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/Collections/Collections.html`
- Describe las estructuras de datos básicas de *Cocoa*: *arrays*, *diccionarios*, *conjuntos*, etc.

[34] ***Number and Value Programming Topics***

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/NumbersandValues/Articles/Values.html`
- Describe las clases envoltorio de tipos de datos básicos de *Cocoa*.

[35] ***String Programming Guide***

- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/Strings/introStrings.html`

- Describe todo lo relacionado con el manejo de cadenas de caracteres al programar con *Cocoa*.

[36] ***File System Programming Guide***
- `https://developer.apple.com/library/ios/#documentation/FileManagement/Conceptual/FileSystemProgrammingGUide/Introduction/Introduction.html`
- Describe el sistema de ficheros de *iOS* y cómo se emplean las *interfaces* del sistema para acceder a ficheros y directorios.

[37] ***Key-Value Observing Programming Guid***
- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html`
- Describe la implementación del patrón Observador empleada en *Cocoa*. Este mecanismo permite notificar a unos objetos cuando las *propiedades* (los datos internos) de otros objetos cambian.

[38] ***Date and Time Programming Guide***
- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/DatesAndTimes/DatesAndTimes.html`
- Describe las clases que gestionan la fecha y hora en *Cocoa*.

[39] ***Timer Programming Topics***
- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/Timers/Timers.html`
- Describe la programación de temporizadores, que sirven para retardar o realizar de forma periódica ciertas acciones.

[40] ***Property List Programming Guide***
- `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/PropertyLists/Introduction/Introduction.html`
- Describe la tecnología de ficheros de *Lista de Propiedades*, que son unos ficheros en lenguaje XML ampliamente empleados por *Apple* para almacenar datos y valores estructurados de forma persistente.

[41] ***System Messaging Programming Topics for iOS***
- `http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/SystemMessaging_TopicsForIOS/Articles/SendingaMailMessage.html`
- Es un breve documento que explica los detalles de implementación de las tecnologías de *Apple* para enviar SMS y emails.

[42] ***Core Audio Overview***
- `https://developer.apple.com/library/ios/#documentation/MusicAudio/Conceptual/CoreAudioOverview/Introduction/Introduction.html`
- Describe cómo usar las tecnologías de audio de *iOS* y *Mac OSX*. Las interfaces de software de audio las proporciona *Core Audio*, que es un conjunto de *frameworks* separados.

Glosario

bug Error, fallo o defecto en un programa informático normalmente debido a errores humanos producidos durante el desarrollo del programa.

bundle Tecnología empleada en las aplicaciones de *Mac OSX* e *iOS* que consiste en empaquetar juntos los archivos ejecutables y los recursos del programa: imágenes, sonidos, textos, etc. dando la sensación al usuario de que la aplicación es un único archivo.

Cocoa Entorno de aplicación de *Apple* compuesto por un conjunto de *frameworks* que proporcionan un entorno de ejecución para aplicaciones que funcionan bajo *Mac OSX* e *iOS* (en este último sistema se denomina concretamente *Cocoa Touch*).

delegación patrón de diseño de software donde un objeto actúa en nombre de, o coordinado con otro objeto.

Entorno de desarrollo integrado Entorno de programación (compilador, editor, depurador, etc.) empaquetado en un programa informático.

Espacio de Nombres Mecanismo que incorporan los entornos de desarrollo de software que permiten aislar partes del proyecto evitando colisiones en los nombres de las clases y otros elementos.

Foundation *framework* de *Apple* que define la capa base de las clases de *Objective-C*, proporcionando clases para el objeto base, de estructuras de datos y de tipos de datos básicos, como strings.

framework Esquema definido para el desarrollo de una aplicación, proporcionándole una estructura global que modela las relaciones generales de las entidades del dominio, imponiendo un diseño al programa y haciendo que sea más sencillo trabajar con tecnologías complejas. Para ello proporciona un conjunto de herramientas reutilizables como librerías, programas, recursos, etc.

