UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

FisioClinicApp: Aplicación Android para la gestión de una clínica de fisioterapia

TITULACIÓN: Graduado en Ingeniería en Tecnologías de la Telecomunicación
AUTOR: Jorge Hernández Ríos
TUTORA: Dra. Carmen Nieves Ojeda Guerra

FECHA: Junio 2017

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

FisioClinicApp: Aplicación Android para la gestión de una clínica de fisioterapia

HOJA DE FIRMAS

Autor:

Fdo: Jorge Hernández Ríos

Tutora:

Fdo: Dra. Carmen Nieves Ojeda Guerra

Fecha: Junio 2017

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

FisioClinicApp: Aplicación Android para la gestión de una clínica de fisioterapia

HOJA DE EVALUACIÓN

Calificación: _____

Presidente/a:

Fdo:

Vocal:

Fdo:

Secretario/a:

Fdo:

Fecha: Junio 2017

SUMMARY

Nowadays, mobile devices have had a huge impact on our lives. Today, they are our digital tool in which we spend more time, especially when it comes to the use of mobile applications. But this impact has not only come to our way of communicating, but also to become important work tools in the business, educational or health sector. Engineers have adapted to this change, using mobile applications as a new solution to solve people's problems. In this way, application development and engineering process are based on the analysis of those problems and then making a mobile application which can solve those issues.

These days, in the majority of companies, management tasks are performed with a computer. Mobile devices are a step further in terms of efficiency and productivity, thanks to advantages such as mobility, cost optimization or the use of the cloud for the availability of information, among others. In particular, the health sector is in a growing trend in the use of mobile devices. It is known as "mHealth" or mobile health, and there are about 100,000 mobile applications with practices such as patient tracking, clinical data collection or even vital signs monitoring.

However, within the health field, in the physiotherapy sector in Spain, the impact of mobile devices has not been appreciated. Taking into account these factors: a growing use of mobile devices for management tasks and a sector with little use of mobile applications, even less for management of their clinics, we get *FisioClinicApp*, an Android application for physiotherapists, that allows patients management, events planning, having a to-do list and register payments of a clinic of physiotherapy, using technologies in the cloud. Every physiotherapist has his/her own user account in his/her own use of *FisioClinicApp* for his/her clinic.

Every task, previously mentioned, is in a *navigation menu*, so the physiotherapists can access to each section of the application easily. Also there is one section where users can manage their personal data and may change their name, e-mail, phone number or password. The development of this application is achieved because of a good design and programming practices, giving priority to research and analyze the identified problem with a high degree of detail. Moreover, it is used an architecture pattern called *Model-View-Presenter* (MVP). The use of this pattern ensures the decoupling of all parts of the application and therefore its maintenance for future possible changes.

For the management of user accounts and the database in the cloud, it is used *Firebase*, a platform in the Google cloud. Taking into account the sensitivity of the information stored in this application, the *Data Protection Act* must be complied, and all information stored in the database will be encrypted to ensure privacy. *FisioClinicApp* stands as an application specifically designed for physiotherapists which want to manage the physiotherapy clinic which they belong.

An extended description of this summary is presented throughout this Final Degree Project, which is organized into parts, chapters and sections for easy un-

derstanding. All the relevant aspects of *FisioClinicApp* development, an application developed by the student Jorge Hernández Ríos, wtih his advisor Carmen Nieves Ojeda Guerra, from the Telecommunications and Electronics Engineering School at ULPGC, are further detailed in this project.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Parte I

Memoria

Capítulo **1**

INTRODUCCIÓN

1.1 Introducción

El *Trabajo Fin de Grado* que se plantea en este documento presenta el desarrollo de una aplicación móvil para terminales Android, que permita facilitar las tareas de gestión que se tienen en una clínica de fisioterapia. Tareas que van desde las típicas de gestión, como guardar eventos en un calendario, hasta el almacenamiento de fichas de cada uno de los pacientes, donde se pretende tener información personal de los mismos, imágenes como radiografías, anotaciones de las sesiones, entre otros.

Hoy en día la mayoría de tareas de gestión dentro de una empresa se encuentran informatizadas, haciendo uso de ordenadores para ello. Con la aplicación móvil objeto de este trabajo se pretende dar un paso más allá en términos de eficiencia y productividad, aprovechando las ventajas que aportan los dispositivos móviles. Si bien en la sanidad general se está comenzado a utilizar más este tipo de tecnologías, no lo se está haciendo tanto en el área de la fisioterapia (como se mostrará posteriormente). Ante esta situación, el Trabajo Fin de Grado que se propone tiene en cuenta estos factores y aporta una solución beneficiosa para los profesionales de la fisioterapia.

1.2 Antecedentes

En la actualidad, el crecimiento del uso de las tecnologías móviles ha dado lugar a que se apueste también por este tipo de tecnologías en el ámbito empresarial. El ordenador supuso una revolución en cuanto a las tareas de gestión para una empresa, pero aún mayor ha sido la revolución de las aplicaciones móviles para este uso, por las ventajas que aportan a las empresas que utilizan una aplicación móvil para su gestión. Algunas de estas ventajas, de las que se pretende aprovechar la aplicación objeto de este trabajo, son:

- **Disponibilidad.** El uso de servidores *"en la nube"* permite que la información esté disponible las 24 horas del día, mejorando así la eficiencia y productividad de la empresa.

- **Movilidad.** Posibilidad de realizar cualquier tarea necesaria en cualquier lugar, dentro del mismo local de la empresa o fuera de ella.
- **Personalización.** Pese a que muchas tareas de gestión son típicas dentro de cualquier empresa, el diseño de una aplicación puede ser personalizado, teniendo en cuenta sus características concretas.
- **Todo en uno.** Existencia de un punto común para todo el trabajo necesario, sin necesidad de tener un gran conjunto de aplicaciones para cada tarea distinta.
- **Optimización de costes.** Se pueden disminuir costes de hardware con políticas como [1] *"Bring Your Own Device"* (*Trae Tu Propio Dispositivo*), ya que los empleados pueden utilizar su propio dispositivo para estas tareas; disminuyendo así el gasto energético y costes fijos de la oficina.

Por otro lado, en el sector de la medicina, se puede apreciar que con el paso de los años se ha aumentado el uso de las aplicaciones móviles como herramientas de apoyo en el día a día. Así, se habla de que, uno de cada dos médicos de familia hoy en día, es considerado como *"e-doctor"* [2], ya que hacen uso de ordenadores, *smartphones* o tabletas para acceder a información sanitaria. Esta tendencia actual se conoce como *mHealth* [3], o salud móvil, y engloba todas las prácticas médicas que se pueden realizar desde un terminal móvil, desde el seguimiento de pacientes, recopilación de datos clínicos o incluso monitorización de signos vitales.

Tal ha sido este impacto que se habla de 100.000 aplicaciones de este tipo en las principales tiendas de aplicaciones, donde el 70 % van destinadas al paciente y el 30 % restante a los profesionales. En España se prevé que estas aplicaciones de salud generarán un negocio de 4.000 millones de euros, sobre todo gestionando enfermedades crónicas. Incluso la *mHealth* ya tiene un rol importante en la MWCB (*Mobile Web Capital Barcelona*) [4].

Sin embargo, en el área dentro de la sanidad que nos compete para el trabajo propuesto, la *fisioterapia*, no se ha notado tanto el impacto de las tecnologías móviles (según el estudio de aplicaciones subidas a las tiendas online y que luego se comenta) y se necesita, por tanto, un esfuerzo en el desarrollo de aplicaciones destinadas a los profesionales fisioterapeutas. Debido a esto, se propone la realización de una aplicación móvil, para ayudar a estos profesionales a hacer de manera rápida, donde y cuando quieran, tareas cotidianas relacionadas con su profesión, que son laboriosas y precisan de un ordenador.

1.3 La tecnología móvil

Tal y como se ha mencionado anteriormente, las tecnologías móviles se han posicionado como uno de los elementos prácticamente imprescindibles de nuestras vidas. Los siguientes datos lo demuestran [5]:

- En España hay más teléfonos móviles inteligentes que ordenadores: un 80 % de los españoles tiene un *smartphone*, mientras que sólo un 73 % tiene ordenador.
- En el mundo, más de la mitad de las visitas que reciben los grandes buscadores proceden de un terminal móvil.
- Un 62 % del tiempo invertido en el mundo online, se realiza desde *smartphones* y tabletas.
- En el mundo, el uso de las aplicaciones (*apps*) ya supone el 54 % del tiempo gastado en el mundo digital.

Por tanto, se está hablando de unos dispositivos que hace años únicamente se utilizaban para llamar o mandar mensajes SMS, y hoy en día se tienen unos impresionantes dispositivos de propósito general, que pueden: tomar fotografías de alta calidad, navegar por Internet, escuchar música, ver películas, chatear con amigos, cuidar la salud, entre otros; y esto es sólo pensando en personas "de a pie", sin embargo, el impacto también ha sido importante en otros sectores de la sociedad, como el ámbito empresarial, educativo o sanitario.

En términos de hardware, estos dispositivos han crecido de manera exponencial. La tecnología en general está creciendo a pasos agigantados, tal como enuncia la Ley de Moore [6]. Así, se está dando lugar a una evolución brutal en cuanto a prestaciones y calidad, aunque los temores de la obsolescencia están presentes.

Para dar un rendimiento óptimo a todo este potente hardware que se tiene hoy en día, es necesario un software a la par, que garantice una eficiente gestión de los recursos y la implementación de las tareas en estos dispositivos. Por ello, en la actualidad existen dos sistemas operativos para terminales móviles dominantes en el mercado: *Android* de Google [7] e *iOS* de Apple [8]. El dominio de estos sistemas operativos es tan grande que, en España [9], según el primer estudio de Kantar del año 2017, llegan a cubrir el 99,5 % del total de los dispositivos, dejando en un residual 0,5 % restante al tercer sistema operativo relevante, que es *Windows Phone*, tal y como se puede apreciar en la figura 1.1.

Teniendo en cuenta estos datos, se pueden comprender porque la aplicación propuesta en este Trabajo Fin de Grado se debe realizar en el sistema operativo Android.

Por otro lado, estos sistemas operativos cuentan con diferentes plataformas para los desarrolladores de aplicaciones móviles, ofreciendo numerosas funcionalidades con el objetivo de facilitar el diseño y desarrollo de aplicaciones. Además, existen diferentes lenguajes de programación para llevar a cabo la implementación de las mismas: *Java* en el caso de dispositivos *Android*, y *Objective-C* y *Swift* para sistemas que utilizan *iOS*. Teniendo en cuenta que a lo largo de esta carrera el lenguaje más utilizado para las diversas asignaturas ha sido Java, refuerza la idea de realizar este Trabajo Fin de Grado en el sistema operativo Android. Asimismo, ambas empresas (Google y Apple) cuentan con plataformas de distribución de aplicaciones móviles:

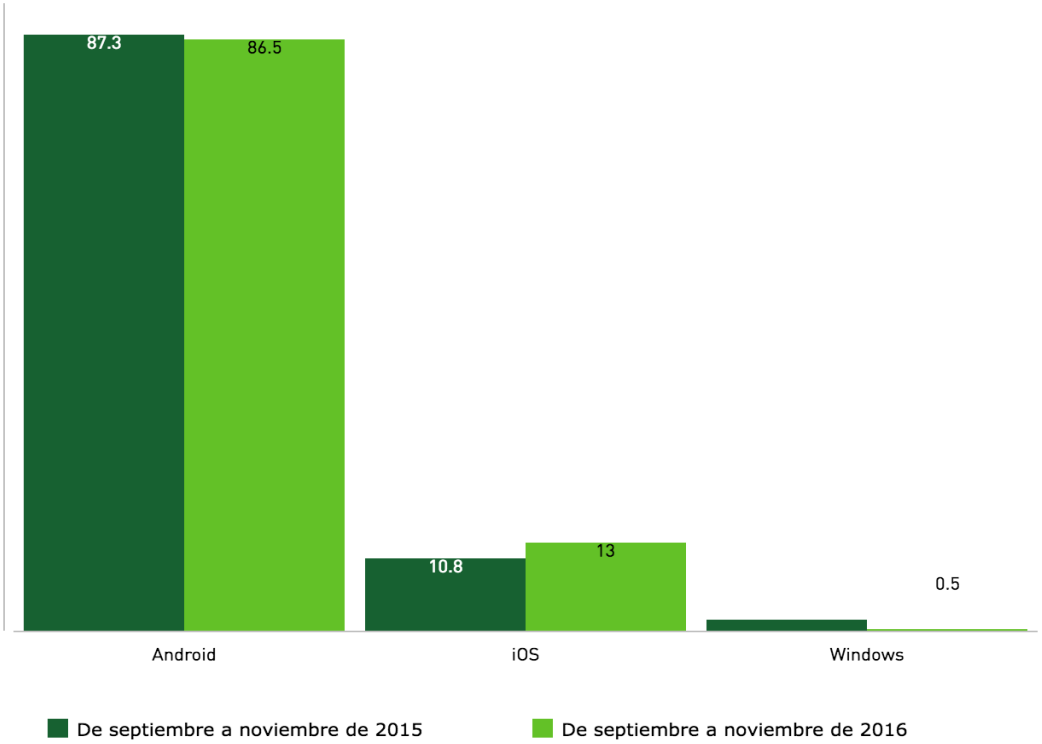

Figura 1.1: Uso de los sistemas operativos para terminales móviles en España

Google Play [10] en el caso de Google, para sus aplicaciones Android, y *App Store* [11], en el caso de Apple para sus aplicaciones iOS.

Dentro del mercado actual de aplicaciones móviles dedicadas a la fisioterapia, la mayoría tienen un carácter didáctico-educativo y sólo algunas tienen alguna similitud con la presentada en este Trabajo Fin de Grado. En la tabla 1.1 se muestran las aplicaciones más importantes dentro de la tienda *Google Play* relacionadas con el área de la fisioterapia. La aplicación propuesta en este trabajo pretende contar con las mismas funciones que las ofertadas en las aplicaciones similares, pero mejorándolas, y añadiendo más funciones interesantes para los profesionales. Además, pretende mejorar la interfaz de usuario respecto a estas aplicaciones homónima, en términos de diseño y usabilidad.

Por otro lado, la aplicación propuesta necesitará hacer uso de una base de datos para almacenar la información con la que se trabaja en ella. Con el objetivo de que esta información sea accesible a los distintos dispositivos, sea móvil o tableta, de los diferentes profesionales con los que cuenta una clínica de fisioterapia, se hará uso de la plataforma *Firebase* de Google [12], la cual tiene entre sus servicios bases de datos en tiempo real "en la nube". Asimismo, esta plataforma, al ser de Google, puede ser accedida desde los terminales Android sin problema. En esta línea, hay que tener en cuenta que se está tratando, no sólo con información personal, sino además bastante delicada, ya que hablamos de la salud de los pacientes de la clínica, por lo que habrá que respetar su privacidad. En este aspecto será preciso cumplir con la Ley Orgánica

Nombre de aplicación	**Descripción**
Fisioterapia a tu alcance	Muestra de ejercicios para rehabilitaciones.
TestOpos Fisioterapia	Tests para oposiciones de fisioterapia.
Anatomia: Atlas de músculos	Guía didáctica anatómica.
I-FISIO	Gestión de pacientes y citas de una clínica de fisioterapia.
Fisio en tu móvil	Aplicación dedicada de la empresa *FisioHogar* para que los clientes puedan realizar consultas y concertar citas.
Fisio Tools Gratis ES	Ayuda a realizar cálculos de terapia respiratoria y ventilación mecánica.
Apuntes Fisioterapia	Apuntes para alumnos universitarios del Grado en Fisioterapia.
Fisio 2	Solicitar cita en la clínica de fisioterapia *Fisio 2*.
Guía de Actos Fisioterapicos	Guía didáctica sobre actuación fisioterapéutica.
Fisio Activo	Pedir cita en la clínica de fisioterapia *Fisio Activo*.

Tabla 1.1: Aplicaciones en español relacionadas con la fisioterapia en *Google Play*

15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal [13], y la plataforma *Firebase* cuenta con mecanismos de seguridad para ello. En este aspecto, para un mayor entendimiento, se recomienda la lectura del Apéndice A, *Seguridad y privacidad de la información*.

1.4 Objetivo

El objetivo final a conseguir con este Trabajo Fin de Grado será desarrollar una aplicación sobre dispositivos móviles Android que permita a profesionales del sector de la fisioterapia contar con un entorno que les facilite las tareas de gestión de una clínica de fisioterapia (figura 1.2), posibilitando las siguientes acciones:

1. Registrarse/loguearse y desconectarse de la aplicación.
2. Consultar, añadir, editar y eliminar pacientes en un archivo de pacientes.
3. Buscar un determinado paciente del archivo de pacientes.
4. Visualizar dentro de la ficha de cada paciente información personal del mismo, como su nombre y apellidos, teléfono o correo electrónico. También podrá visualizar una galería de imágenes dedicada a ese paciente, además de poder tomar anotaciones, por ejemplo, de sus sesiones con el mismo.

5. Disponer de un calendario, en el que poder acceder a cada uno de los días del mismo.
6. Crear, editar y eliminar eventos al acceder a un día determinado del calendario.
7. Añadir y eliminar tareas en un apartado dedicado.
8. Registrar los pagos realizados por los diferentes pacientes, y también los pendientes de pago.
9. Acceder a un panel de usuario donde poder cambiar su contraseña o poder darse de baja de la aplicación.
10. En caso de tener el rol de administrador dentro de la clínica, se puede eliminar usuarios de la aplicación (en caso de que algún empleado sea despedido o haya abandonado dicha clínica).

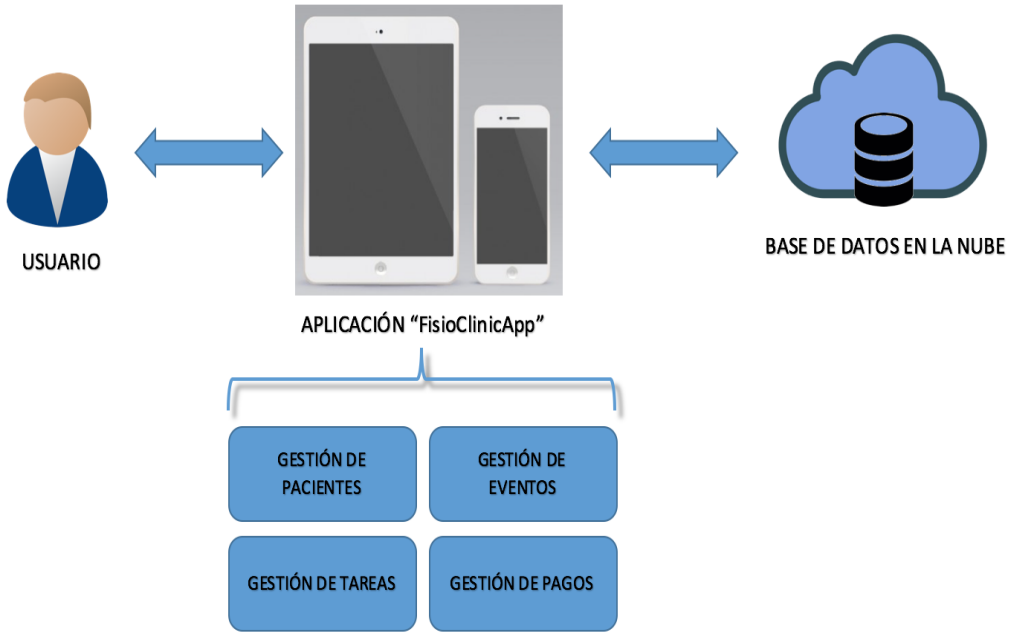

Figura 1.2: Gráfico objetivo de la aplicación

Inicialmente la aplicación pedirá al usuario de la misma registrarse/loguearse. Una vez hecho esto, se accederá a la aplicación como tal, pudiendo elegir entre distintas acciones, como son: *Inicio*, *Pacientes*, *Calendario*, *Tareas*, *Pagos* y *Mi usuario*.

Así, una vez que se introduce el *login*, el usuario (empleado de la clínica de fisioterapia) encontrará cada día los eventos registrados, para poder conocer nada más abrir la aplicación qué trabajo tiene durante ese día (acción *Inicio*). A través, de la acción *Pacientes* podrá ver todos los pacientes registrados en la aplicación,

pudiendo añadir, editar, buscar y eliminar pacientes. Podrá acceder a cada uno de ellos, donde contará con tres apartados: uno dedicado a la información personal del mismo, otro para poder ver imágenes como radiografías o fotografías del paciente, y otro para tomar anotaciones.

En lo que respecta a la acción *Calendario*, el usuario podrá navegar por un calendario, accediendo a cada uno de los días del año y podrá ver qué eventos tiene registrados, pudiendo añadir, editar o eliminar los eventos en un día específico.

Por otro lado, en la acción *Tareas* se permite crear tareas que corresponden con las diferentes labores que se deban hacer dentro de la clínica de fisioterapia, quedando así en esta lista como recordatorios para los empleados de la misma. Una vez realizadas, se podrán dar por finalizadas en el mismo apartado. De manera similar funciona la acción *Pagos*, donde se podrá registrar pagos realizados, además de pagos pendientes, los cuales pasarán a la lista de pagos realizados una vez esto ocurra.

Por último, también se permite al usuario de la aplicación, gestionar su cuenta de usuario (mediante la acción *Mi usuario*), donde podrá cambiar su contraseña si lo desea, además de poder darse de baja de la aplicación por cualquier motivo.

Para realizar esta aplicación, que sea fácil de usar y mantener, se deberá cumplir los siguientes objetivos parciales:

O1. Analizar los requisitos de usuario de la aplicación.

O2. Implementar una interfaz de usuario que siga los principios de usabilidad enunciados por Jakob Nielsen [14] y realizar un test de usabilidad que será probado por usuarios finales de la aplicación.

O3. Usar una metodología de desarrollo de software para implementar la aplicación en el lenguaje Java para Android, que utilice una arquitectura completamente desacoplada, de forma que cualquier cambio importante sólo afecte a la parte cambiada, no al resto de la aplicación. Este objetivo se subdivide en:

- Especificar el diseño del código.
- Especificar el modelo de datos.
- Implementar la aplicación.

O4. Implementar una batería de test del modelo de datos para comprobar el acceso correcto a los datos (que es la parte más crítica de la aplicación).

1.5 Peticionario

El Trabajo Fin de Grado presentado en esta memoria ha sido elaborado para cumplir el Real Decreto 1993/2007, de 29 de octubre (BOE del 30 de octubre), por el que se establece la ordenación de las enseñanzas universitarias oficiales, y su actualización en el Real Decreto 861/2010, de 2 de julio, que indica que "...todas las enseñanzas oficiales concluirán con la elaboración y defensa de un Trabajo Fin de Título que ha de formar parte del plan de estudios y estar orientado a la evaluación de las competencias asociadas al título."

1.6 Organización de la memoria

La memoria que analiza la información y desarrollo de este trabajo se divide en seis capítulos, el pliego de condiciones y el presupuesto. A continuación, se comenta brevemente los capítulos incluidos:

- El capítulo 1 realiza una introducción a la tecnología propia de la programación de aplicaciones para móviles, los antecedentes del trabajo y especifica el objetivo principal del trabajo.
- El capítulo 2 amplía la información de la aplicación delimitando el sistema y buscando la funcionalidad que debe ofrecer desde la perspectiva del usuario final. Presenta los actores, casos de uso y la interfaz de usuario de la aplicación.
- El capítulo 3 muestra las especificaciones detalladas de todos los objetos, incluyendo sus características y operaciones. Asimismo, se analiza el modelo de datos.
- El capítulo 4 utiliza el resultado del capítulo anterior para generar el código final en el lenguaje de programación elegido.
- El capítulo 5 verifica si el resultado es el que se quería, mediante una serie de pruebas sobre la interfaz de usuario y el modelo de la aplicación.
- El capítulo 6 incluye una serie de conclusiones del trabajo final y las posibles mejoras del mismo.
- Por último, se adjunta el pliego de condiciones que especifica qué condiciones y requisitos ha de satisfacer la aplicación; y el presupuesto, que detalla los costes totales del software generado.

Capítulo 2

MODELO DE REQUISITOS

2.1 Introducción

Un modelo, en el desarrollo de software, define cómo solucionar los problemas que aparecen en el desarrollo de una aplicación. Para desarrollar el software, existen diferentes metodologías, como la *Objectory* [16], que definen los distintos tipos de modelos que se pueden encontrar en este proceso. Así, los modelos básicos son: **de Requisitos, de Diseño, de Implementación y de Pruebas.** Asimismo, el desarrollo de software debe ser registrado a lo largo de todo el proceso, dando lugar a distintos documentos **(Documentación)**. De todos los modelos indicados, en este capítulo se tratará el de requisitos, como primer modelo a definir en el desarrollo de la aplicación objeto de este Trabajo de Fin de Grado.

El **modelo de requisitos** [17] tiene como objetivo delimitar el sistema y capturar la funcionalidad que debe ofrecer desde la perspectiva del usuario (es el contrato entre el desarrollador y el usuario final). El modelo de requisitos, que se va a presentar en este capítulo, está basado en *el modelo de casos de uso*. Este modelo es el primero en desarrollarse, ya que es la base para la formación del resto de los modelos.

El modelo de requisitos consiste, básicamente, de tres modelos principales:

- **Modelo de comportamiento:** se basa directamente en el modelo de casos de uso y especifica la funcionalidad, desde el punto de vista del usuario. Tiene dos conceptos claves:
  - *Actores*: representan los distintos papeles que los usuarios pueden jugar en el sistema.
  - *Casos de uso*: representa qué pueden hacer los actores con respecto al sistema.
- **Modelo de presentación o de interfaces o de borde:** especifica cómo interactúa el sistema con actores externos al ejecutar los casos de uso, es decir, especifica cómo se verán las interfaces gráficas y que función tiene cada una de ellas.
- **Modelo de información o del dominio del problema:** conceptualiza el sistema según los objetos que representan las entidades básicas de la aplicación.

Esta separación en tres modelos diferentes es importante para conseguir una mayor estabilidad en el desarrollo del sistema, permitiendo minimizar los efectos de cada uno sobre los otros dos. En este Trabajo Fin de Grado se presentan los dos primeros sub-modelos de requisitos: *Modelo de comportamiento* y *Modelo de interfaces*, para caracterizar la aplicación que se va a desarrollar. El tercer sub-modelo se sustituye por el estudio de los diagramas de secuencia que se hará en el capítulo siguiente (*Modelo de diseño*). Inicialmente, y antes de realizar estos sub-modelos, se realiza una descripción del problema, tal y como se le haría a un usuario de la aplicación.

2.2 Descripción del Problema

La descripción del problema es una definición muy inicial de las necesidades, que sirve como punto de partida para comprender los requisitos del sistema, es decir, debe ser una descripción de lo que se necesita y no una propuesta de solución.

En este Trabajo de Fin de Grado se pretende desarrollar una *Aplicación Android para la gestión de una clínica de fisioterapia*. Esta aplicación permite al usuario del dispositivo, que serán los profesionales de la clínica en cuestión, realizar las diferentes tareas cotidianas que se llevan a cabo dentro de una clínica de fisioterapia. Estas tareas son tales como organizar un calendario en el que anotar eventos como citas, apuntar tareas a realizar de manera interna, llevar un registro de los pacientes con sus correspondientes datos, citas o notas sobre sus sesiones, o anotar los pagos pendientes y realizados.

Actualmente, si se analiza el día a día dentro de una clínica de fisioterapia, se comprueba que ha habido avances en lo que respecta a tareas de gestión, ya que prácticamente la totalidad se encuentran informatizadas. Sin embargo, se requiere de un ordenador, que es un elemento estático dentro de la oficina, y de diversas aplicaciones para cada una de las tareas. No es un problema grave como tal, pero con la aplicación objeto de este trabajo se aporta una mejora en eficiencia y productividad, aprovechando la movilidad que aportan los dispositivos móviles, la disponibilidad de la información las 24 horas del día y en cualquier lugar al contar con servicios en la nube, y el aglutinar todas las aplicaciones necesarias en una única.

Basándonos en estas virtudes y las tareas mencionadas previamente, la aplicación contará con una serie de apartados para dar cabida a cada una de ellas, pudiendo navegar entre ellas a través de un menú de navegación global. Estos apartados principales tienen los siguientes nombres: *Pacientes, Calendario, Tareas* y *Pagos*.

Así, la aplicación tendrá una pantalla de entrada que permite *loguearse* con una cuenta de usuario ya existente. Al realizar el *login*, se accederá a la aplicación como tal, donde contará con una pantalla que le muestre los diferentes eventos fijados para tal día. Esta pantalla inicial está considerada también como un apartado de

la aplicación, llamado *Inicio*, al cual se puede acceder desde el menú de navegación global de la aplicación. Haciendo uso de él, podrá acceder a las secciones de la aplicación comentadas anteriormente, además de una dedicada a la gestión de su cuenta de usuario, denominada *Mi usuario*. En definitiva, estos son los apartados de la aplicación y su finalidad:

- **Inicio:** muestra las eventos registradas para el día actual.
- **Pacientes:** contendrá cada una de las fichas de los pacientes de la clínica, con su información personal, imágenes y anotaciones.
- **Calendario:** permite acceder a cada uno de los días del año, pudiendo observar qué eventos están planificados, además de poder crear nuevos, editarlos e, incluso, eliminarlos.
- **Tareas:** habilita una lista de tareas a realizar de manera interna dentro de la clínica, además de poder darlas por finalizadas posteriormente.
- **Pagos:** sirve para anotar los pagos realizados por los clientes, pero también los que quedan pendientes por realizar.
- **Mi usuario:** permite cambiar información de la cuenta de usuario, además de darse de baja de la aplicación.

2.3 Modelo de Comportamiento

El **modelo de comportamiento** [17] describe las diferentes formas de uso de un sistema, en el que cada una de esas formas se conocen como *caso de uso*. Cada caso de uso se compone de una secuencia de eventos iniciada por el usuario. Para comprender los casos de uso del sistema, es necesario saber cómo los usuarios lo van a usar o *actor* (el actor no corresponde directamente con un usuario). Como se muestra en la figura 2.1 el actor y el caso de uso representan los dos elementos básicos de este modelo.

Figura 2.1: Representación de las entidades básicas para el modelo de comportamiento

2.3.1 Actores

Los **actores** [17] corresponden con el papel que un usuario puede jugar dentro de la aplicación (son entidades distintas a los usuarios). Los actores modelan cualquier entidad externa al sistema, además no están restringidos a ser personas físicas, pudiendo representar otros sistemas externos al actual. Cada uno de estos actores podrá ejecutar una o más tareas del sistema.

Los actores se identifican antes que los casos de uso, para que éstos sean la herramienta principal para encontrar los casos de uso. Al definir todos los actores y los casos de uso se define la funcionalidad completa del sistema.

A la hora de definir los actores, se identifican primero aquellos que son la razón principal del sistema, conocidos como **actores primarios**, que son los que rigen la secuencia lógica de ejecución del sistema. Además existen otros actores que supervisan y mantienen el sistema, conocidos como **actores secundarios**. Éstos corresponden, por lo general, a máquinas o sistemas externos.

Para especificar los actores de la aplicación, se pueden diferenciar los siguientes: un actor primario definido como *Usuario*, que es el encargado de interactuar con la aplicación, pudiendo crear, leer, modificar y eliminar (funciones CRUD, *Create-Read-Update-Delete*) información con la que se trabaja en la aplicación (citas, fichas personales de los clientes, tareas, pagos, etc). Esta información se encontrará almacenada en una *Base de datos*, por tanto, será un *actor secundario* a tener en cuenta. En la figura 2.2 se presenta un diagrama que representa al sistema como una "caja negra" y los diferentes actores como entidades externas a él.

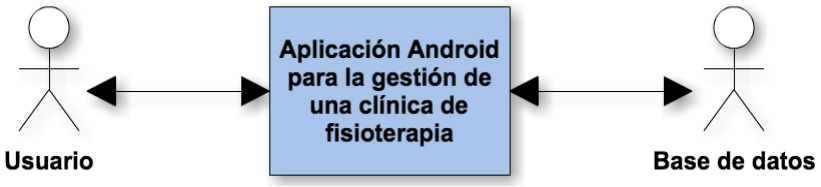

Figura 2.2: Delimitación del sistema para la aplicación *FisioClinicApp*

2.3.2 Casos de uso

Después de haber definido los actores se define la funcionalidad del sistema a través de los **casos de uso**. Cada caso de uso constituye un flujo completo de eventos que muestra la interacción entre los actores y el sistema (es decir, qué van a hacer los actores). El actor primario es el encargado de empezar esta interacción y los casos de uso se muestran como respuesta al evento anterior. La ejecución del caso de uso acaba cuando algún actor genera un evento que requiere un caso de uso nuevo.

En la *Aplicación Android para la gestión de una clínica de fisioterapia* que se presenta, se utiliza al *Usuario* (actor primario) como punto de partida. Así, se pueden definir los casos de uso principales (figura 2.3):

- *Acceso a pacientes.*
- *Acceso a calendario.*
- *Acceso a tareas.*
- *Control de pagos.*
- *Manejo datos usuario.*

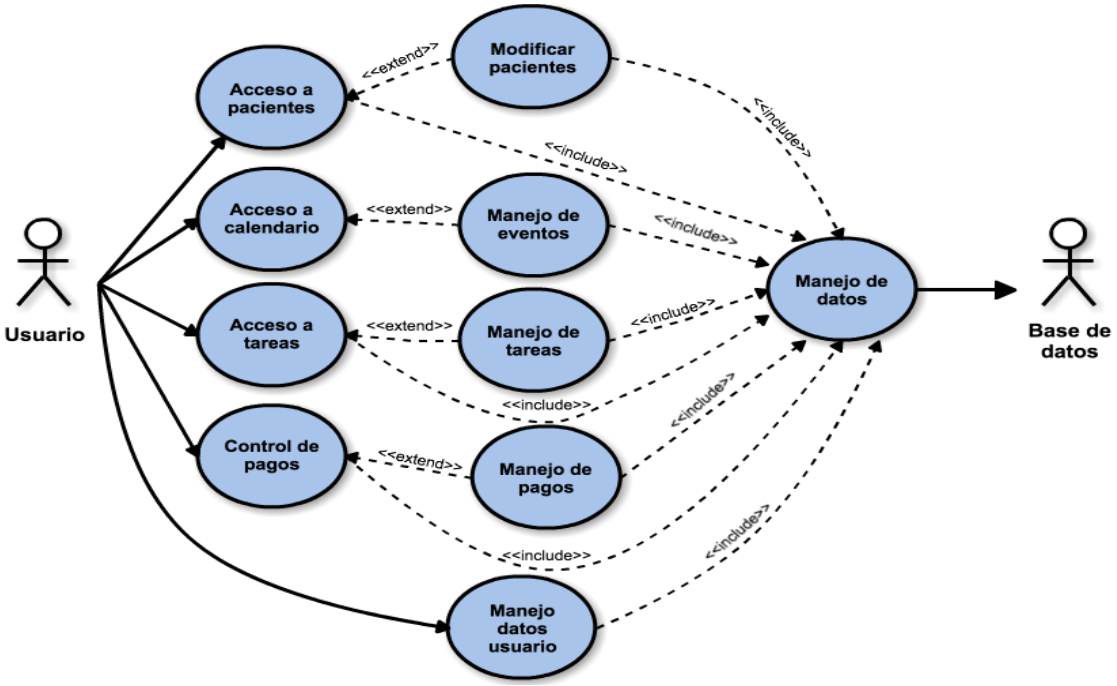

Figura 2.3: Diagrama de *Casos de Uso* para la aplicación *FisioClinicApp*

Aunque la idea es mantener el modelo de casos de uso lo más sencillo posible, existen ocasiones en los que la funcionalidad debe ser puesta en casos de uso separados o bien realizar una subdivisión del caso de uso. Existen dos enfoques para expresar variantes:

1. Si las diferencias entre los casos de uso son pequeñas, se pueden definir *subflujos* separados dentro de un mismo caso de uso. En la sección 2.5.2 se muestran los flujos y subflujos de los casos de uso de la aplicación.
2. Si las diferencias entre los casos de uso son grandes, se deben describir como casos de uso separados. Para estos casos de uso se utilizan principalmente las relaciones de *extensión*, *inclusión* y *generalización*.

2.3.2.1. Extensión

La **extensión** se utiliza para modelar secuencias de eventos opcionales de casos de uso que, al manejarse de manera independiente, pueden ser insertados o eliminados. Especifica cómo un caso de uso puede *insertarse* en otro para extender la función del anterior. El caso de uso donde se va a insertar la nueva funcionalidad debe ser independiente del caso de uso insertado. El caso de uso original se ejecuta hasta donde se inserta el nuevo caso de uso. Después de que este nuevo caso de uso haya terminado su función, el curso original de la secuencia continúa como si nada hubiera ocurrido.

Como se muestra en la figura 2.3, la notación para extensión es la etiqueta «*extend*». Así, en la aplicación objeto de este trabajo se pueden ver los siguientes casos de uso con el tipo *extensión*:

- *Modificar pacientes*: extiende de *Acceso a pacientes* ya que es un caso de uso que no se dará siempre que el actor primario acceda a la lista de pacientes de la aplicación, sino cuando éste lo decida. Este caso de uso será seleccionado por el usuario pero a partir del caso de uso *Acceso a pacientes*.
- *Manejo de eventos*: extiende de *Acceso a calendario* ya que es un caso de uso que no se dará siempre que el actor primario acceda al calendario de la aplicación, sino cuando éste seleccione un día concreto del calendario. Este caso de uso será seleccionado por el usuario pero a partir del caso de uso *Acceso a calendario*.
- *Manejo de tareas*: extiende de *Acceso a tareas* ya que es un caso de uso que no se dará siempre que el actor primario acceda a las tareas de la aplicación, sino cuando éste quiera añadir, modificar o eliminar una tarea concreta. Este caso de uso será seleccionado por el usuario pero a partir del caso de uso *Acceso a tareas*.
- *Manejo de pagos*: extiende de *Control de pagos* ya que es un caso de uso que no se dará siempre que el actor primario acceda al control de pagos de la aplicación, sino cuando éste quiera añadir, modificar o eliminar el pago de un cliente. Este caso de uso será seleccionado por el usuario pero a partir del caso de uso *Control de pagos*.

2.3.2.2. Inclusión

La **inclusión** se define como una sección de un caso de uso que es parte obligatoria del caso de uso básico. El caso de uso que se va a insertar depende del caso de uso anterior. Como se muestra en la figura 2.3, la notación para inclusión es la etiqueta «*include*».

Así, en la aplicación objeto de este trabajo se pueden ver que existe un caso de *inclusión*: *Manejo de datos*. Este caso de uso es el encargado de las acciones oportunas para el manejo de los datos almacenados en la *Base de datos*. No puede ser seleccionado por el actor primario sino que va incluido en el resto de casos de uso.

2.3.2.3. Generalización

Una relación adicional entre casos de uso es la **generalización** que apoya la reutilización de los casos de uso. Mediante esta relación es necesario describir las partes similares una sola vez, en lugar de repetirlas para todos los casos de uso de comportamiento común. En la generalización hay dos tipos de casos de uso:

- A los casos de uso que se extraen se les llama casos de uso *abstractos*, ya que sirven para describir partes similares que son comunes a otros casos de uso (no se instancian).
- A los casos de uso que realmente son instanciados se les conoce como casos de uso *concretos*.

Las descripciones de los casos de uso abstractos se incluyen en las descripciones de casos de uso concretos. Los casos de uso abstractos también pueden ser usados por otros caso de uso abstractos. Normalmente los comportamientos similares entre casos de uso se identifican después de describir los casos de uso, aunque en algunos casos es posible identificarlos antes.

En la aplicación *FisioClinicApp* no se encuentra, a priori, ningún caso de uso de generalización.

2.3.3 Documentación

Como se ha dicho anteriormente, el modelo de casos de uso debe estar documentado, por lo que una de las principales partes del modelo es una descripción detallada de cada uno de los actores y casos de uso identificados. El formato de documentación para los actores se muestra a continuación:

Actor *Nombre del actor*

- **Casos de uso**: Nombre del caso de uso o casos de uso en los que actúa.
- **Tipo**: *Primario* o *Secundario*.

- **Descripción**: Razón de ser del actor.

Las descripciones de los casos de uso representan todas las interacciones del actor o actores con el sistema. En esta etapa no se incluyen eventos internos del sistema, ya que se trata en la parte de diseño (en el que se analizarán los diagramas de secuencias). El formato de documentación para los casos de uso se muestra a continuación:

Caso de uso *Nombre del caso de uso*

- **Actores**: Actores primarios y secundarios que actúan en el caso de uso.
- **Tipo**: *Básico, Inclusión, Extensión, Generalización.*
- **Propósito**: Razón de ser del caso de uso.
- **Resumen**: Resumen del caso de uso.
- **Precondiciones**: Condiciones que se tienen que satisfacer para poder ejecutar el caso de uso.
- **Flujo Principal**: El flujo de eventos más importante del caso de uso, donde dependiendo de las acciones de los actores se continuará con alguno de los subflujos.
- **Subflujos**: Los flujos secundarios del caso de uso o subflujos.
  - Estos subflujos están numerados como (S-1), (S-2), etc.
- **Excepciones**: Excepciones que se pueden ocurrir durante el caso.
  - Excepciones numeradas como (E-1), (E-2), etc.

En la sección 2.5 se detallan los actores y casos de uso de la aplicación usando el formato indicado anteriormente.

2.4 Modelo de Interfaces

El **modelo de interfaces** describe la presentación de información entre los actores y el sistema. Para ello, se especifica cómo se verán las interfaces de usuario al ejecutar cada uno de los casos de uso, lo cual ayuda al usuario a visualizarlos según sean mostrados por el sistema. Cuando se diseñan las interfaces de usuario, es esencial tener a los usuarios involucrados, siendo de máxima importancia que las interfaces reflejen la visión lógica del sistema.

En la sección 2.5.2 se muestran las interfaces usadas en la aplicación diseñada en este Trabajo de Fin de Grado, a la vez que se describen los casos de uso de dicha aplicación.

2.5 Actores y casos de uso de la aplicación FisioClinicApp

En esta sección se muestra la documentación de los actores y casos de uso, junto con el diseño de las interfaces, que serán usadas como prototipo del sistema.

2.5.1 Actores

En la aplicación *FisioClinicApp* existen dos actores (figura 2.2), que se muestran a continuación:

Actor *Usuario*

- **Casos de uso**: Acceso a pacientes, Acceso a calendario, Acceso a tareas, Control de pagos, Manejo datos usuario, Modificar pacientes, Manejo de eventos, Manejo de tareas y Manejo de pagos.
- **Tipo**: *Primario*.
- **Descripción**: Usuario registrado de la aplicación que puede realizar todas las tareas asociadas a la misma.

Actor *Base de datos*

- **Casos de uso**: Manejo de datos.
- **Tipo**: *Secundario*.
- **Descripción**: Sistema principal para guardar la información de la aplicación.

2.5.2 **Casos de uso**

Como apoyo en la descripción de los casos de uso de la aplicación se muestran las vistas propuestas para la aplicación. Estas vistas se han diseñado, inicialmente, usando una herramienta de prototipado que se usará para realizar un test a los usuarios potenciales de la aplicación, con el fin de probar la usabilidad de ésta. El test de usabilidad y los resultados del mismo se presentan en el capítulo *Modelo de pruebas*.

En el diseño de la interfaz de usuario se han planteado una serie de vistas (que constituyen las distintas pantallas de la aplicación) que se pasan a presentar a continuación.

Inicialmente, al entrar en la aplicación, se muestra la pantalla *Login* (figura 2.4), a través de la cual, el usuario puede entrar en la aplicación. Tal y como se observa en esta figura, en la pantalla *Login* se puede apreciar el logo identificativo de la aplicación, así como dos campos de texto: correo electrónico y contraseña.

Figura 2.4: Pantalla *Login* de la aplicación

Si los datos introducidos son correctos, el usuario que ya esté registrado, al seleccionar sobre el botón "ENTRAR", podrá acceder a la pantalla de inicio de la aplicación, llamada *Inicio*, correspondiente con la figura 2.6, que se comentará posteriormente. En la aplicación, el correo electrónico del usuario funcionará como identificador único del mismo, por lo que al entrar en la aplicación se reconocerá a

qué clínica de fisioterapia pertenece, y por tanto se accederá al uso de la aplicación dedicado a su organización.

Asimismo, y como se puede apreciar en la figura 2.4, no existe en esta aplicación, ninguna funcionalidad dedicada al registro de usuarios. Esto es porque la aplicación cuenta con un sistema de usuarios cerrado para cada clínica de fisioterapia, consiguiendo de esta manera que sólo accedan a la aplicación los profesionales de cada clínica. La función de registrar los usuarios para la aplicación será tarea de un usuario administrador, normalmente el encargado de cada clínica, quedando esta funcionalidad fuera del objeto de esta aplicación. En este aspecto, se recomienda la lectura del Apéndice B, *Gestión de usuarios*, para un mayor entendimiento.

Pese a este sistema cerrado de usuarios, la aplicación podrá ser descargada libremente por cualquier persona. De ahí que se habilite en la pantalla *Login* (figura 2.4), el texto "¿No tienes cuenta?". Al seleccionar este texto, se muestra un mensaje expresando que, en caso de querer hacer uso de la aplicación, deberá ponerse en contacto con el encargado de su clínica de fisioterapia, para que le proporcione una cuenta de usuario. Este caso se puede apreciar en la imagen de la izquierda de la figura 2.5. Asimismo, si el usuario no recuerda su contraseña, puede seleccionar el texto "¿Te olvidaste de la contraseña?", mostrándose la pantalla de la imagen derecha de la figura 2.5.

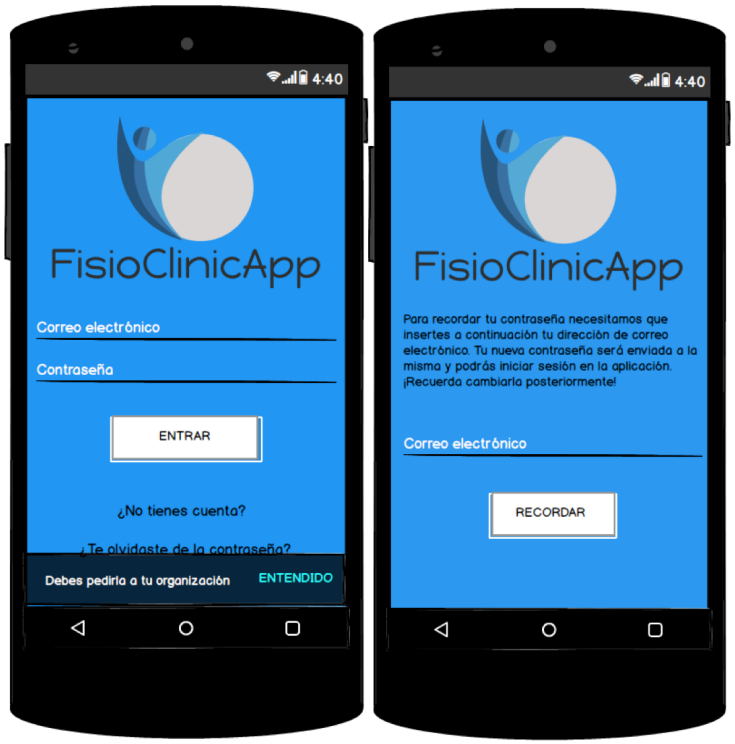

Figura 2.5: Pantalla *Login* con mensaje aclaratorio y Pantalla para recordar contraseña

Una vez iniciada la sesión dentro de la aplicación, se accede a la pantalla *Inicio* (figura 2.6). Como se puede observar, en la parte superior de esta pantalla se muestra el nombre de la clínica de fisioterapia, en la que trabaja el usuario que ha iniciado sesión, además de un botón en el lado izquierdo, que habilita el menú lateral de navegación de la aplicación, que se muestra en la figura 2.7.

En la pantalla *Inicio* (figura 2.6) y justo debajo del nombre de la clínica, se muestra un texto en forma de saludo al usuario, indicándole el día que es y presentándole la lista de eventos que tiene su clínica de fisioterapia para el día actual (con los eventos del usuario destacados en color azul).