Human Interface Guidelines Son una serie de recomendaciones que se ofrecen a los desarrolladores para que la experiencia de los usuarios de la aplicación sea la adecuada, siendo el objetivo principal crear una experiencia sólida y consistente en el sistema operativo en el que se desenvuelve la aplicación. Parte de este objetivo se consigue usando diseño visual homogéneo, compartiendo el

aspecto de elementos como botones, cajas de texto, iconos, etc. entre todas las aplicaciones.

inlet Término empleado en la jerga de *Pure Data* para designar a los puertos de entrada de los objetos.

Interface Builder Aplicación de *Apple* suministrada con el IDE *XCode* para el diseño de GUIs.

Interfaz gráfica de usuario Programa informático que proporciona un entorno visual como interfaz de usuario usando imágenes y elementos gráficos para mostrar la información y realizar acciones manipulando las imágenes.

iOS Sistema operativo de *Apple* basado en *Mac OSX* instalado en sus dispositivos táctiles.

libpd Librería de *Pure Data* para externalizar su funcionamiento hacia dispositivos móviles.

Mac OSX Sistema operativo de *Apple* basado en *UNIX* instalado en sus ordenadores personales.

Navigation Controller Es un *View Controller* especializado que gestiona la navegación de otros *View Controllers* organizados jerárquicamente y almacenándolos en una pila, mostrando una barra de navegación en la parte superior de la pantalla.

Objective-C Lenguaje de programación ampliamente empleado por *Apple* que añade funcionalidad orientada a objetos al lenguaje *C*.

outlet Término empleado en la jerga de *Pure Data* para designar a los puertos de salida de los objetos.

patch Una ventana individual de programación de Pure Data.

protocolo Mecanismo del lenguaje de programación *Objective-C* que define una interfaz que otros objetos deben implementar, por ejemplo un *delegado*, compuesto por una lista de declaraciones de métodos sin estar ligados a la definición de una clase.

Pure Data Lenguaje de programación orientado a flujo diseñado para generar audio.

sandbox mecanismo de seguridad informático que aísla a los procesos entre sí restringiendo su ámbito de actuación, de forma que un fallo en un programa no pueda propagarse a otros programas o al sistema operativo.

Storyboard Funcionalidad de *Apple* que permite a los desarrolladores de aplicaciones para *iOS* diseñar toda la GUIs de la aplicación estableciendo relaciones entre *View Controllers*.

string Tecnicismo empleado en programación para denominar a las cadenas de caracteres de texto.

UIKit *framework* de *Apple* que proporciona las clases para crear las GUIs de *iOS*. Proporciona un objeto que representa a la aplicación, gestión de eventos, dibujado, ventanas, vistas y controles diseñados específicamente para interfaces basadas en pantallas táctiles.

Uniform Type Identifier Tecnología de *Apple* empleada en los sistemas *iOS* y *Mac OSX* que permite importar y exportar datos entre las *apps*.

vanilla Versión estándar de un programa informático tal y como se distribuyó originalmente, sin modificaciones aplicadas por terceras partes. El nombre surgió porque el helado estándar típico era de sabor a vainilla.

View Controller Clase del *framework* `UIKit` de *Apple* que se enmarca en la capa de *Controlador* del patrón MVC empleado en el desarrollo de aplicaciones para *iOS*.

XCode Es el IDE suministrado por *Apple* que permite desarrollar software para dispositivos.

Siglas

API *Application Programming Interface*, Interfaz de programación de aplicaciones.

GUI *Graphical User Interface*, Interfaz gráfica de usuario.

HIG Human Interface Guidelines, Directrices de interfaz humana.

IDE *Integrated Development Environment*, Entorno de desarrollo integrado.

MVC *Model View Controller*, Modelo Vista Controlador.

SDK *Software Development Kit*, Kit de desarrollo de software.

UTI Uniform Type Identifier.