Figura 2.6: Pantalla *Inicio* de la aplicación

En la figura 2.7 se muestran las posibles acciones que se pueden realizar desde el menú de la aplicación. Estas acciones son: "Inicio" (seleccionando este ítem se navega hacia la pantalla *Inicio*), "Pacientes" (seleccionando este ítem se navega hacia la pantalla dedicada a la gestión de los pacientes de la clínica, que se muestra en la figura 2.8), "Calendario" (seleccionando este ítem se navega hacia la pantalla dedicada a la gestión de los eventos de la clínica, que se presenta en la figura 2.16), "Tareas" (seleccionando este ítem se navega hacia la pantalla dedicada a la gestión de tareas a realizar, que se muestra en la figura 2.20), "Pagos" (seleccionando este ítem se navega hacia la pantalla dedicada al registro de pagos de la clínica, que se puede observar en la figura 2.22) y "Mi usuario" (seleccionando este ítem se navega hacia la pantalla dedicada a la gestión de su cuenta de usuario, que se presenta en la figura 2.26). Además, se puede observar que en el cabecero de este menú se encuentra el logotipo de la clínica de fisioterapia en cuestión.

Figura 2.7: Menú de navegación de la aplicación

Tras analizar estas pantallas principales, se pasa a explicar detalladamente los casos de uso de la aplicación, y cómo se interactúa con la interfaz de usuario.

2.5.2.1. Caso de uso *Acceso a pacientes*

El caso de uso *Acceso a pacientes* está vinculado a la elección del ítem "Pacientes" dentro del menú de navegación de la aplicación (figura 2.7). El detalle de este caso de uso se muestra a continuación:

Caso de uso *Acceso a pacientes*

- **Actores**: Usuario.
- **Tipo**: *Básico*.
- **Propósito**: Presentar los pacientes.
- **Resumen**: Muestra los pacientes de la clínica de fisioterapia a la que pertenece el usuario "logueado" (fisioterapeuta de la clínica).

- **Precondiciones**: El usuario debe estar "logueado" y el menú de navegación desplegado.

- **Flujo Principal**: A través del menú de navegación de la aplicación, el usuario accede al ítem *Pacientes* (figura 2.7), presentándosele una lista con cada uno de los pacientes de su clínica, pudiendo elegir alguno de ellos (figura 2.8) para acceder a su información particular.

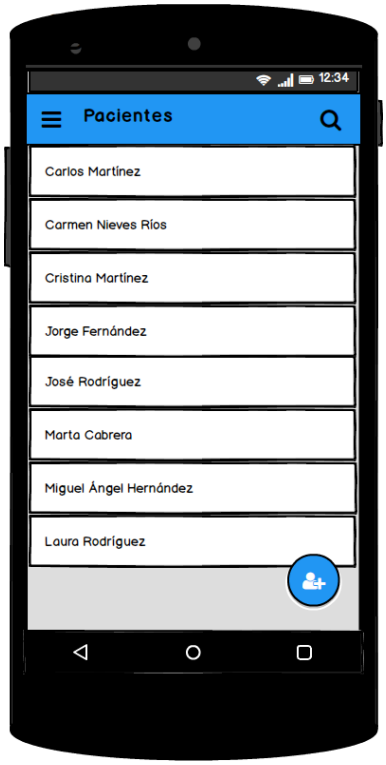

Figura 2.8: Pantalla con la lista de pacientes de la clínica de fisioterapia

- **Subflujos**:

  - *S-1 Acceder a ficha del paciente*: El usuario podrá seleccionar un determinado paciente y entrar a su ficha, donde podrá ver su información personal, además de manejar su archivo de imágenes y notas. Las pantallas relacionadas con este subflujo se pueden apreciar en la figura 2.9.
  - *S-2 Buscar paciente*: El usuario podrá seleccionar el icono "lupa", que le permitirá, en la misma pantalla, buscar un determinado paciente. Las pantallas relacionadas con este subflujo se pueden apreciar en la figura 2.10.

- **Excepciones**: Ninguno.

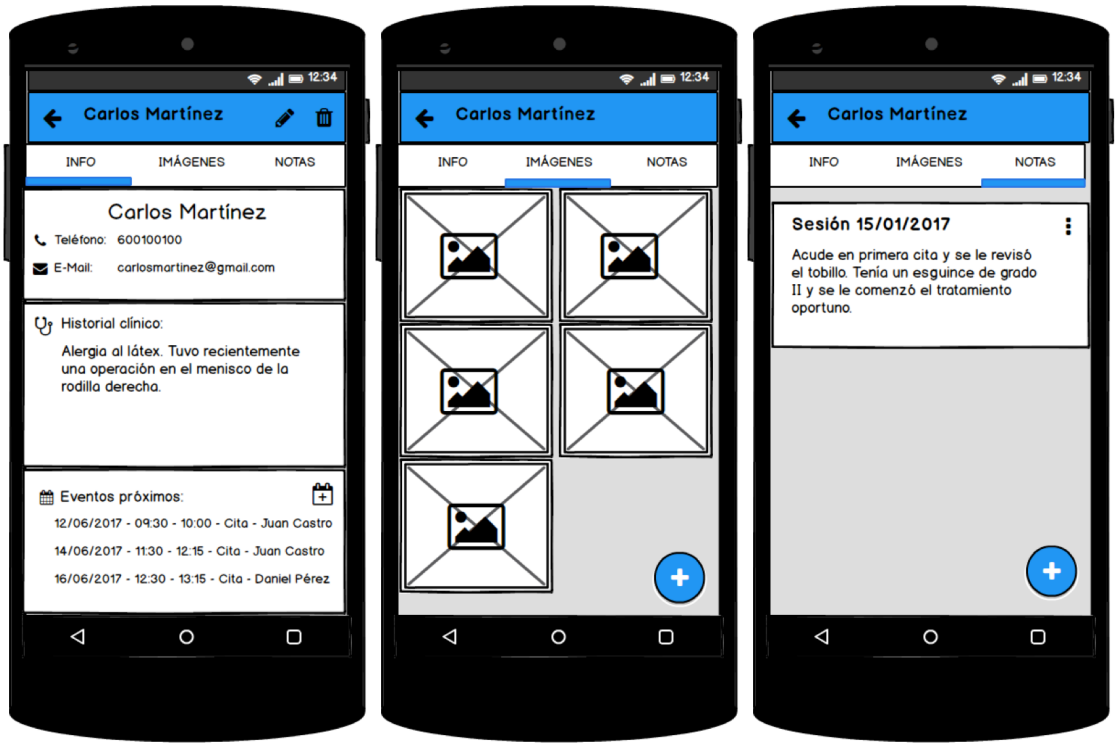

Figura 2.9: Pantallas relacionadas con la ficha personal de cada paciente

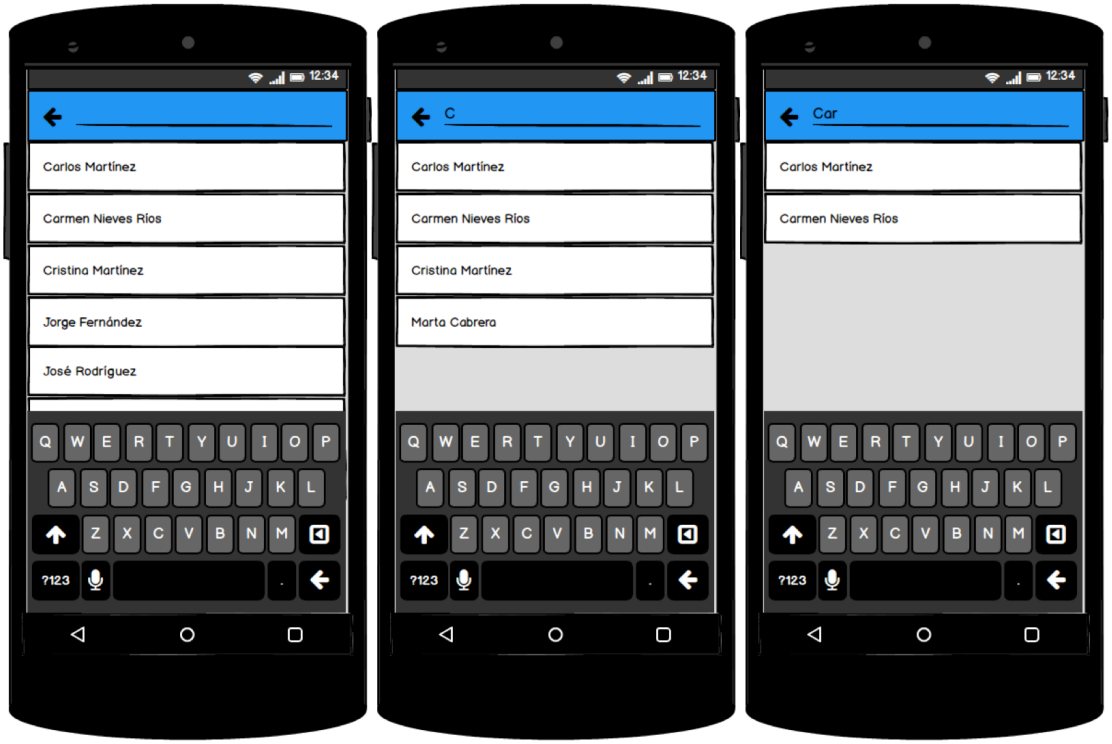

Figura 2.10: Pantallas relacionadas con la búsqueda de un paciente de la lista

2.5.2.2. Caso de uso *Modificar pacientes*

El caso de uso *Modificar paciente* es una extensión del caso de uso *Acceso a pacientes*, pues se da cuando el usuario decide realizar alguna función sobre los pacientes como tales. El detalle de este caso de uso es como sigue:

Caso de uso *Modificar pacientes*

- **Actores**: Usuario.
- **Tipo**: *Extensión.*
- **Propósito**: Manejo de la información de los pacientes.
- **Resumen**: Caso de uso iniciado por el usuario cuando desea realizar alguna modificación en lo que se refiere a los pacientes de la aplicación (crear, editar o eliminar).
- **Precondiciones**: El usuario debe estar "logueado" y haber accedido al ítem *Pacientes* de la aplicación, mostrándose la lista de pacientes en pantalla.
- **Flujo Principal**: Una vez el usuario accede al ítem *Pacientes* (figura 2.8), tiene a su disposición tres tareas principales para manipular los pacientes: crearlos, editarlos o eliminarlos. Cada una de ellas tendrá una pantalla correspondiente, donde llevar a cabo la tarea.
- **Subflujos**:
  - *S-1 Añadir paciente*: Esta función se hará desde la propia lista de pacientes (figura 2.8), a través del botón flotante de la pantalla. Cuando se selecciona este botón, se accede a la pantalla dedicada a añadir pacientes que se puede observar en la figura 2.11.
  - *S-2 Editar paciente*: Para editar un paciente, el usuario ha de acceder a la ficha de dicho paciente (figura 2.9), a través de la lista (figura 2.8). Una vez en esta pantalla, el usuario puede hacer uso del icono "lápiz" de la barra superior de la pantalla, permitiéndosele acceder a la pantalla específica para la edición de pacientes, que se puede ver en la figura 2.12.
  - *S-3 Eliminar paciente*: En la pantalla que muestra la ficha de un paciente (figura 2.9, el usuario puede hacer uso del icono "papelera" para eliminar al paciente. Las pantallas relacionadas con este subflujo se muestran en la figura 2.13.
  - *S-4 Añadir imagen de paciente*: Para añadir una imagen de un paciente, el usuario ha de acceder a la solapa IMÁGENES de la ficha de dicho paciente (figura 2.9), a través de la lista (figura 2.8). Una vez en esta pantalla, el usuario puede hacer uso del botón flotante (botón "+"), permitiéndosele añadir una nueva imagen y una descripción de la misma, tal y como se puede ver en la imagen izquierda de la figura 2.14.

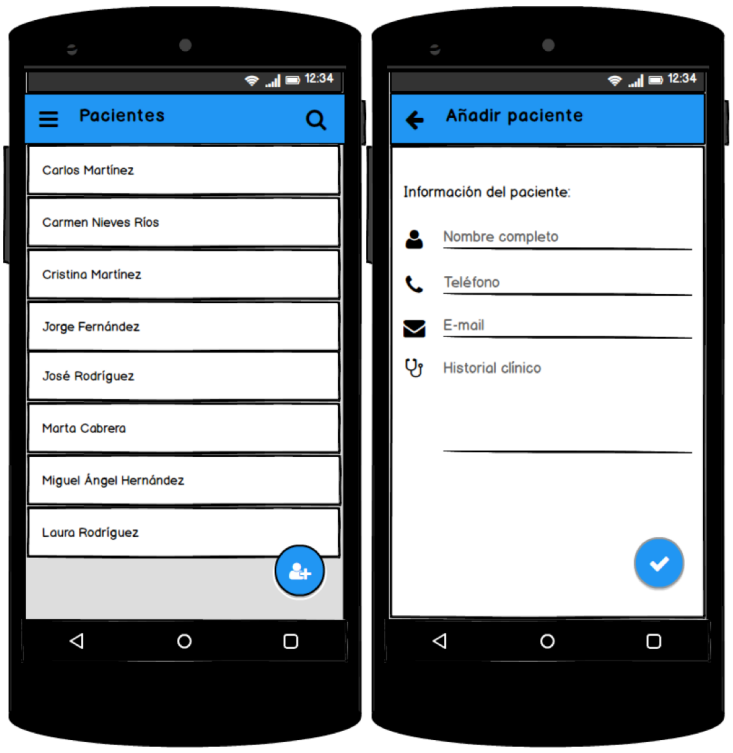

Figura 2.11: Pantallas relacionadas con la adición de un paciente

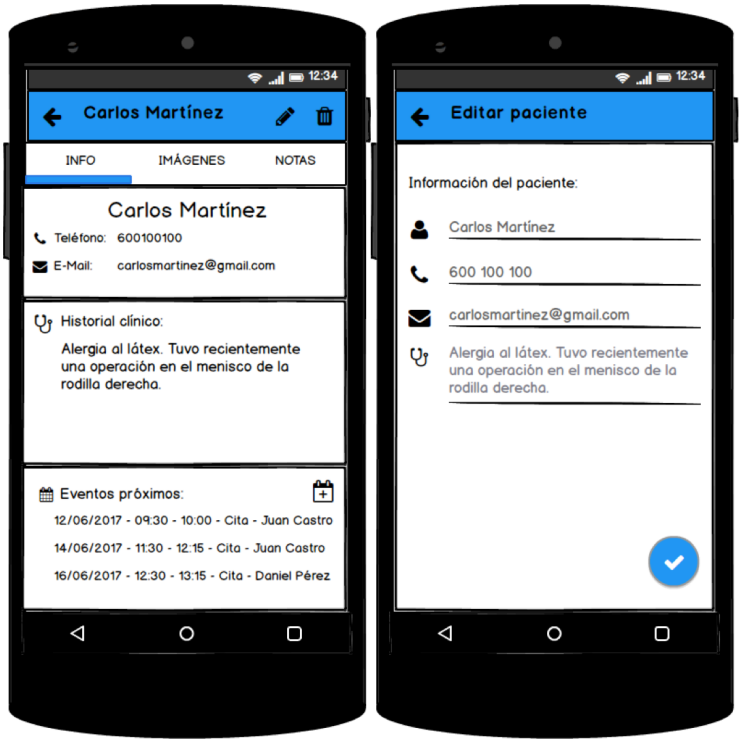

Figura 2.12: Pantallas relacionadas con la edición de un paciente

Figura 2.13: Pantallas relacionadas con la eliminación de un paciente

- *S-5 Ver detalle de la imagen*: Para ver el detalle de una imagen de un paciente, el usuario ha de acceder a la solapa IMÁGENES de la ficha de dicho paciente (figura 2.9), a través de la lista (figura 2.8). Una vez en esta pantalla, el usuario puede seleccionar una imagen detalladamente, es decir, la imagen con su descripción, tal y como se puede ver en la imagen central de la figura 2.14.
- *S-6 Eliminar imagen*: Para eliminar una imagen de un paciente, el usuario ha de acceder a la solapa IMÁGENES de la ficha de dicho paciente (figura 2.9), a través de la lista (figura 2.8). Una vez en esta pantalla, el usuario selecciona la imagen que desea eliminar, accediendo al detalle de la imagen (imagen central de la figura 2.14). Tras ello, podrá utilizar el botón "papelera" para eliminar dicha imagen, tras haber realizado la confirmación que le solicita el diálogo que se puede apreciar en la imagen de la derecha de la figura 2.14.
- *S-7 Añadir nota de paciente*: Para añadir una nota sobre un paciente, el usuario ha de acceder a la solapa NOTAS de la ficha de dicho paciente (figura 2.9), a través de la lista (figura 2.8). Una vez en esta pantalla, el usuario puede hacer uso del botón flotante (botón "+"), permitiéndosele añadir una nueva nota sobre el paciente, tal y como se puede ver en la imagen de la izquierda de la figura 2.15.
- *S-8 Editar nota de paciente*: Para editar una nota sobre un paciente, el usuario ha de acceder a la solapa NOTAS de la ficha de dicho paciente

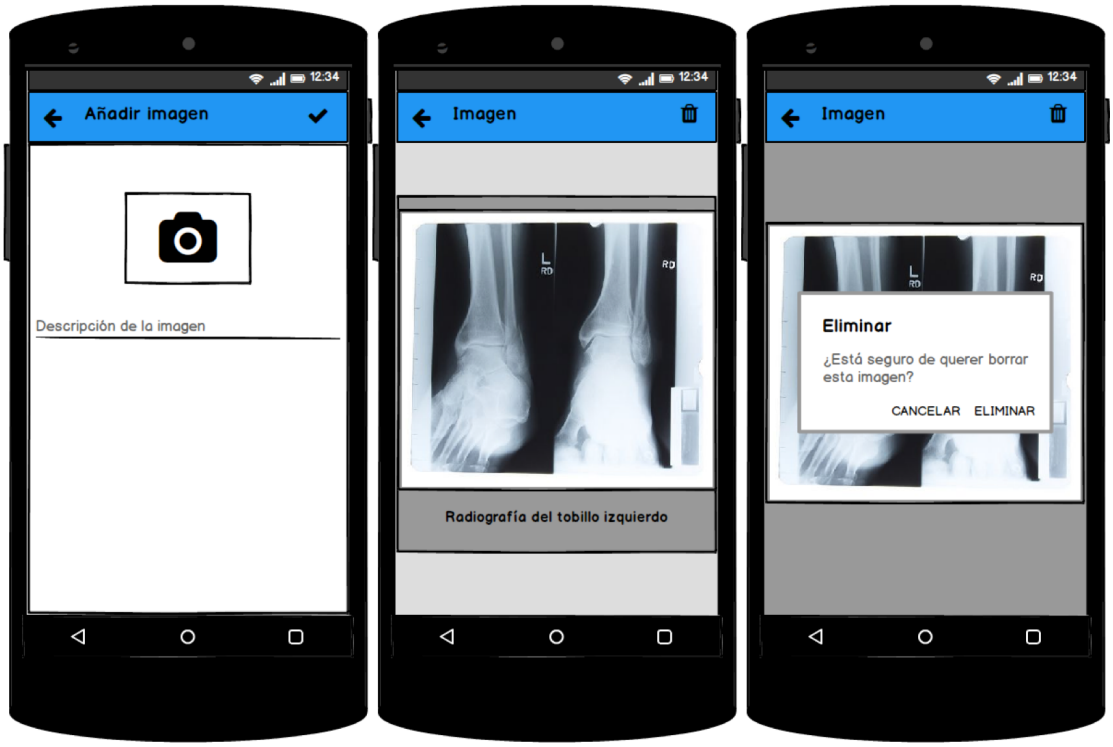

Figura 2.14: Pantalla para añadir una nueva imagen de un paciente, Pantalla para ver el detalle de una imagen y Pantalla para eliminar una imagen de un paciente

(figura 2.9), a través de la lista (figura 2.8). Una vez en esta pantalla, el usuario puede elegir qué nota quiere editar y seleccionando, a continuación, el menú flotante de esa nota. Al realizar esta acción, el usuario podrá modificar la nota en una pantalla que se muestra en la imagen central de la figura 2.15

- *S-9 Eliminar nota de paciente*: Para eliminar una nota sobre un paciente, el usuario ha de acceder a la solapa NOTAS de la ficha de dicho paciente (figura 2.9), a través de la lista (figura 2.8). Una vez en esta pantalla, el usuario puede elegir qué nota quiere eliminar y seleccionando, a continuación, el menú flotante de esa nota. Al realizar esta acción, el usuario podrá eliminar la nota tras realizar la confirmación oportuna, tal como se muestra en la imagen de la derecha de la figura 2.15.

- **Excepciones**: Ninguno.

2.5.2.3. Caso de uso *Acceso a calendario*

El caso de uso *Acceso a calendario* está vinculado a la elección del ítem "Calendario" dentro del menú de navegación de la aplicación (figura 2.7). El detalle de este caso de uso es el siguiente:

Figura 2.15: Pantalla para añadir nuevas notas sobre un paciente, Pantalla para editar una nota de un paciente y Pantalla para la eliminación de una nota de un paciente

Caso de uso *Acceso a calendario*

- **Actores**: Usuario.
- **Tipo**: *Básico*.
- **Propósito**: Presentación del calendario de la aplicación.
- **Resumen**: Muestra el calendario de la aplicación dedicado a la gestión de los eventos de la clínica de fisioterapia del usuario.
- **Precondiciones**: El usuario debe estar "logueado" y el menú de navegación desplegado.
- **Flujo Principal**: A través del menú de navegación de la aplicación (figura 2.7), el usuario accede al ítem *Calendario* (figura 2.16), donde encontrará el calendario de la aplicación para la gestión de los eventos de la clínica de fisioterapia.
- **Subflujos**: Ninguno.
- **Excepciones**: Ninguna.

Figura 2.16: Pantalla con el calendario de la aplicación

2.5.2.4. Caso de uso *Manejo de eventos*

El caso de uso *Manejo de eventos* es una extensión del caso de uso *Acceso a calendario*, pues se da cuando el usuario decide acceder a una fecha concreta del calendario para manipular (añadir, editar o eliminar) algún evento marcado para esa fecha. El detalle del caso de uso se muestra a continuación:

Caso de uso *Manejo de eventos*

- **Actores**: Usuario.
- **Tipo**: *Extensión.*
- **Propósito**: Manipulación de los eventos del calendario.
- **Resumen**: Caso de uso iniciado por el usuario cuando desea realizar alguna gestión referente a los eventos de la aplicación (crear, listar, editar o eliminar).
- **Precondiciones**: El usuario debe estar "logueado" y haber accedido al ítem *Calendario* de la aplicación, además de haber seleccionado algún día en concreto del calendario.

- **Flujo Principal**: Una vez el usuario accede al ítem *Calendario* (figura 2.16), el usuario ha de elegir el día con el que desea interactuar. En él podrá hacer cada una de las tareas principales para este apartado: crear, listar, editar y eliminar eventos. Cada una de ellas será un subflujo y tendrá una pantalla correspondiente donde llevarla a cabo.
- **Subflujos**:
  - *S-1 Listar eventos*: Al seleccionar un día específico dentro del *Calendario* (figura 2.16), el usuario podrá ver qué eventos existen hasta el momento para el día seleccionado. En la figura 2.17 se muestra un ejemplo de lista de eventos para un determinado día del calendario.

Figura 2.17: Pantalla con la lista de eventos para un día en concreto

  - *S-2 Añadir evento*: Una vez elegido un determinado día, el usuario puede seleccionar el botón flotante de la pantalla (figura 2.17), que permite acceder a una nueva vista, en la que el usuario puede crear un evento para ese día. En la figura 2.18 se muestra el formulario para añadir eventos.
  - *S-3 Editar evento*: Una vez elegido un determinado día, el usuario puede seleccionar el icono "lápiz" asociado al evento (figura 2.17), que permite visualizar una vista dedicada a la edición de eventos similar a la de la figura 2.18.
  - *S-4 Eliminar evento*: Una vez elegido un determinado día, el usuario puede seleccionar el icono "papelera" asociado al evento (figura 2.17), que permite eliminar el evento seleccionado (figura 2.19).

Figura 2.18: Pantalla dedicada a la adición de eventos

Figura 2.19: Pantalla con muestra de mensaje a la hora de eliminar un evento

- **Excepciones**:
  - *E-1 Fallo de formulario:* El usuario no ha rellenado el formulario de forma correcta.

2.5.2.5. Caso de uso *Acceso a tareas*

Este caso de uso está vinculado a la elección del ítem "Tareas" dentro del menú de navegación de la aplicación (figura 2.7). El detalle es el siguiente:

Caso de uso *Acceso a tareas*

- **Actores**: Usuario.
- **Tipo**: *Básico*.
- **Propósito**: Presentación del listado de tareas.
- **Resumen**: Muestra el listado de tareas de la aplicación para que los usuarios de la clínica de fisioterapia puedan anotar tareas a realizar, además de darlas por finalizadas.
- **Precondiciones**: El usuario debe estar "logueado" y el menú de navegación desplegado.
- **Flujo Principal**: A través del menú de navegación de la aplicación (figura 2.7), el usuario accede al ítem *Tareas* (figura 2.20), donde encontrará una lista de tareas, pendientes o realizadas, de la clínica de fisioterapia.
- **Subflujos**: Ninguno.
- **Excepciones**: Ninguno.

2.5.2.6. Caso de uso *Manejo de tareas*

El caso de uso *Manejo de tareas* es una extensión del caso de uso *Acceso a tareas*, ya que se da cuando el usuario desea manipular las tareas del listado de tareas presentado en la figura 2.20. El detalle del caso de uso es el siguiente:

Caso de uso *Manejo de tareas*

- **Actores**: Usuario.

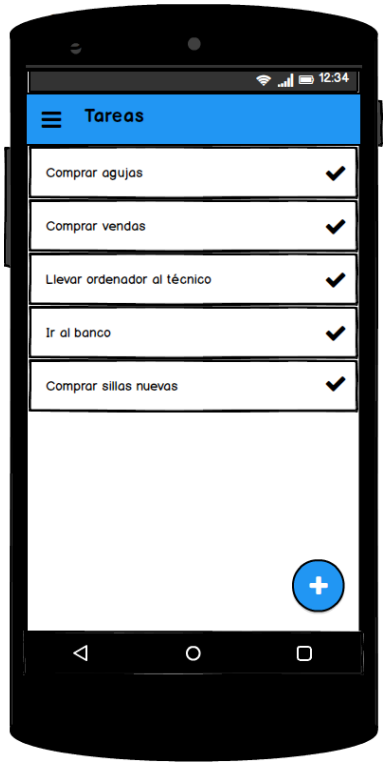

Figura 2.20: Pantalla el listado de tareas

- **Tipo**: *Extensión.*
- **Propósito**: Manipulación de la lista de tareas de la clínica.
- **Resumen**: Caso de uso iniciado por el usuario cuando desea realizar alguna gestión referente a las tareas de la aplicación (crear o eliminar).
- **Precondiciones**: El usuario debe estar "logueado" y haber accedido al ítem *Tareas* de la aplicación.
- **Flujo Principal**: Una vez el usuario accede al ítem *Tareas* (figura 2.20), el usuario podrá ver la lista de tareas que tiene su clínica de fisioterapia anotadas. El usuario, además de poder añadir nuevas tareas a esta lista, podrá eliminarlas definitivamente, siendo cada una de estas funciones un subflujo de este caso de uso.
- **Subflujos**:
  - *S-1 Añadir tarea*: Una vez mostrada la lista de tareas, tal y como se presenta en la pantalla de la figura 2.20, el usuario puede seleccionar el botón flotante de dicha pantalla, apareciendo un diálogo sobre la misma, en el que se le pide rellenar un campo con la descripción de la nueva tarea, tal y como se muestra en la figura 2.21.

Figura 2.21: Pantalla que muestra el diálogo habilitado para crear una tarea

- *S-2 Finalizar tarea*: Estas tareas presentan, como se aprecia en la pantalla (figura 2.20), un icono con símbolo "check", que, seleccionándolo, permite dar la tarea por finalizada y, por tanto, eliminarla de la lista.

- **Excepciones**: Ninguno.

2.5.2.7. Caso de uso *Control de pagos*

El caso de uso *Control de pagos* está vinculado a la elección del ítem "Pagos" dentro del menú de navegación de la aplicación (figura 2.7). El detalle del caso de uso se describe a continuación:

Caso de uso *Control de pagos*

- **Actores**: Usuario.
- **Tipo**: *Básico*.
- **Propósito**: Presentación de los listados de pagos realizados y pendientes.

- **Resumen**: Presenta dos listados referidos al control de pagos de la clínica: pagos realizados y pagos pendientes de realizar.
- **Precondiciones**: El usuario debe estar "logueado" y el menú desplegado.
- **Flujo Principal**: A través del menú de navegación de la aplicación (figura 2.7), el usuario accede al ítem *Pagos* (figura 2.22), donde encontrará una lista de pagos, pendientes o realizados.

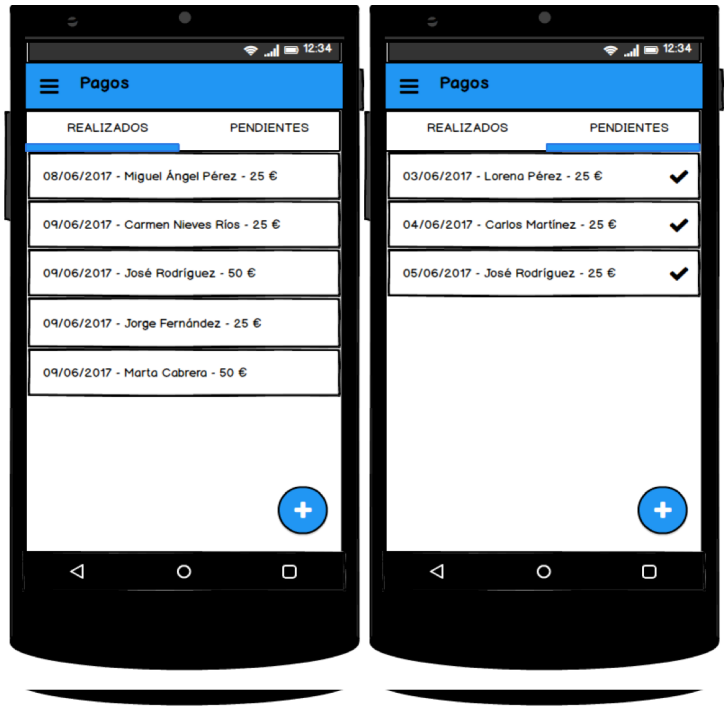

Figura 2.22: Pantalla del listado de pagos

- **Subflujos**: Ninguno.
- **Excepciones**: Ninguno.

2.5.2.8. Caso de uso *Manejo de pagos*

El caso de uso *Manejo de pagos* es una extensión del caso de uso *Control de pagos*, dándose cuando el usuario desea hacer algún tipo de gestión con los pagos de los diferentes listados presentados en la figura 2.22. El detalle del caso de uso es:

Caso de uso *Manejo de pagos*

- **Actores**: Usuario.

- **Tipo**: *Extensión.*
- **Propósito**: Gestión de los pagos de los pacientes de la clínica de fisioterapia.
- **Resumen**: El usuario podrá gestionar los pagos que se hacen o quedan pendientes por hacer, añadiéndolos a través de las pantallas que se muestran en la figura 2.22.
- **Precondiciones**: El usuario debe estar "logueado" y haber accedido al ítem *Pagos* de la aplicación.
- **Flujo Principal**: Cuando el usuario accede al ítem *Pagos* (figura 2.22), el usuario podrá acceder a los listados de pagos realizados y pendientes gracias a las pestañas que se encuentran en la barra superior. En cada uno de ellos, podrá añadir nuevos pagos. Además, en el caso de estar en el listado de pagos pendientes, se puede dar por realizados estos pagos, pasando, por tanto, al listado de pagos realizados. Cada una de estas funciones es un subflujo de este caso de uso.
- **Subflujos**:
  - *S-1 Añadir pago realizado*: En la pantalla de los pagos realizados, el usuario puede seleccionar el botón flotante "+", para añadir un pago. Para ello, se despliega un diálogo en el que se rellenan la información del pago, tal y como se muestra en la figura 2.23.
  - *S-2 Añadir pago pendiente*: En el caso de estar en el listado de pagos pendientes, el usuario puede seleccionar el botón flotante "+", para añadir un pago. Para ello, se despliega un diálogo en el que se rellenan la información del pago, tal y como se muestra en la figura 2.24.
  - *S-3 Dar pago pendiente por realizado*: Los pagos pendientes cuentan con un botón "check", que permite cambiar el pago indicado a la lista de pagos realizados, tal y como muestra la figura 2.25.
- **Excepciones**:
  - *E-1 Fallo de formulario:* El usuario no ha rellenado el formulario de forma correcta.

2.5.2.9. Caso de uso *Manejo datos usuario*

El caso de uso *Manejo datos usuario* está vinculado a la elección del ítem "Mi usuario" dentro del menú de navegación de la aplicación (figura 2.7). El detalle del caso de uso es el siguiente:

Figura 2.23: Pantalla que muestra el diálogo con el formulario a rellenar para añadir un pago realizado

Caso de uso *Manejo datos usuario*

- **Actores**: Usuario.
- **Tipo**: *Básico*.
- **Propósito**: Presentación de la información del usuario y posible modificación de la misma.
- **Resumen**: Presenta, al usuario, su información de perfil, pudiendo éste modificar la dirección de correo electrónico, su *password* y darse de baja en la aplicación (si deja su puesto de trabajo).
- **Precondiciones**: El usuario debe estar "logueado" y el menú de navegación desplegado.
- **Flujo Principal**: A través del menú de navegación de la aplicación (figura 2.7), el usuario accede al ítem *Mi usuario* (figura 2.26), donde encontrará la información del perfil.

  Asimismo, esta pantalla dispondrá de un menú desplegable en la parte superior derecha, a través del cual se puede modificar la contraseña, editar la información y darse de baja. Estas acciones corresponde, cada una, con un subflujo del caso de uso. Estos son:

Figura 2.24: Pantalla que muestra el diálogo con el formulario a rellenar para añadir un pago pendiente

Figura 2.25: Pantalla que presenta el diálogo a la hora de dar un pago pendiente por realizado

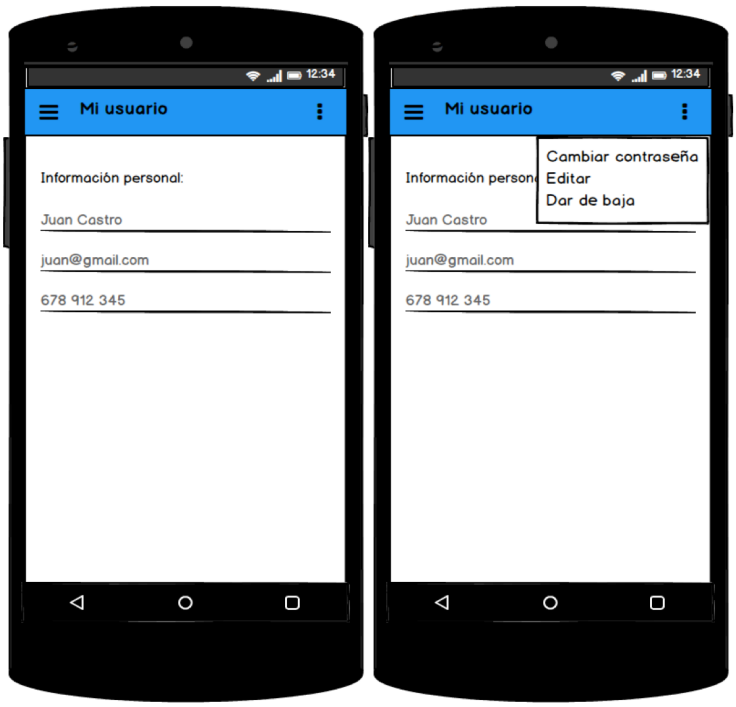

Figura 2.26: Pantalla *Mi Usuario* y su menú desplegable

- **Subflujos**:
  - *S-1 Cambiar contraseña*: En la pantalla de los datos del perfil, el usuario puede seleccionar el menú flotante y seleccionar el ítem "Cambiar contraseña". Para ello, se despliega un diálogo en el que se rellenan la información de la nueva contraseña, tal y como se muestra en la figura 2.27.
  - *S-2 Editar perfil*: En la pantalla de los datos del perfil, el usuario puede seleccionar el menú flotante y seleccionar el ítem "Editar". Para ello, y después de modificar la información en la pantalla del perfil, se despliega un diálogo en el que se confirma la modificación, tal y como se muestra en la figura 2.28.
  - *S-3 Dar de baja*: En la pantalla de los datos del perfil, el usuario puede seleccionar el menú flotante y seleccionar el ítem "Dar de baja". Para ello, se despliega un diálogo en el que el usuario debe confirmar la baja, tal y como se muestra en la figura 2.29.
- **Excepciones**: Ninguno.

2.5.2.10. Caso de uso *Manejo de datos*

El caso de uso *Manejo de datos* es el caso de uso principal del actor *Base de datos*, el cual es imprescindible en esta aplicación, ya que interviene en prácticamente todos

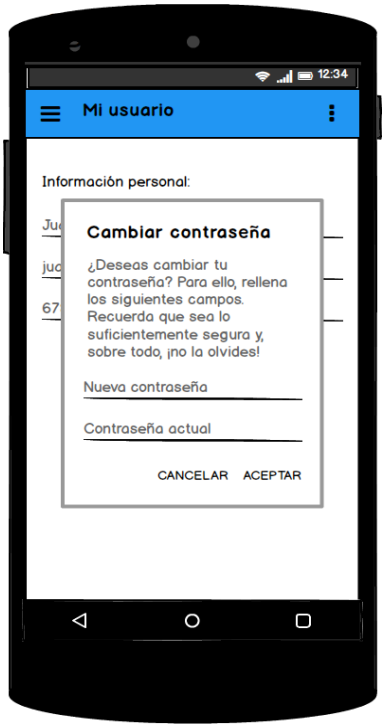

Figura 2.27: Pantalla para cambiar el password del usuario

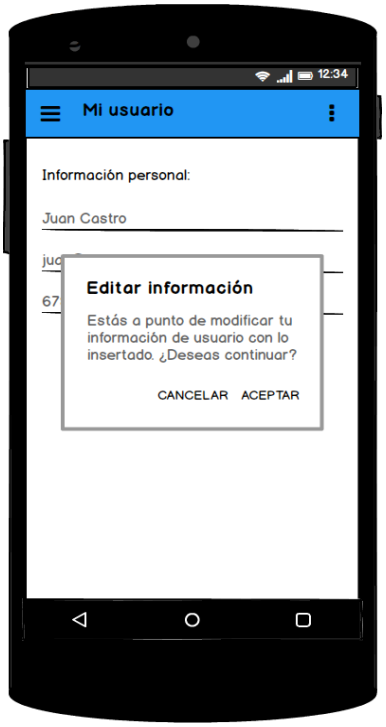

Figura 2.28: Pantalla para confirmar la modificación del perfil del usuario

los casos de uso comentados, al requerirse la gestión de datos constantemente. El detalle de este caso de uso es el siguiente:

Figura 2.29: Pantalla para confirmar la baja del usuario

Caso de uso *Manejo de datos*

- **Actores**: Base de datos.
- **Tipo**: *Inclusión*.
- **Propósito**: Interactuar con la base de datos externa para garantizar el correcto funcionamiento de la aplicación.
- **Resumen**: Este caso de uso se utiliza para que la aplicación obtenga y actualice de y a la base de datos, toda la información requerida o modificada por el usuario (gestión de datos que provee la base de datos).
- **Precondiciones**: Tener acceso a Internet.
- **Flujo Principal**: Accede a la base de datos externa (en la "nube") para insertar, recuperar, modificar o eliminar información. Este caso de uso está incluido en todos los casos de uso de la aplicación, excepto en el caso de uso *Acceso a calendario*.
- **Subflujos**:
  - *S-1 Recuperar información*: Esta funcionalidad permite al usuario obtener la información de pacientes, eventos, tareas o pagos.
  - *S-2 Modificar información*: Esta funcionalidad permite al usuario modificar información de pacientes, eventos, tareas o pagos.

- *S-3 Eliminar información*: Esta funcionalidad permite al usuario eliminar información de pacientes, eventos, tareas o pagos.
- *S-4 Agregar información*: Esta funcionalidad permite al usuario añadir nueva información de pacientes, eventos, tareas o pagos.

- **Excepciones**:
  - *E-1 Fallo de acceso a Internet:* El usuario no puede conectarse a Internet para recuperar y enviar la información necesaria.

Capítulo **3**

MODELO DE DISEÑO

3.1 Descripción de la arquitectura MVP

La arquitectura MVP o *Modelo-Vista-Presentador* [18] es una arquitectura que tiene como objetivo separar la interfaz de usuario de la lógica de las aplicaciones. Básicamente, esta arquitectura consiste en tres componentes (figura 3.1):

- La *vista*. Compuesta de las ventanas y controles que forman la interfaz de usuario de la aplicación.
- El *modelo*. Donde se lleva a cabo toda la lógica de negocio.
- El *presentador*. Escucha los eventos que se producen en la vista y ejecuta las acciones necesarias a través del modelo. Además, puede tener acceso a las vistas a través de las interfaces que la vista debe implementar.

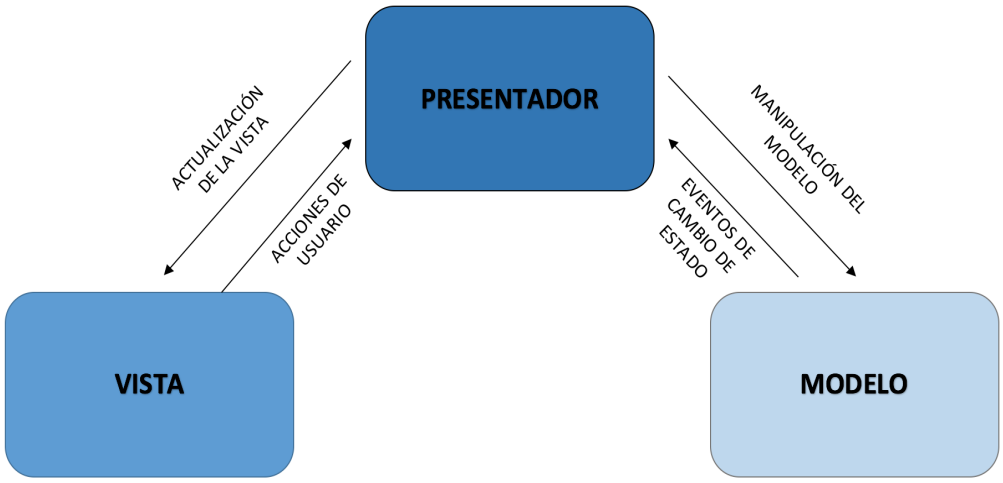

Figura 3.1: Representación de la arquitectura *Modelo-Vista-Presentador*

La explicación de esta arquitectura es bastante sencilla. Por un lado, se tiene la vista que se encarga de mostrar la información al usuario y de interactuar con él

para hacer ciertas operaciones. Por otro lado, está el modelo que, ignorante de cómo la información es mostrada al usuario, realiza toda la lógica de negocio usando las entidades del dominio. Y por último, se tiene al presentador que es el que interactúa con ambos componentes sin que haya ningún tipo de dependencia entre ellos.

La vista en la arquitectura MVP debe ser pasiva, es decir, cuando el usuario provoque un evento sobre la interfaz de usuario, la vista debe delegar en el presentador el tratamiento de ese evento. A su vez, el modelo realiza las acciones que le solicita el presentador cuando corresponda (acciones como recuperar información de una base de datos interna o externa, entre otros). Cuando la información solicitada está disponible, el modelo "notifica" que los datos están accesibles y el presentador, que está a la espera, "observando" al modelo (objeto "observado") responderá a la notificación como corresponda a la aplicación desarrollada. Es importante indicar que en la arquitectura MVP, el modelo "no conoce" al presentador ni a la vista.

Además de los tres componentes indicados, la aplicación a desarrollar contará con un componente extra, al cual se llamará, *Mediador*, que realizará aquel proceso que no es particular de ningún presentador en concreto, sino que se encargará de tareas que todos los presentadores tendrán que realizar y que por conveniencia, se concentrarán en un único punto del código. Normalmente, estas operaciones tienen que ver con el sistema operativo del terminal.

Por último, en la descripción de esta arquitectura se puede ver que todos los componentes están perfectamente desacoplados, lo que implica que cualquier cambio sobre cualquier componente, no afecta al resto. Asimismo, el código final puede ser probado con relativamente poco esfuerzo y de forma simultánea a la implementación de los diferentes componentes.

3.1.1 Clases identificadas en la aplicación

Una vez completado el *Modelo de requisitos* se pueden hacer un primer boceto de las clases identificadas en la aplicación. Así, por cada pantalla de la aplicación (según el prototipo realizado), habrá una clase relacionada (se denominan clases de la vista). Asimismo, con cada clase de la vista habrá también una clase relacionada (denominadas clases del presentador). Por último, habrá una clase para el modelo que se encargará del manejo de los datos. Según este criterio, las clases identificadas en esta aplicación son:

- Clase *VistaLogin* con su interfaz *IVistaLogin* (relacionada con la vista mostrada en la figura 2.4).
- Clase *VistaRecordarPassword* con su interfaz *IVistaRecordarPassword* (relacionada con la vista mostrada en la figura 2.5).

- Clase *VistaInicio* con su interfaz *IVistaInicio* (relacionada con la vista mostrada en la figura 2.6).
- Clase *VistaPacientes* con su interfaz *IVistaPacientes* (relacionada con la vista mostrada en la figura 2.8).
- Clase *VistaFichaPaciente* con su interfaz *IVistaPaciente* (relacionada con las imágenes mostradas en la figura 2.9).
- Clase *VistaAgregarEditarPaciente* con su interfaz *IVistaAgregarEditarPaciente* (relacionada con la imagen de la derecha de las vistas mostradas en las figuras 2.11 y 2.12. Como se puede apreciar, ambas imágenes son iguales).
- Clase *VistaAgregarImagen* con su interfaz *IVistaAgregarImagen* (relacionada con la vista mostrada en la parte izquierda de la figura 2.14).
- Clase *VistaDetalleImagen* con su interfaz *IVistaDetalleImagen* (relacionada con la vista mostrada en la imagen central de la figura 2.14).
- Clase *VistaAgregarEditarNota* con su interfaz *IVistaAgregarEditarNota* (relacionada con la vista mostrada en la imagen de la izquierda y la imagen central de la figura 2.15. Como se puede apreciar, ambas imágenes son iguales).
- Clase *VistaCalendario* con su interfaz *IVistaCalendario* (relacionada con la vista mostrada en la figura 2.16).
- Clase *VistaEventos* con su interfaz *IVistaEventos* (relacionada con la vista mostrada en la figura 2.17).
- Clase *VistaAgregarEditarEvento* con su interfaz *IVistaAgregarEditarEvento* (relacionada con la vista mostrada en la figura 2.18).
- Clase *VistaPagos* con su interfaz *IVistaPagosRealizados* (relacionada con la figura 2.22).
- Clase *VistaTareas* con su interfaz *IVistaTareas* (relacionada con la vista mostrada en la figura 2.20).
- Clase *VistaUsuario* con su interfaz *IVistaUsuario* (relacionada con la vista mostrada en la figura 2.26).
- Clase *PresentadorLogin* con su interfaz *IPresentadorLogin* (presentador de la vista *VistaLogin*).
- Clase *PresentadorRecordarPassword* con su interfaz *IPresentadorRecordarPassword* (presentador de la vista *VistaRecordarPassword*).
- Clase *PresentadorInicio* con su interfaz *IPresentadorInicio* (presentador de la vista *VistaInicio*).
- Clase *PresentadorPacientes* con su interfaz *IPresentadorPacientes* (presentador de la vista *VistaPacientes*).

- Clase *PresentadorFichaPaciente* con su interfaz *IPresentadorFichaPaciente* (presentador de la vista *VistaFichaPaciente*).
- Clase *PresentadorAgregarEditarPaciente* con su interfaz *IPresentadorAgregarEditarPaciente* (presentador de la vista *VistaAgregarEditarPaciente*).
- Clase *PresentadorAgregarImagen* con su interfaz *IPresentadorAgregarImagen* (presentador de la vista *VistaAgregarImagen*).
- Clase *PresentadorDetalleImagen* con su interfaz *IPresentadorDetalleImagen* (presentador de la vista *VistaDetalleImagen*).
- Clase *PresentadorAgregarEditarNota* con su interfaz *IPresentadorAgregarEditarNota* (presentador de la vista *VistaAgregarEditarNota*).
- Clase *PresentadorCalendario* con su interfaz *IPresentadorCalendario* (presentador de la vista *VistaCalendario*).
- Clase *PresentadorEventos* con su interfaz *IPresentadorEventos* (presentador de la vista *VistaEventos*).
- Clase *PresentadorAgregarEditarEvento* con su interfaz *IPresentadorAgregarEditarEvento* (presentador de la vista *VistaAgregarEditarEvento*).
- Clase *PresentadorPagos* con su interfaz *IPresentadorPagos* (presentador de la vista *VistaPagos*).
- Clase *PresentadorTareas* con su interfaz *IPresentadorTareas* (presentador de la vista *VistaTareas*).
- Clase *PresentadorUsuario* con su interfaz *IPresentadorUsuario* (presentador de la vista *VistaUsuario*).
- Clase *Modelo* con su interfaz *IModelo*.

3.2 Diagramas de secuencia

Partiendo del prototipo realizado en el *Modelo de requisitos*, se tratará de descubrir los métodos a implementar en cada clase identificada en el apartado anterior. Para ello, se hará uso de los diagramas de secuencia para definir el comportamiento de las interfaces de estas clases.

Así, analizando los posibles usos que tiene la aplicación, se obtienen unos diagramas de secuencia determinados. Los diagramas de secuencia describen el uso de la aplicación según los eventos enviados entre los objetos de la arquitectura MVP y presentan aspectos dinámicos de un sistema, a diferencia de los diagramas de clases, que muestran información estática. Por esta razón, los diagramas de secuencia

utilizan objetos mientras que los diagramas de clases utilizan clases como elementos básicos.

Cada objeto en el diagrama se representa con una línea vertical, que corresponde al eje temporal, donde el tiempo avanza hacia abajo. En estos diagramas se muestran los eventos que ocurren en el tiempo, los cuales son enviados de un objeto a otro. El orden de los objetos no es importante, sino el orden en el que ocurren los eventos y la dependencia entre ellos, es decir, qué consecuencias tiene el envío de un evento.

Los mensajes enviados entre objetos corresponden con los métodos que hay que definir en las interfaces implementadas por las clases de esos objetos. Así, un mensaje enviado entre un objeto vista y un objeto presentador se representa mediante una flecha que va desde la línea vertical del objeto vista, hacia la línea vertical del objeto presentador, y tiene un identificador que corresponde con el nombre del método. En el diagrama de secuencia no se muestran los datos enviados o recibidos, sólo se muestran los identificadores de los mensajes enviados.

En los siguientes apartados se presentan los diagramas secuencia de la *Aplicación para la gestión de una clínica de fisioterapia*, representándose cada caso de uso. Cada uno de los métodos que aparecen en estos diagramas, se explica en las siguientes secciones, indicando los parámetros y el tipo de datos que devuelven (si es el caso). Asimismo, hay que indicar que la información marcada en rojo está relacionada con el *Mediador* de la aplicación (que se explicará en el capítulo *Modelo de implementación* más detalladamente).

3.2.1 Diagramas de secuencia relacionados con el acceso a la aplicación

En la figura 3.2 se presenta el flujo de mensajes intercambiados entre la vista, el presentador y el modelo, cuando el usuario desea iniciar sesión en la aplicación, y por tanto hacer uso de las funcionalidades que da la misma. Teniendo en cuenta la pantalla que se muestra figura 2.4, básicamente el presentador se encargará de recoger el evento producido por el usuario cuando éste inserta en la vista la información necesaria para realizar el inicio de sesión (correo electrónico y contraseña). Estos datos serán comunicados al modelo a través del presentador, realizando el modelo la comprobación necesaria en la base de datos. Suponiendo que es correcto, el presentador solicitará al mediador de la aplicación que cargue la vista correspondiente, es decir, la vista de inicio (pantalla correspondiente a la figura 2.6). Dado que esta pantalla muestra la lista de eventos que tiene su clínica de fisioterapia el día actual, esta vista, al ser cargada, solicitará al presentador la información necesaria para mostrar al usuario. Por ello, el presentador solicitará estos datos al modelo, que cuando los obtenga de la base de datos, se los proporcionará al presentador, que se encargará de pasarlos a la vista para, finalmente, mostrarlos al usuario.

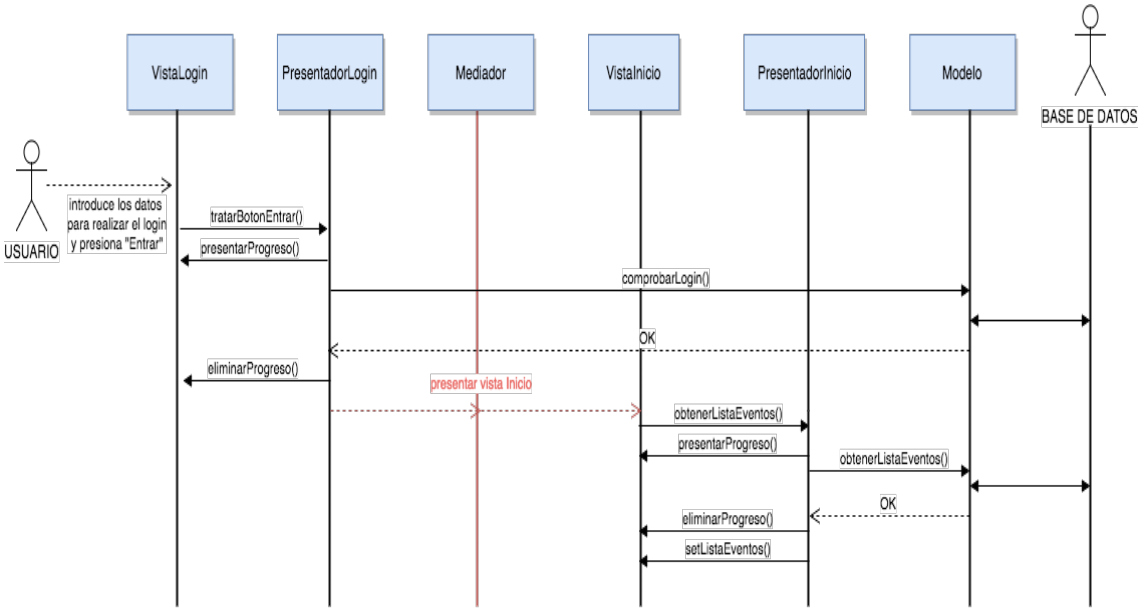

Figura 3.2: Diagrama de secuencia relacionado con el inicio de sesión en la aplicación

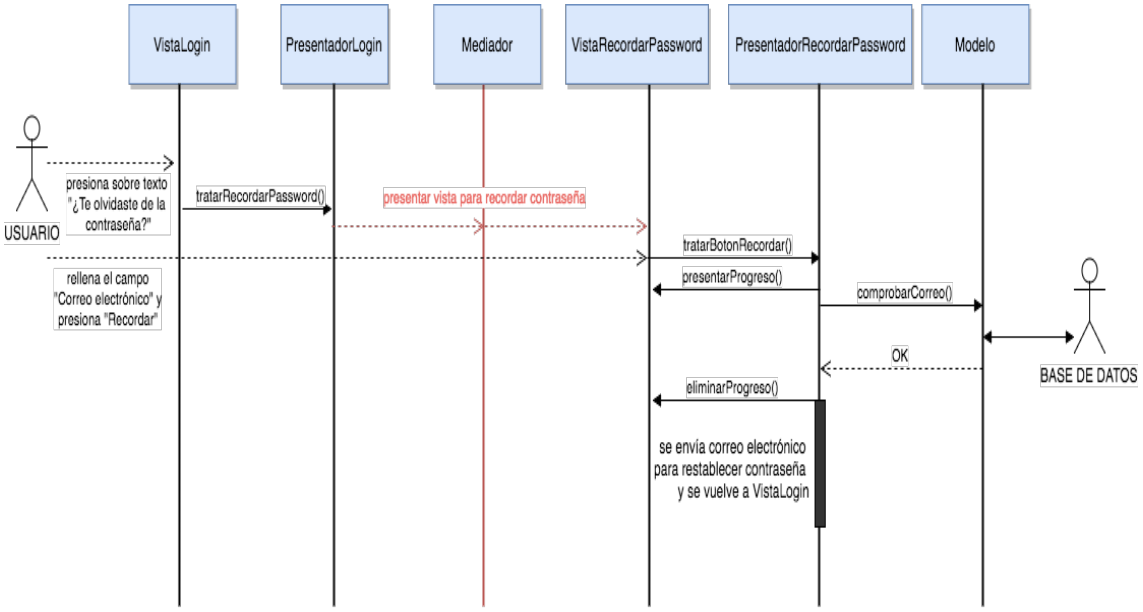

Figura 3.3: Diagrama de secuencia relacionado con la recuperación de la contraseña

Por otro lado, cuando un usuario ha olvidado su contraseña y desea solicitar que se la recuerden, ocurren los eventos que se muestran en la figura 3.3. Para ello, el usuario presiona sobre el texto "¿Te olvidaste de la contraseña?" (que se puede observar en la figura 2.4), el presentador se encargará de pedir al mediador de la aplicación que lance la vista relacionada con la recuperación de la contraseña (pantalla que se muestra a la derecha de la figura 2.5). En esa vista, el usuario se encargará de insertar su correo electrónico, que será recuperado por el presentador,

para comunicarlo al modelo y, tras la comprobación oportuna, enviar un correo electrónico al usuario, con su contraseña.

Asimismo, también se puede dar la posibilidad que el usuario presione sobre el texto "¿No tienes cuenta?". Cuando esto ocurre, la vista solicita al presentador que trate el evento, encargándose éste de presentar al usuario un mensaje a través de la vista. Este mensaje le indica al usuario que la aplicación tiene un sistema de registro cerrado, así que debe solicitar una cuenta de usuario al encargado de su clínica de fisioterapia. Este flujo de mensajes se muestra en la figura 3.4.

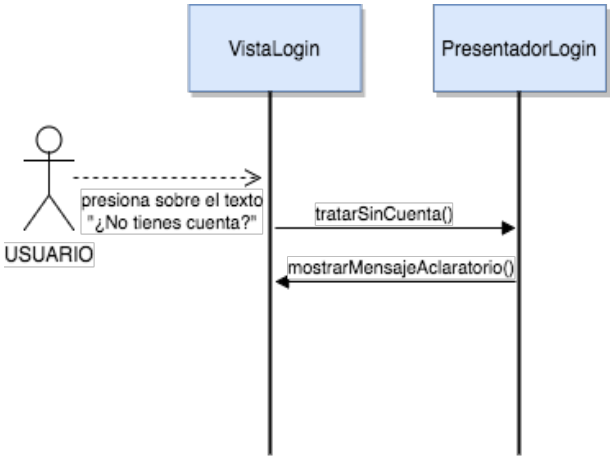

Figura 3.4: Diagrama de secuencia relacionado con la muestra del mensaje aclaratorio sobre la habilitación del registro

3.2.2 **Diagrama de secuencia relacionado con el ítem Inicio del menú**

En la figura 3.5 se presenta el flujo de mensajes intercambiados entre la vista, el presentador y el modelo, cuando el usuario accede al ítem "Inicio" del menú de navegación de la aplicación (figura 2.7). Este menú de navegación se encuentra en varias vistas, y serán ellas las que indiquen a su presentador que el ítem de menú específico para "Inicio" ha sido presionado, por lo que el presentador correspondiente se encargará de solicitar al mediador de la aplicación que se cargue la nueva vista. Al cargarse la vista, ésta le solicita a su presentador que realice las acciones oportunas, que en este caso será cargar la lista de eventos del día actual para la clínica de fisioterapia del usuario. Para realizar esta acción, el presentador le solicitará al modelo los datos, que una vez recuperados de la base de datos, se le entregarán al presentador, que los hará llegar a la vista, para mostrarlos finalmente al usuario. El resultado final se puede apreciar en la figura 2.6.

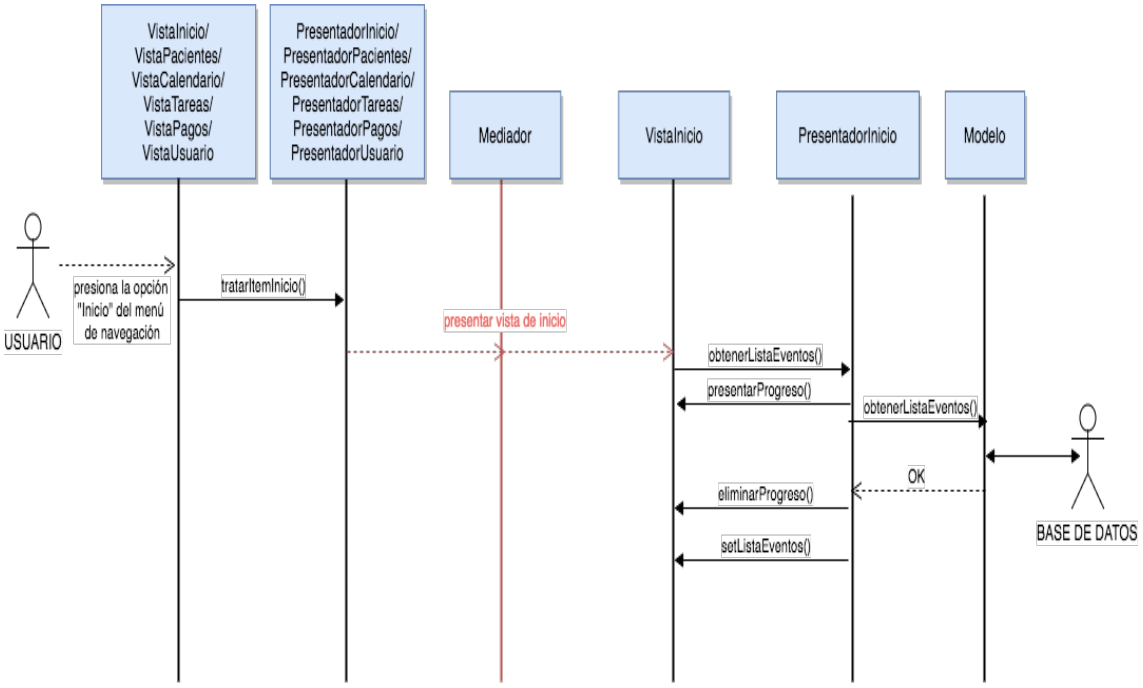

Figura 3.5: Diagrama de secuencia relacionado con la visualización de la lista de eventos del día actual tras acceder al item de menú *Inicio*

3.2.3 **Diagramas de secuencia relacionados con el ítem Pacientes del menú**

En la figura 3.6 se presenta el flujo de mensajes intercambiados entre la vista, el presentador y el modelo, cuando el usuario accede al ítem "Pacientes" del menú de navegación de la aplicación (figura 2.7). Este menú de navegación se encuentra en varias vistas, y serán ellas las que indiquen a su presentador que el ítem de menú específico para "Pacientes" ha sido presionado, por lo que el presentador correspondiente se encargará de solicitar al mediador de la aplicación que se cargue la nueva vista. Al cargarse la vista, ésta le solicita a su presentador que realice las acciones oportunas, que en este caso será cargar la lista de pacientes que tiene la clínica de fisioterapia a la que pertenece el usuario. Para realizar esta acción, el presentador acudirá al modelo con el objetivo de obtener estos pacientes y, una vez obtenidos, los comunicará a la vista, para que ésta, finalmente, pueda mostrarlos al usuario. El resultado de este diagrama de secuencia se aprecia en la figura 2.8.

Cuando se presiona un paciente de la lista, se accede a su ficha. La relación de mensajes enviados entre la vista, el presentador y el modelo para esta acción, se muestra en el diagrama de secuencia de la figura 3.7. Así, una vez que el usuario seleccione un paciente de la lista, la vista delegará el tratamiento del evento a su presentador. Éste, conociendo el paciente seleccionado por el usuario, pedirá al mediador de la aplicación que lance la vista que presenta la ficha de un paciente. Esta vista, cuyas pantallas se muestran en la figura 2.9, contiene tres solapas, que

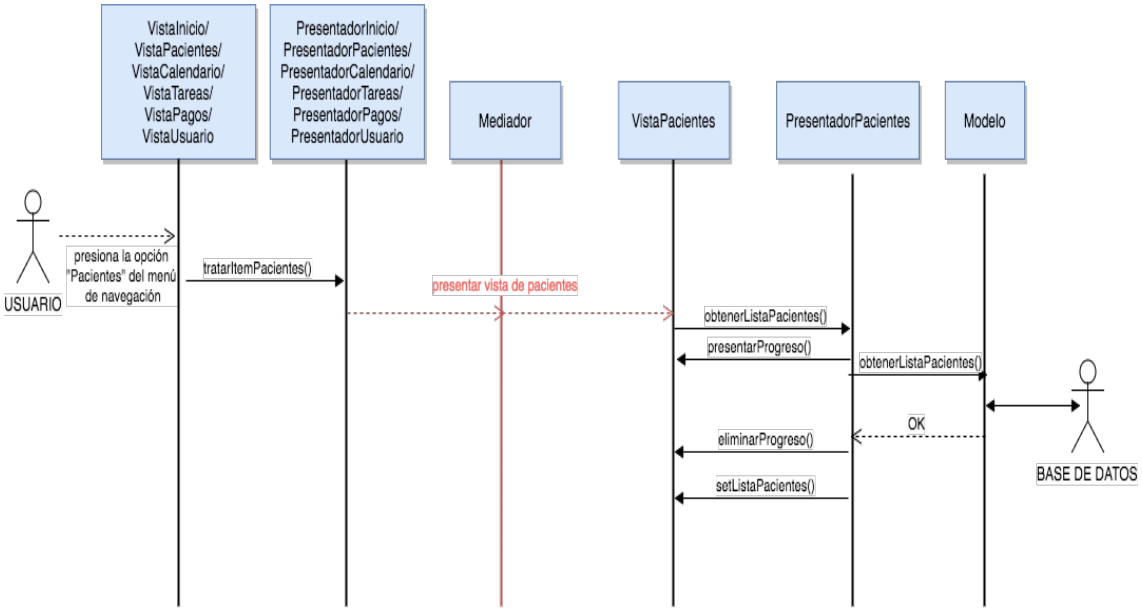

Figura 3.6: Diagrama de secuencia relacionado con la visualización de la lista de pacientes tras acceder al item de menú *Pacientes*

distinguen tres partes diferentes de la ficha del paciente. Cuando esta vista es lanzada, inicialmente se carga la solapa "INFO", que muestra la información personal del paciente y sus próximos eventos en la clínica. Esta información será solicitada al presentador, que se encargará de obtenerla mediante la petición correspondiente al modelo, una para la información personal del paciente y otra para sus eventos próximos. Una vez obtenida toda esta información, el presentador se la entregará a la vista para mostrarla finalmente al usuario. Las otras dos solapas, "IMÁGENES" y "NOTAS", como sus nombres indican, mostrarán imágenes y notas del paciente. Cuando se accede a cualquiera de estas solapas, el proceso de obtención de la información es el mismo, tal como se aprecia en el diagrama de la figura 3.7, con la diferencia de la información a obtener como tal (imágenes o notas): la vista solicitará al presentador que desea determinada información para mostrar, encargándose, por tanto, el presentador de las gestiones necesarias con el modelo. Cuando esta información le es comunicada al presentador, por parte del modelo, será transmitida a la vista, para finalmente mostrársela al usuario.

Dentro de la solapa "INFO" se da la posibilidad de añadir un evento para el paciente que se está visualizando. La figura 3.8 muestra el diagrama de secuencia para esta situación, en la que simplemente el presentador es notificado por la vista de que el botón específico para esta acción ha sido presionado, y por tanto comunicará al mediador que lance la nueva vista para agregar este nuevo evento para el paciente.

Además de poder visualizar la ficha de los pacientes, se puede agregar nuevos pacientes así como editar o eliminar los ya existentes. La figura 3.9 muestra el diagrama de secuencia para cuando se desea añadir un paciente (pantallas que se muestran en la figura 2.11). La vista de la lista de pacientes, tras detectar que se

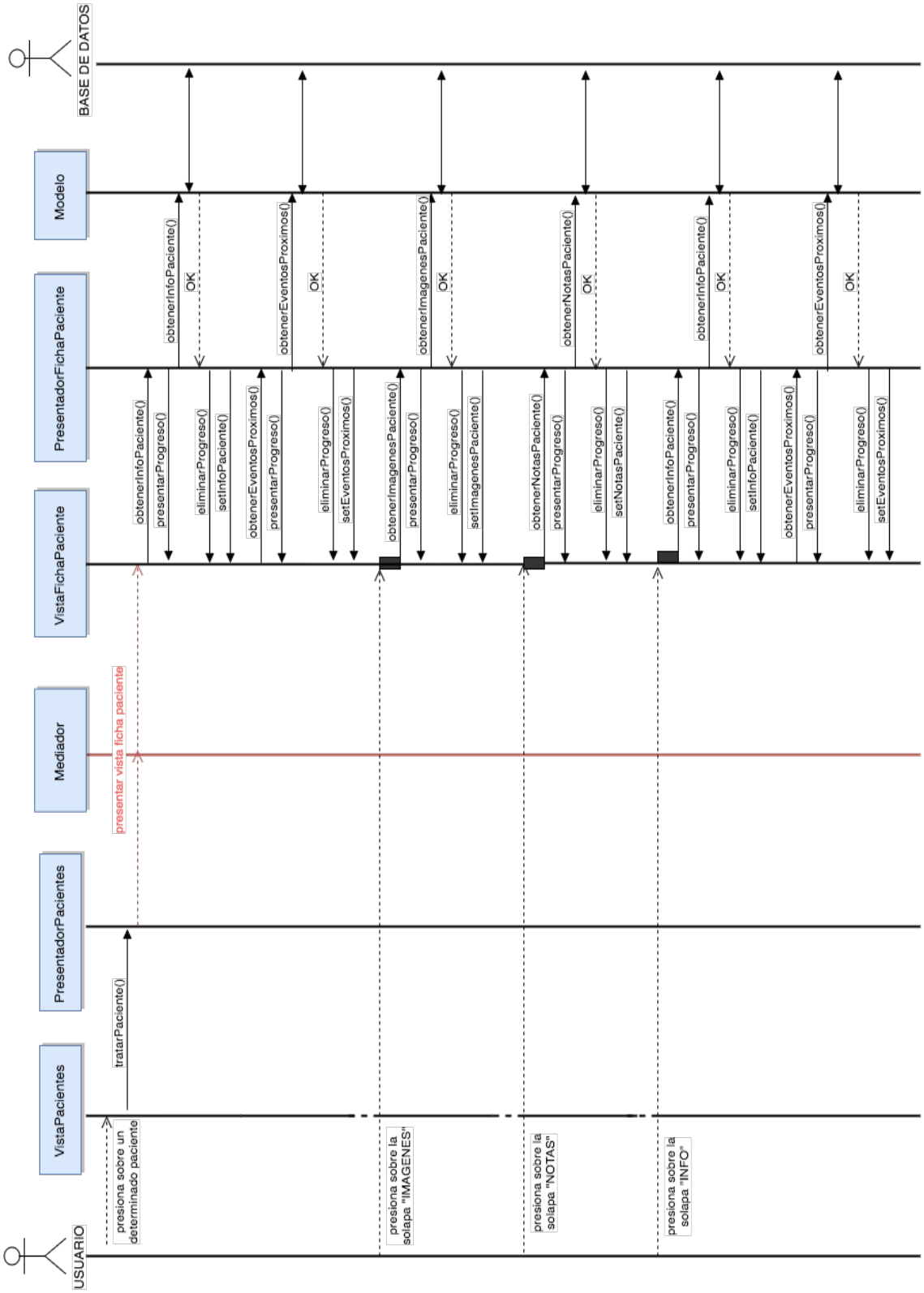

Figura 3.7: Diagrama de secuencia relacionado con la visualización de la ficha de paciente seleccionado

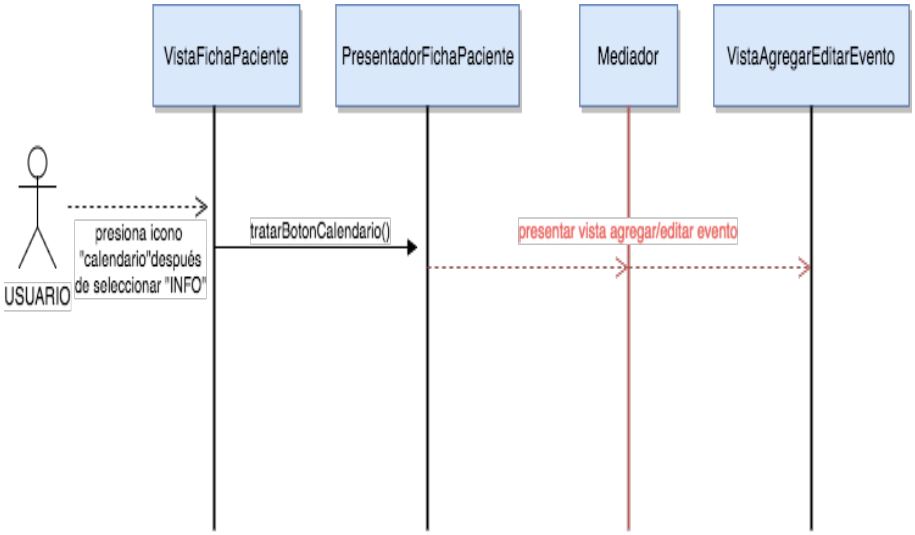

Figura 3.8: Diagrama de secuencia relacionado con la agregación de un evento para el paciente a través de su ficha

ha presionado el botón específico para añadir pacientes, comunica al presentador este evento, y éste continúa el proceso solicitando al mediador de la aplicación el lanzamiento de la vista dedicada a agregar un paciente.

La vista para agregar un paciente contiene una serie de campos, que cuando son rellenados y se presiona el botón correspondiente a la finalización del formulario, el presentador capta este evento y procede, pidiendo al modelo la adición de este nuevo paciente. Al finalizarse el proceso, el presentador pide esta vez al mediador la vuelta a la vista de la lista de pacientes, donde se podrá comprobar que el nuevo paciente se ha agregado a la lista.

En cuanto a la edición de un paciente, la figura 3.10 presenta el diagrama de secuencia correspondiente, en el que se puede apreciar la similitud con el explicado anteriormente, ya que comparten la misma vista (estas pantallas se muestran en la figura 2.12). La diferencia reside, además de que se accede a través de la vista de la ficha del paciente, en que tras haber hecho los cambios correspondientes y la vista haber comunicado al presentador la finalización de los mismos, el presentador solicitará al modelo que haga una actualización de los datos del paciente.

Por último, la eliminación de un paciente tiene un diagrama de secuencia que se muestra en la figura 3.11. Así, en la vista de la ficha del paciente existe un botón dedicado a la eliminación del paciente. Cuando éste se presiona, la vista lo comunica al presentador, que pedirá a la vista mostrar un diálogo en el que se solicita la confirmación del usuario (se muestra en la figura 2.13). Si el usuario reafirma su deseo de eliminar el paciente, el presentador será informado y procederá a solicitar al modelo, que elimine al paciente de la base de datos. Finalmente, el presentador solicitará al mediador de la aplicación volver a la vista de la lista de pacientes, donde el paciente ya no estará presente.

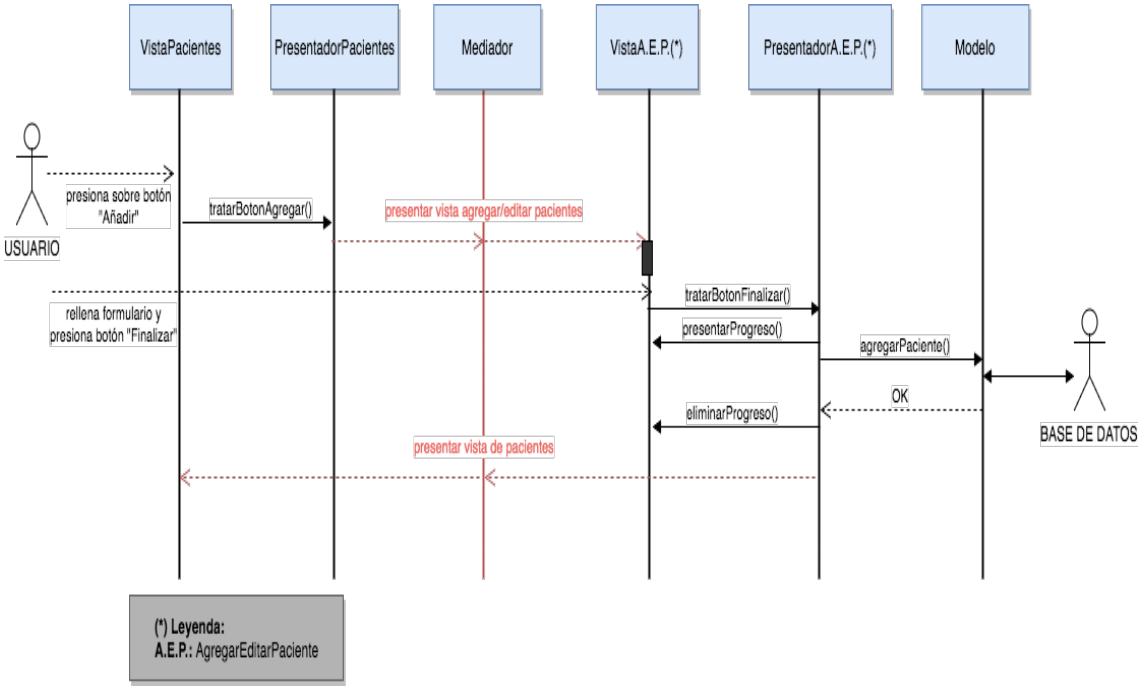

Figura 3.9: Diagrama de secuencia relacionado con la agregación de un paciente

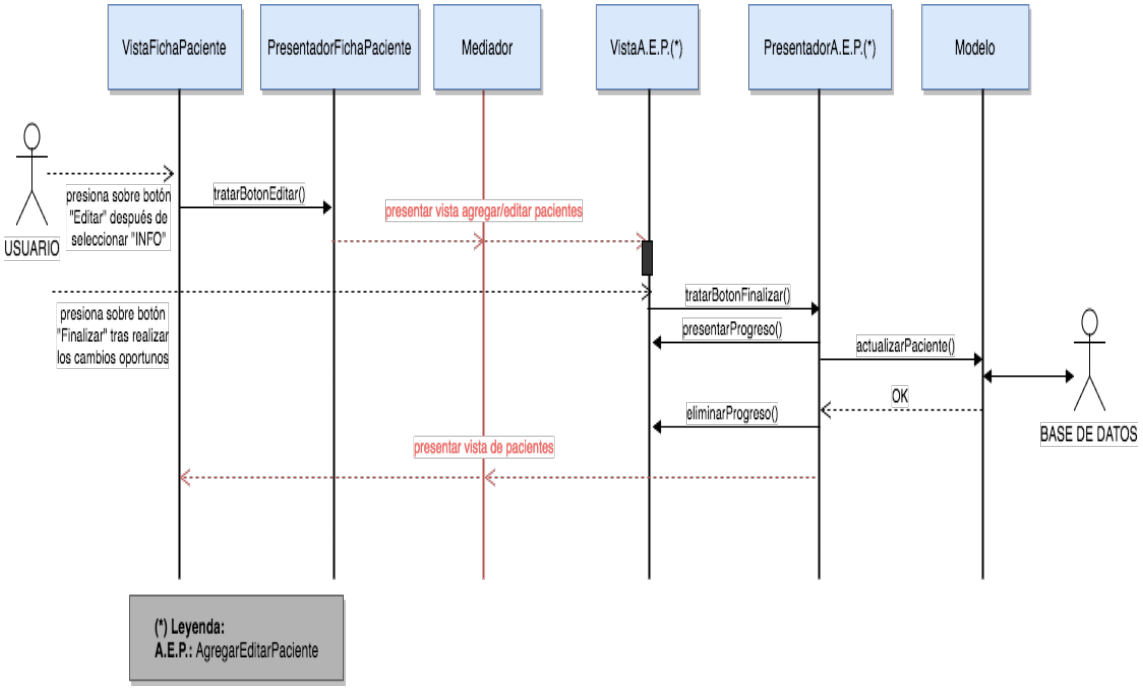

Figura 3.10: Diagrama de secuencia relacionado con la edición de los datos de un paciente

Como se mencionó previamente, la solapa “IMÁGENES” de la vista de la ficha del paciente, muestra el archivo de imágenes del paciente. Estas imágenes pueden

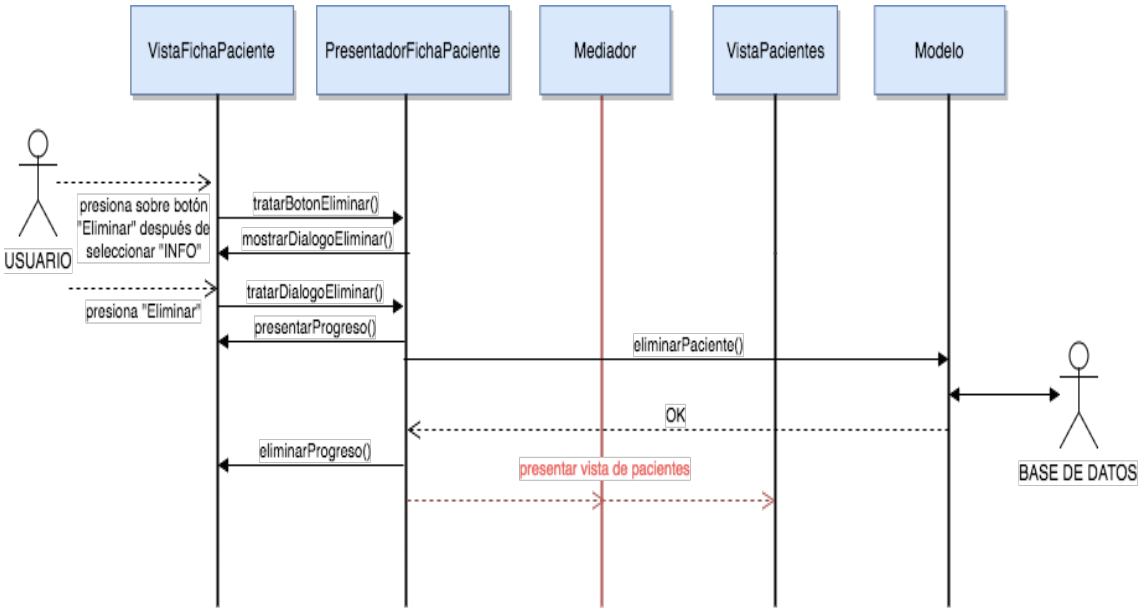

Figura 3.11: Diagrama de secuencia relacionado con la eliminación de un paciente

ser visualizadas al detalle, además de añadir nuevas o eliminar alguna de las ya existentes (pantallas que se muestran en la figura 2.14). En el caso de querer añadir una nueva imagen, el proceso se representa en el diagrama de secuencia de la figura 3.12. Una vez se accede a la solapa indicada, al presionar sobre el botón específico para añadir una imagen, la vista comunicará al presentador este evento y éste pide al mediador de la aplicación acceder a la vista para agregar la imagen. Una vez rellenado los campos del formulario, se confirma la finalización a través del botón habilitado para ello. Cuando se presiona y la vista lo comunica al presentador, éste acudirá al modelo para que se proceda a agregar dicha imagen al archivo de imágenes del paciente, en la base de datos. Si la agregación de la imagen ha sido un éxito, el presentador solicitará al mediador volver a la vista de la ficha de paciente, en su solapa “IMÁGENES”.

Por otro lado, si se desea visualizar cualquiera de estas imágenes de manera ampliada, además de poder conocer su descripción, basta con presionar sobre una de ellas. Este evento es captado por la vista, que será transmitido al presentador, para que el mediador muestre la vista dedicada a la visualización de las imágenes al detalle. Al cargar esta vista, se necesita presentar la imagen, por tanto es necesario obtenerla de la lista de imágenes previamente descargada, así que la vista lo pide al presentador, el cual hace la petición al modelo, para que le entregue la imagen a visualizar. Finalmente, ésta será enviada a la vista, que se encargará de mostrarla, además de la descripción de la misma en el campo habilitado para ello. El diagrama de secuencia que muestra este flujo de mensajes intercambiados se encuentra en la figura 3.13.

En la vista de visualización de la imagen, se encuentra habilitado un botón para eliminar la imagen. En este caso, el diagrama de secuencia está representado en la

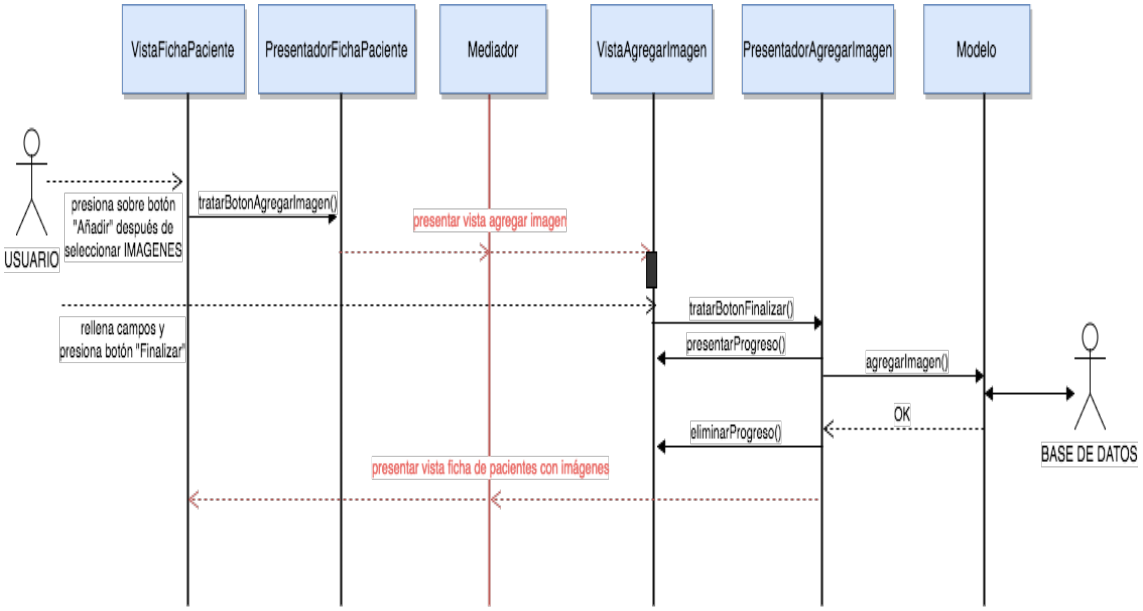

Figura 3.12: Diagrama de secuencia relacionado con la agregación de una imagen imagen

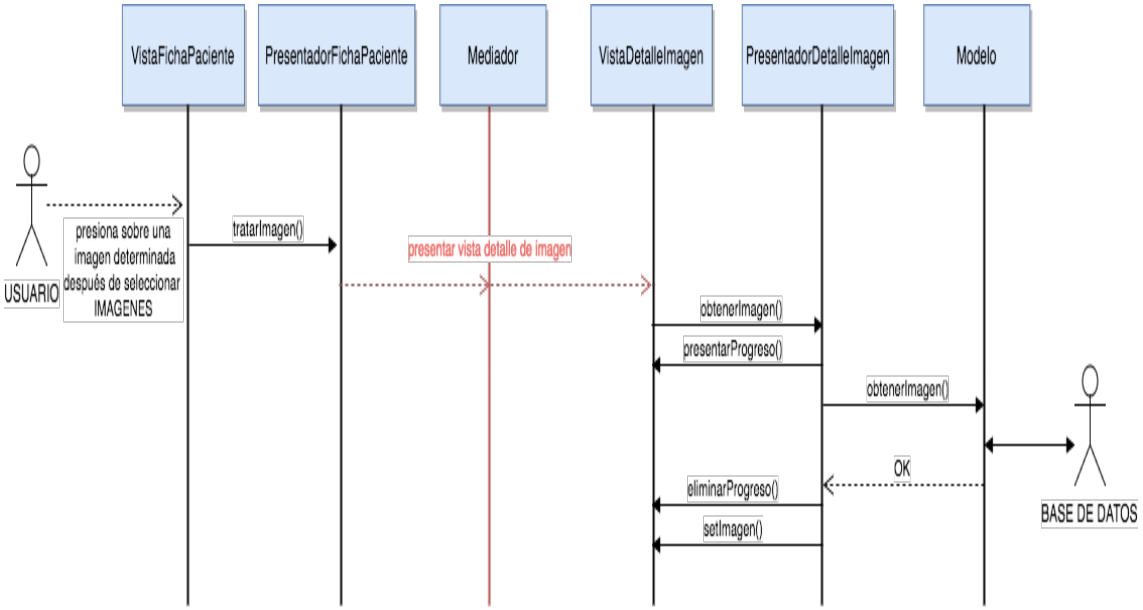

Figura 3.13: Diagrama de secuencia relacionado con la visualización del detalle de una imagen

figura 3.14. Cuando el usuario selecciona este botón, la vista comunica el evento al presentador que procederá a comunicarle al modelo la eliminación de la imagen de la base de datos. El modelo ejecuta dicha eliminación y tras ello el presentador pedirá al mediador volver a la vista de la ficha del paciente, en concreto en la solapa "IMÁGENES".

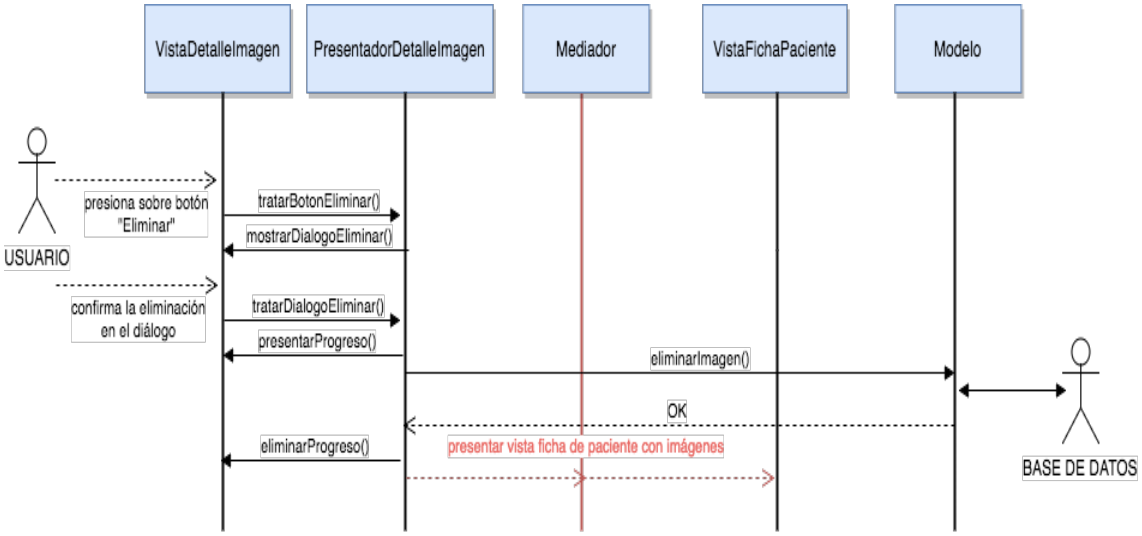

Figura 3.14: Diagrama de secuencia relacionado con la eliminación de una imagen

La última solapa de la ficha de paciente que queda por explicar es la solapa "NOTAS". Además de visualizar las notas, como ya se explicó en el diagrama de secuencia de la figura 3.7, se pueden añadir nuevas notas, además de editar y eliminar las ya existentes (pantallas que se muestran en la figura 2.15). En el caso de añadir una nueva nota, el diagrama de secuencia se muestra en la figura 3.15. Para acceder a la vista dedicada a la adición de notas, existe un botón en esta solapa, que al seleccionarlo, la vista delegará en el presentador para que se encargue de solicitar al mediador de la aplicación que cargue la vista para añadir la nota. Una vez en esta vista, el usuario dispondrá de un formulario que ha de rellenar y de un botón para finalizar. Cuando el usuario presione el botón, la vista comunicará este evento al presentador, para que haga las tareas necesarias para agregar la nota, por medio del modelo, a la base de datos. Cuando el modelo termine de añadirla, el presentador pedirá al mediador volver a la vista de la ficha del paciente, en su solapa "NOTAS".

En el caso de la edición de las notas, se puede comprobar la similitud del diagrama de secuencia dedicado a ello (figura 3.16) con el anteriormente explicado, y es debido a que se utiliza la misma vista para esta acción. Las diferencias residen en que para acceder a ella, en este caso, se hace a través de un botón superpuesto en cada nota, y en que una vez hecho los cambios y la vista le solicite al presentador que los trate, éste pide al modelo una actualización de la información de la nota.

Por último, el diagrama de secuencia dedicado a explicar el flujo de mensajes cuando se dispone a la eliminación de una nota, se encuentra en la figura 3.17. Además del botón para editar una nota, también se encuentra otro botón dedicado a eliminar una nota. Cuando este se selecciona, la vista lo indica al presentador, el cuál interactúa de nuevo con la vista para mostrar un diálogo para pedir la confirmación de la eliminación al usuario. La vista delegará en el presentador para que proceda a la eliminación, para que finalmente el modelo se encargue de ello.

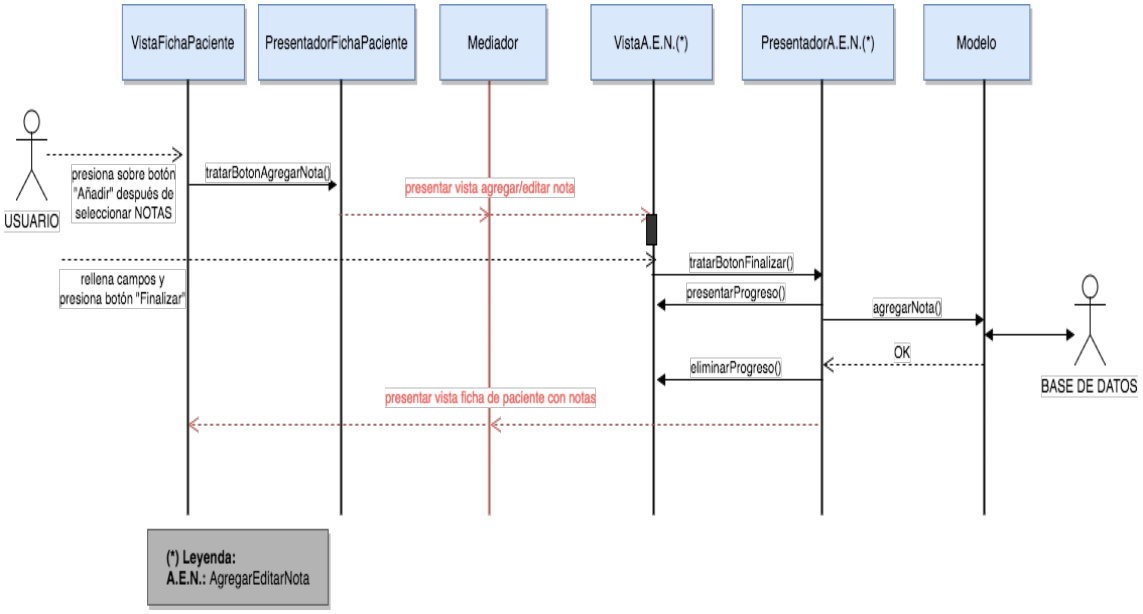

Figura 3.15: Diagrama de secuencia relacionado con la agregación de una nota

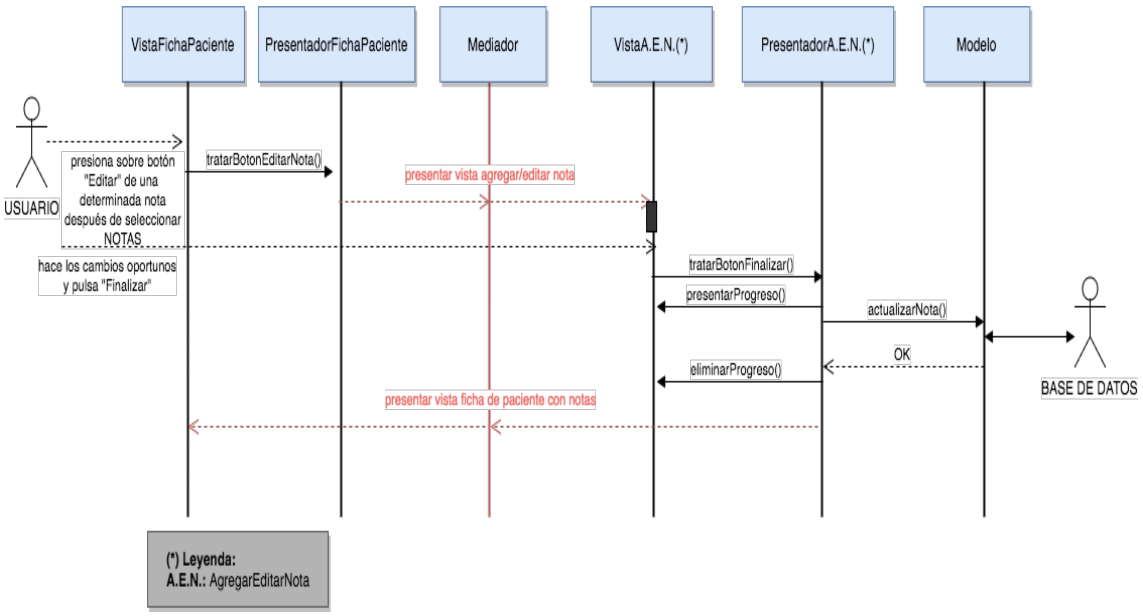

Figura 3.16: Diagrama de secuencia relacionado con la edición de una nota

3.2.4 **Diagramas de secuencia relacionados con el ítem Calendario del menú**

En la figura 3.18 se presenta el flujo de mensajes intercambiados entre la vista, el presentador y el modelo, cuando el usuario accede al ítem “Calendario” del menú de navegación de la aplicación (figura 2.7). Este menú de navegación se encuentra en varias vistas, y serán ellas las que indiquen a su presentador que el ítem de

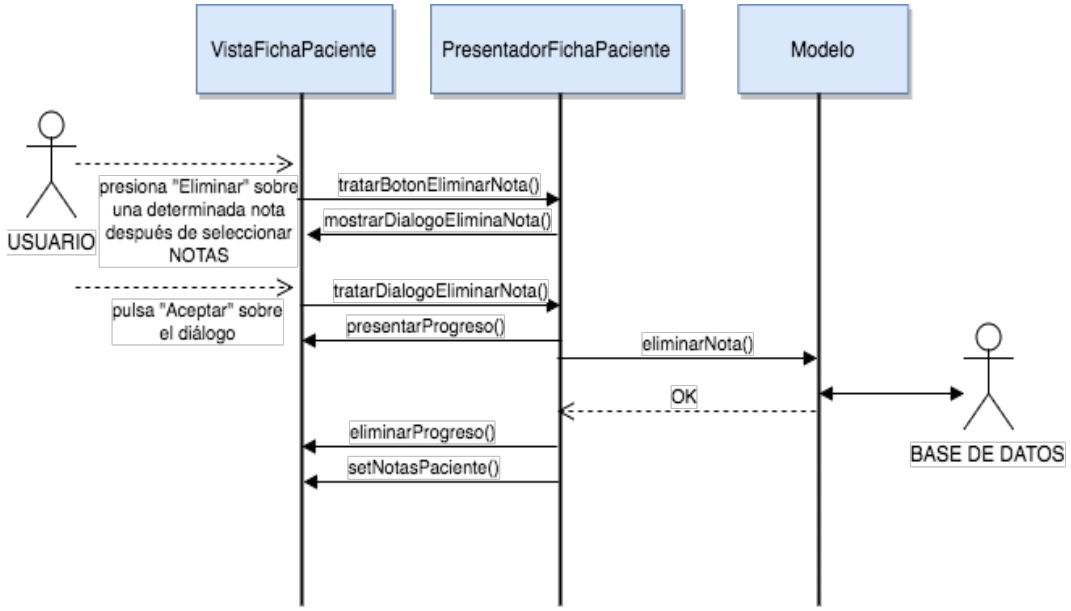

Figura 3.17: Diagrama de secuencia relacionado con la eliminación de una nota

menú específico para "Calendario" ha sido presionado, por lo que el presentador correspondiente se encargará de solicitar al mediador de la aplicación que cargue la vista del calendario.

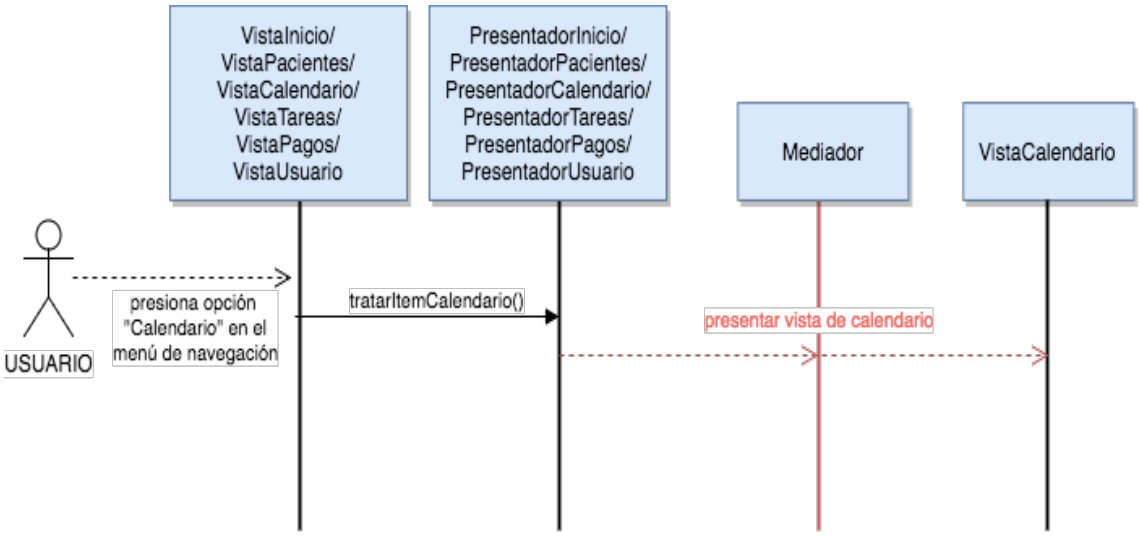

Figura 3.18: Diagrama de secuencia relacionado con la visualización del calendario

En la vista del calendario (figura 2.16) el usuario puede seleccionar una determinada fecha y visualizar la lista de eventos que están registrados para el día seleccionado. El diagrama de secuencia de la figura 3.19 explica el flujo de eventos necesario para ello: en primer lugar el usuario ha de presionar sobre un día del calendario. La vista informa al presentador de que el usuario ha presionado sobre

un día, y éste solicita al mediador que lance la vista dedicada a los eventos. El día seleccionado servirá para la obtención de la lista de eventos, ya que el modelo sabrá reconocer qué eventos recuperaar, tras habérselo solicitado el presentador. La lista de eventos se le proporcionará al presentador, que la transmitirá a la vista para que sea mostrada al usuario. El resultado final de estos sucesos se puede apreciar en la figura 2.17.

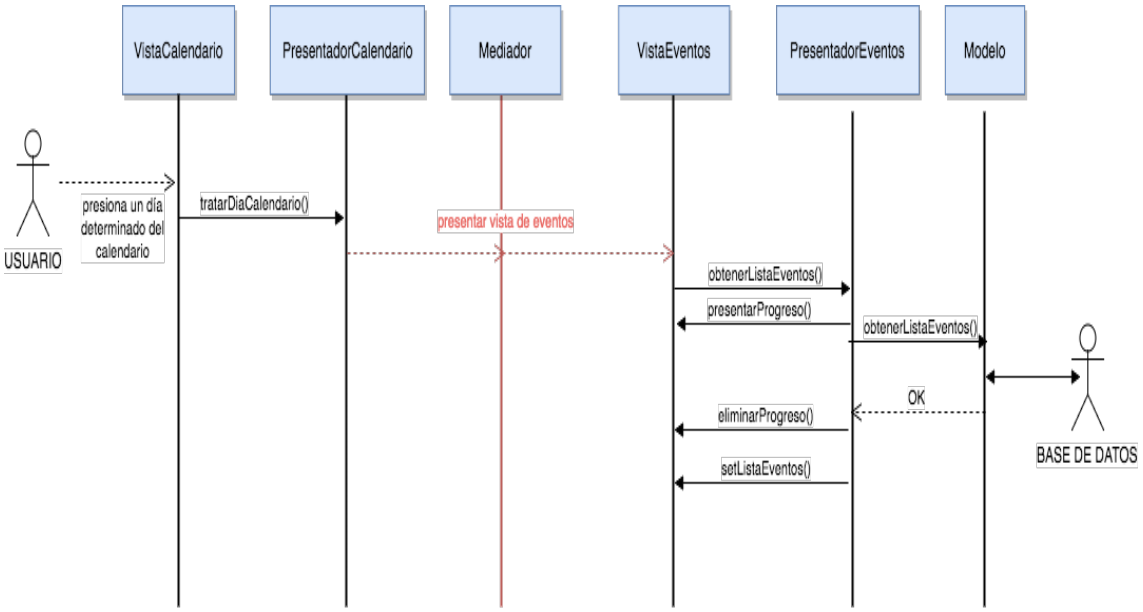

Figura 3.19: Diagrama de secuencia relacionado con la visualización de la lista de eventos del día seleccionado

En lo que respecta a la gestión de los eventos de la lista, se pueden realizar diferentes acciones sobre ellos: añadir nuevos eventos y también editar y eliminar eventos que ya existen. En primer lugar, el diagrama de secuencia que se muestra en la figura 3.20. Existe una vista dedicada a la adición de eventos, y se puede acceder a ella tanto desde la vista del calendario como desde la vista de la lista de eventos, ambas utilizando un botón específico para ello. Cuando este evento es tratado por el presentador, se solicita al mediador que se lance la nueva vista con el objetivo de añadir un nuevo evento. En ella existirá un formulario a rellenar por el usuario, y cuando finalice con él, tras presionar el botón correspondiente, la vista lo notificará al presentador, y este se encargará de proporcionar la información insertada por el usuario al modelo, para que ejecute la adición y, por tanto, añadir el nuevo evento. Finalmente, suponiendo que todo ha ido correctamente, se volverá a la vista de la lista de eventos, puesto que el presentador lo pedirá al mediador.

Por otro lado está la edición de eventos, cuyo diagrama de secuencia se encuentra representado en la figura 3.21. Como se puede apreciar, es prácticamente igual al explicado previamente, puesto que se comparte la misma vista para ambas acciones. Las únicas diferencias son dos: el acceso a la vista de edición de eventos se hace a través de un botón habilitado en cada evento y que, una vez finalizados los cambios

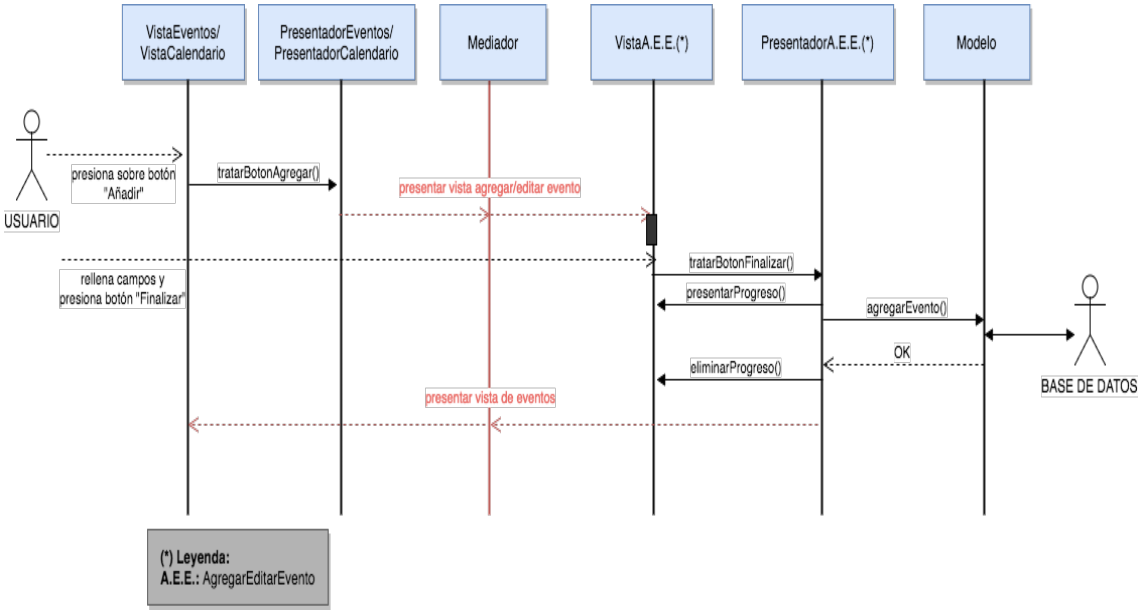

Figura 3.20: Diagrama de secuencia relacionado con la agregación de un evento

a realizar en la vista para la edición de eventos, el presentador pedirá al modelo, en este caso, que se actualice el evento.

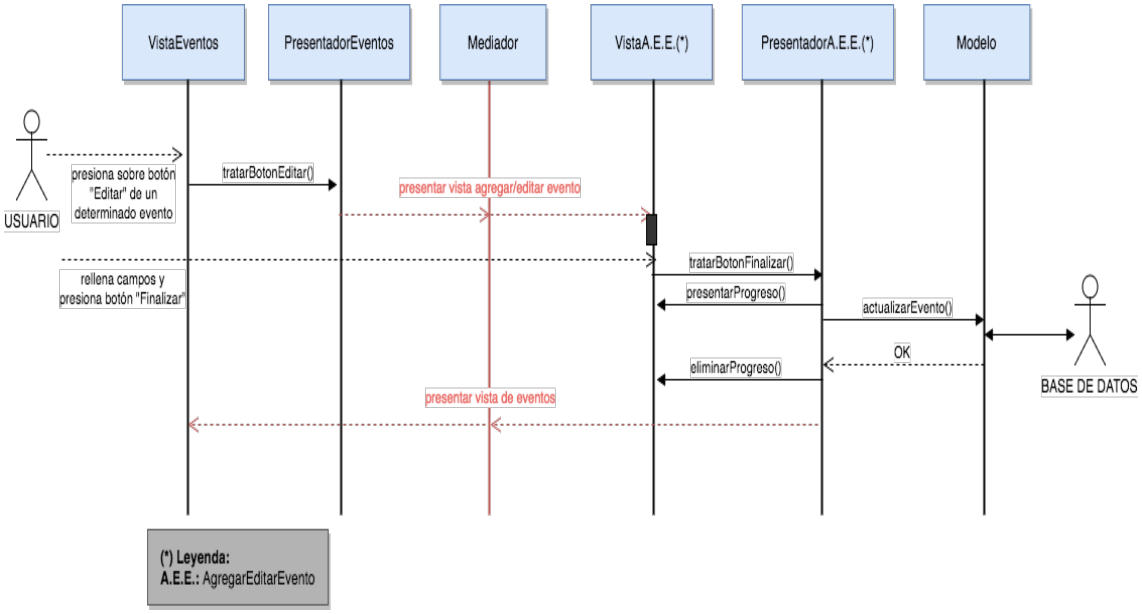

Figura 3.21: Diagrama de secuencia relacionado con la edición de un evento

La acción restante en cuanto a la gestión de eventos es la eliminación de los mismos, y se muestra el flujo de eventos en el diagrama de secuencia de la figura 3.22. Además del botón para la edición del evento, existe otro también para eliminar. Cuando es presionado, el presentador tratará este evento solicitando a la vista que

muestre un diálogo que requiere de la confirmación del usuario para continuar con la eliminación. Si el usuario accede finalmente a esta eliminación, el presentador se encargará de comunicar al modelo que se debe eliminar el evento de la base de datos. Una vez esto haya ocurrido, se mostrará de nuevo la lista de eventos actualizada.

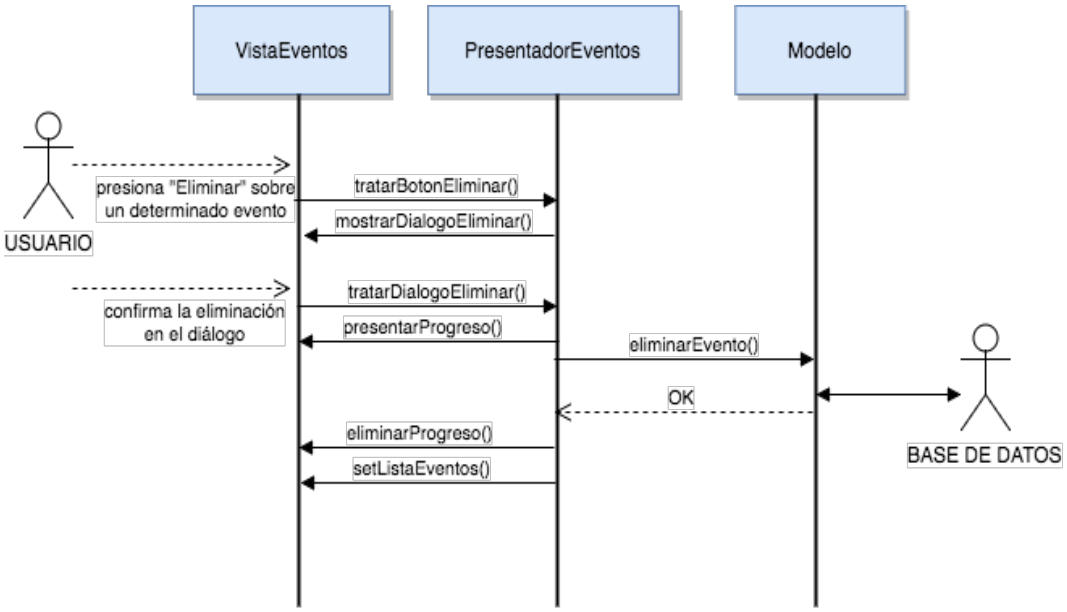

Figura 3.22: Diagrama de secuencia relacionado con la eliminación de un evento

3.2.5 Diagramas de secuencia relacionados con el ítem Tareas del menú

En la figura 3.23 se presenta el flujo de mensajes intercambiados entre la vista, el presentador y el modelo, cuando el usuario accede al ítem "Tareas" del menú de navegación de la aplicación (figura 2.7). Este menú de navegación se encuentra en varias vistas, y serán ellas las que indiquen a su presentador que el ítem de menú específico para "Tareas" ha sido presionado, por lo que su presentador se encargará de solicitar al mediador de la aplicación, que se cargue la vista de tareas. Por tanto, cuando esta vista es cargada, será necesario que muestre la lista de tareas, la cual delegará en su obtención al presentador, que se encargará de interactuar con el modelo para solicitar y recibir esta lista. Cuando el modelo haya recuperado la lista y se la notifique al presentador adecuado, éste se la entregará a la vista de tareas para que la muestre al usuario. El resultado final se puede observar en la vista de la figura 2.20.

En cuanto al manejo de estas tareas, existen dos acciones sobre ellas: añadir nuevas tareas y finalizar las ya existentes. Respecto a añadir una nueva tarea al listado, el diagrama de secuencia que representa este proceso se encuentra en la figura 3.24. El usuario puede añadir una nueva tarea al presionar sobre el botón específico

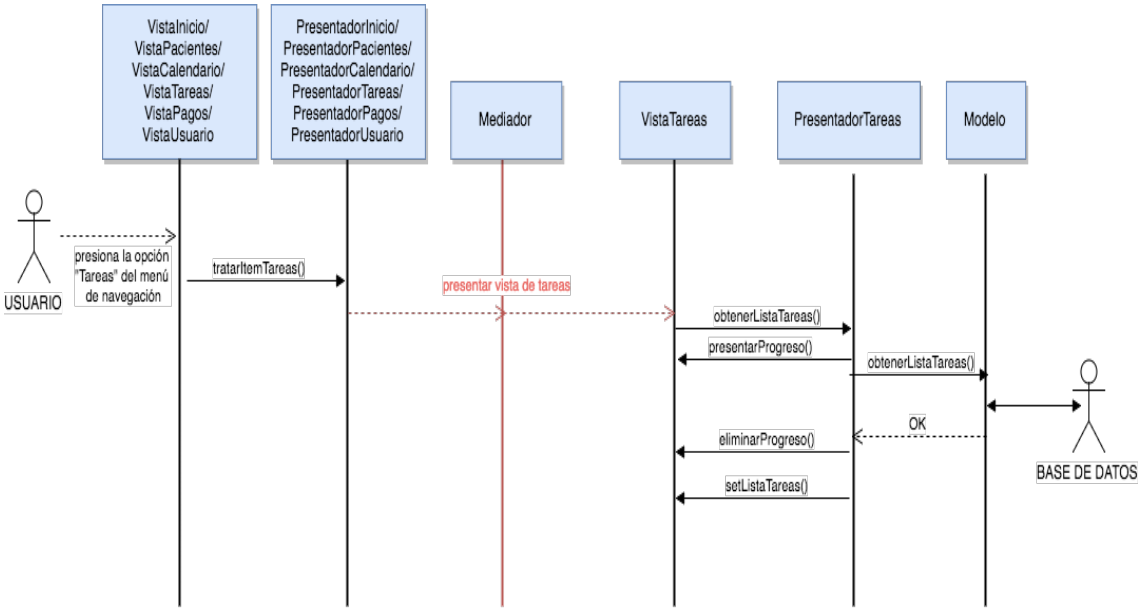

Figura 3.23: Diagrama de secuencia relacionado con la visualización de la lista de tareas

para ello, en la vista de la lista de tareas. Cuando esto ocurre, el presentador trata este evento encargando a la vista mostrar un diálogo con un formulario a rellenar por el usuario (pantalla que se muestra en la figura 2.21). Una vez el usuario lo rellene y presione sobre el botón para finalizar, la vista comunicará este hecho al presentador, que continuará a su vez la comunicación con el modelo, solicitando la adición de esta nueva tarea. Finalizada esta operación, el presentador volverá a interactuar con la vista, para que ésta recargue la nueva lista de tareas, en la que aparece la nueva tarea añadida recientemente.

Por otro lado, como se mencionó previamente, existe la posibilidad de finalizar las tareas y, por tanto, que desaparezcan de este listado. El diagrama de secuencia para este caso es el de la figura 3.25 y explica lo siguiente: cada tarea tiene un botón contextual habilitado para esta función, por lo que cuando es presionado, el presentador recibirá el aviso de la vista, con lo que tratará el evento pidiendo al modelo que ejecute la finalización de esta tarea, es decir, la desaparición de la misma de la aplicación. Cuando este proceso haya finalizado, el presentador volverá a comunicarse con la vista, que se encargará de mostrar la lista de tareas actualizada, en la que ya no aparecerá esta tarea.

3.2.6 **Diagramas de secuencia relacionados con el ítem Pagos del menú**

En la figura 3.26 se presenta el flujo de mensajes intercambiados entre la vista, el presentador y el modelo, cuando el usuario accede al ítem “Pagos” del menú de

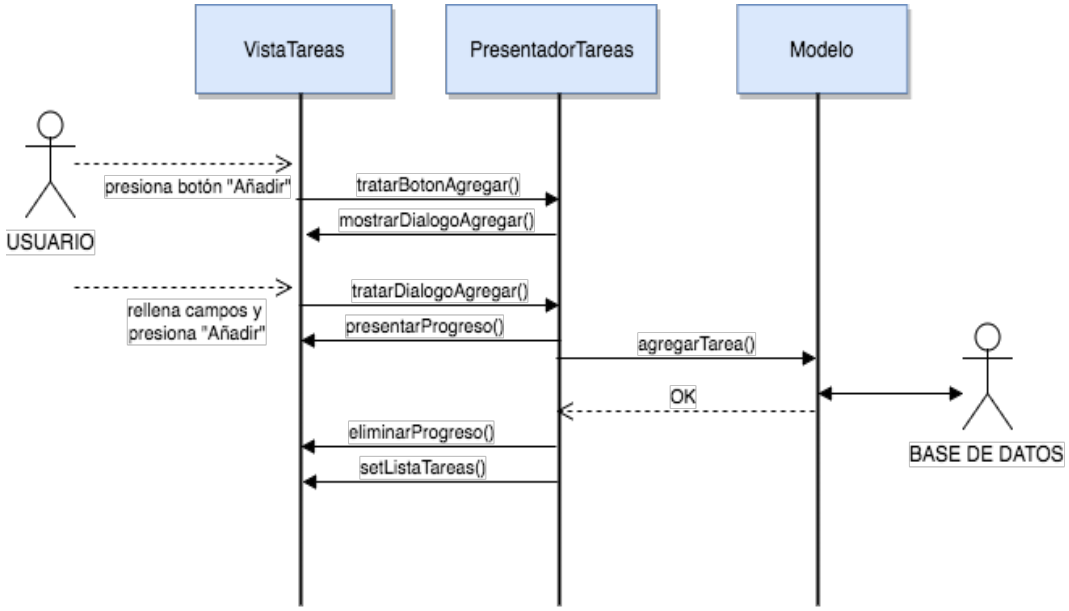

Figura 3.24: Diagrama de secuencia relacionado con la agregación de una tarea

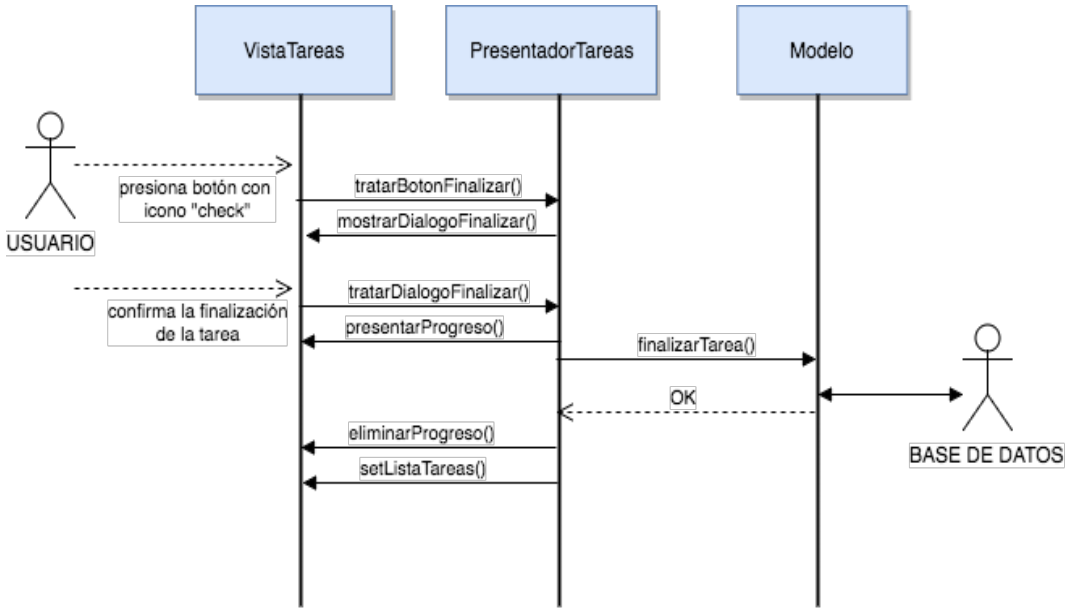

Figura 3.25: Diagrama de secuencia relacionado con la finalización de una tarea

navegación de la aplicación (figura 2.7). Este menú de navegación se encuentra en varias vistas, y serán ellas las que indiquen a su presentador que, el ítem de menú específico para "Pagos", ha sido presionado, por lo que el presentador correspondiente se encargará de solicitar al mediador de la aplicación que se cargue la nueva vista, en este caso del listado de pagos.

Esta vista presenta dos solapas (que se muestran en la figura 2.22), las cuales

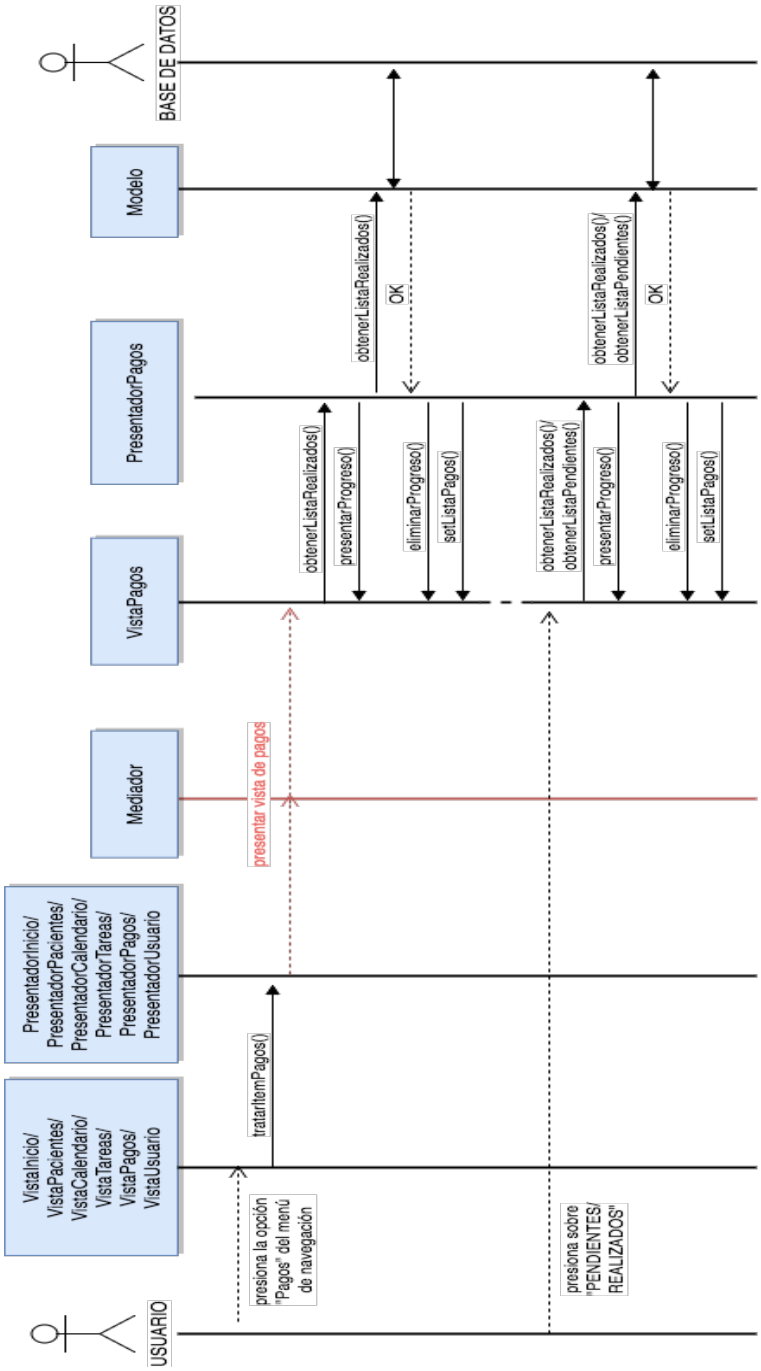

Figura 3.26: Diagrama de secuencia relacionado con la visualización de las listas de pagos pendientes y realizados

diferencian dos tipos de pagos: pagos realizados (solapa “REALIZADOS”) y pagos pendientes (solapa *PENDIENTES*). Inicialmente, al cargar esta vista, se muestra la solapa “REALIZADOS”, por lo que será necesario obtener la lista de pagos realizados. Este requisito se le hace llegar al presentador, que se encarga de interactuar con el modelo para solicitar este tipo de pagos. Una vez recibidos, el presentador se comunicará con la vista para entregarle la lista de pagos realizados, que finalmente

mostrará al usuario. En caso de que el usuario desee acceder a la solapa “PENDIENTES”, el flujo de mensajes es el mismo que el caso anterior, salvo que el presentador solicitará, en este caso, los pagos pendientes.

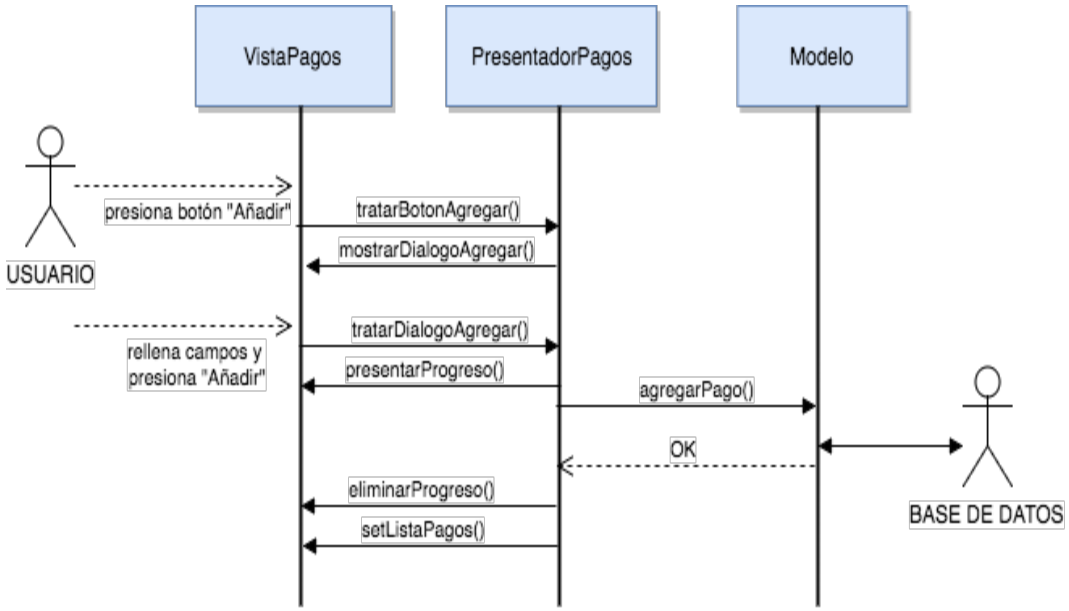

Figura 3.27: Diagrama de secuencia relacionado con la agregación de un pago

Los pagos cuentan en su haber dos tipos de acciones en lo referente a su manejo: añadir pagos y dar un pago pendiente por realizado. En el caso de que el usuario desee añadir un pago, el flujo de eventos que sucede para ello se muestra en el diagrama de secuencia de la figura 3.27. En ambas solapas se encuentra un botón específico para realizar esta tarea y cuando es presionado el presentador tratará este evento solicitando a la vista que muestre un diálogo específico para añadir pagos (se puede observar en las figuras 2.23 y 2.24). Este diálogo contiene en su haber un formulario, que será rellenado por el usuario. La vista informará al presentador que el usuario ha presionado sobre el botón para finalizar el formulario, y en ese momento el presentador hará llegar los datos al modelo para que añade este nuevo pago. Una vez realizada la adición, el presentador concluirá el proceso solicitando a la vista que muestre la lista de pagos actualizada, en la que se encuentra el nuevo pago.

Por otro lado, como se mencionó previamente, los pagos de tipo *pendiente*, es decir, los que se encuentran en la solapa “PENDIENTES”, tienen un botón que será utilizado cuando el usuario desee dar por realizado este pago, es decir, será traspasado a la lista de pagos realizados. El diagrama de secuencia que explica este proceso se encuentra en la figura 3.28 y explica lo siguiente: cuando el botón mencionado es presionado por el usuario, será comunicado este hecho por la vista al presentador, volviendo éste a interactuar con la vista al querer mostrar un diálogo concreto para solicitar la confirmación de esta tarea al usuario (se muestra en la figura 2.25). Si éste decide continuar, la vista delegará en el presentador para que

haga la tarea oportuna, que será comunicarse con el modelo para indicar que ese pago, en concreto, ha pasado a ser de tipo *realizado*. Finalmente, y como ha habido cambios en ambas listas, será necesaria una actualización de las mismas, que se presentará a través de la vista.

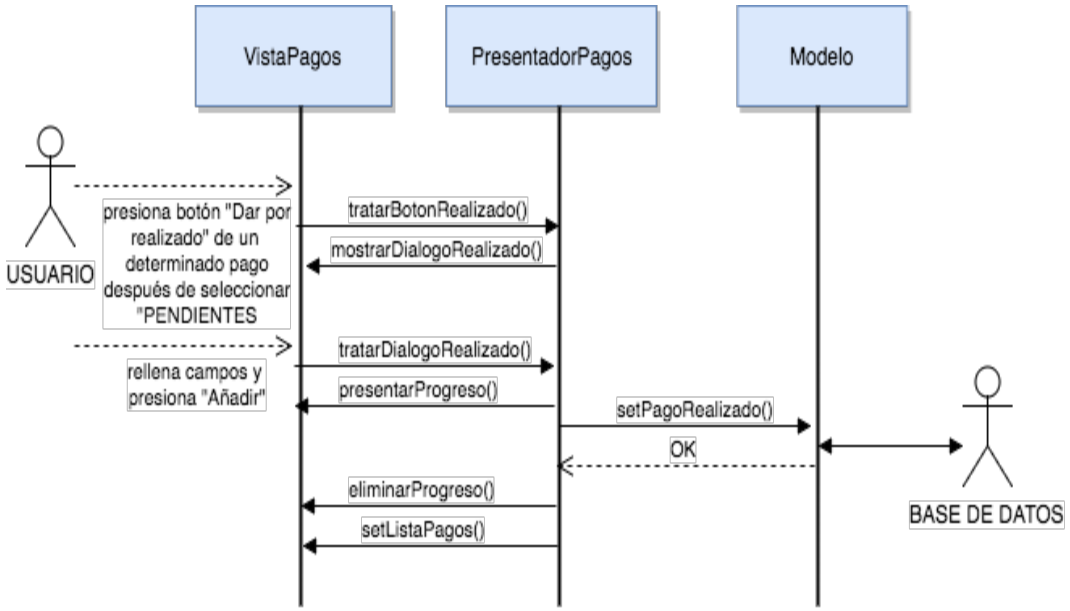

Figura 3.28: Diagrama de secuencia relacionado con el cambio de pago pendiente por realizado

3.2.7 **Diagramas de secuencia relacionados con el ítem Mi usuario del menú**

En la figura 3.29 se presenta el flujo de mensajes intercambiados entre la vista, el presentador y el modelo, cuando el usuario accede al ítem “Mi usuario” del menú de navegación de la aplicación (figura 2.7) Este menú de navegación se encuentra en varias vistas, y serán ellas las que indiquen a su presentador que el ítem de menú específico para “Mi usuario” ha sido presionado, por lo que el presentador correspondiente se encargará de solicitar al mediador de la aplicación que se cargue la nueva vista (esta vista se puede observar en la figura 2.26). Cuando esto ocurre, la vista solicita al presentador que obtenga la información del usuario que está usando la aplicación. Lo siguiente que hará el presentador es interactuar con el modelo para obtener dicha información y cuando la obtenga, entregarla a la vista para poder mostrarla al usuario.

Existen una serie de acciones que puede realizar un usuario con su cuenta en la aplicación: cambiar su contraseña, editar su información de usuario y dar de baja su cuenta. Estas tres acciones se recogen en los diagramas de secuencia de las figuras 3.30, 3.31 y 3.32, respectivamente.

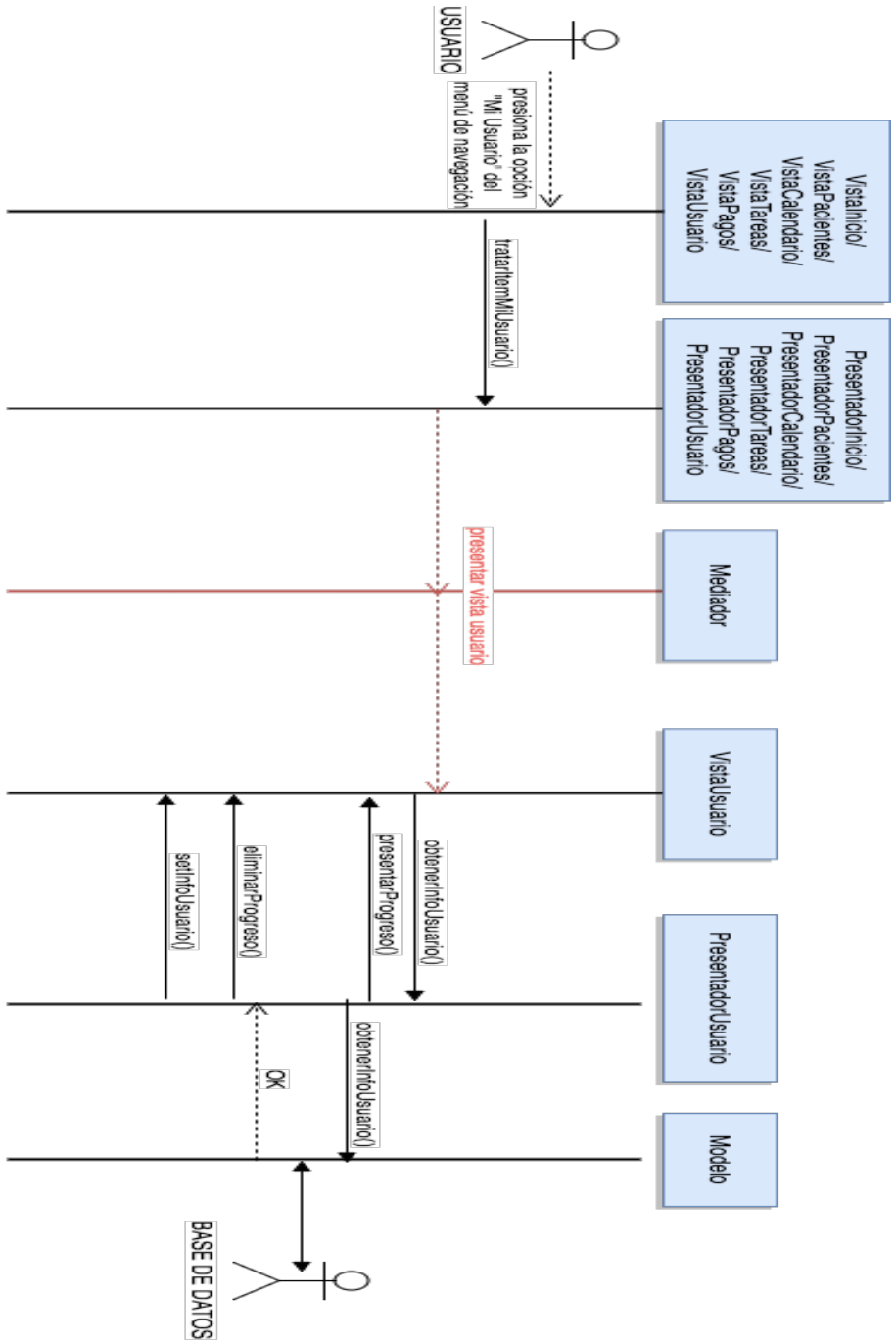

Figura 3.29: Diagrama de secuencia relacionado con la visualización de la información de la cuenta de usuario

Al fijarse en cada uno de ellos, se puede observar que son semejantes en su flujo de eventos: cada acción tiene su botón específico en el menú de la barra superior de la vista, y al presionar alguno de ellos, el presentador tratará este evento mostrando un diálogo específico para cada acción (las acciones se pueden observar en las figuras 2.27, 2.28 y 2.29, respectivamente). El usuario tomará la decisión oportuna y luego presionará sobre el botón para finalizar la interacción con el diálogo, por lo

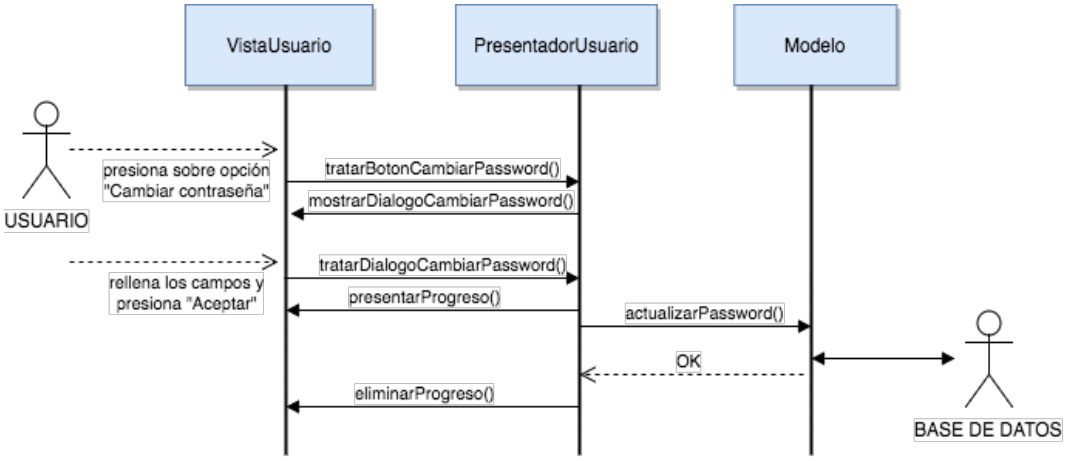

Figura 3.30: Diagrama de secuencia relacionado con la modificación de la contraseña de la cuenta de usuario

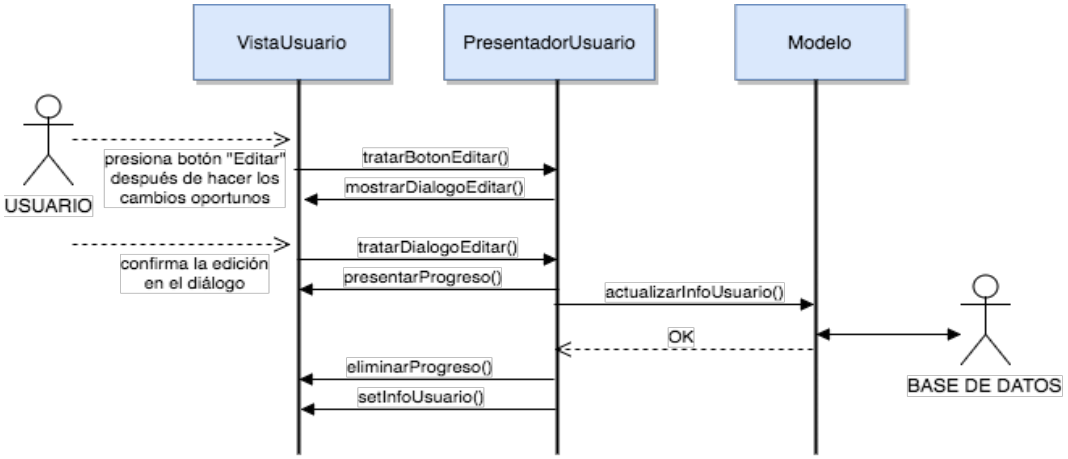

Figura 3.31: Diagrama de secuencia relacionado con la edición de la información de la cuenta de usuario

que la vista delegará en el presentador para que haga las gestiones necesarias, que supondrán una interacción con el modelo. Dependiendo de la acción, el presentador solicitará al modelo bien sea actualizar la contraseña, actualizar la información de usuario o eliminar la cuenta de usuario definitivamente. En el caso de estar actualizando la información de usuario, se volverá a la vista mostrando la información de usuario actualizada, mientras que cuando se da de baja a un usuario, se volverá a la vista de *login*, ya que esa cuenta de usuario deja de existir.

3.3 **Diseño de las clases e interfaces del modelo**

Partiendo de los diagramas de secuencia generados en el apartado anterior, la clase Modelo está compuestas por una serie de métodos, que se definen en su interfaz

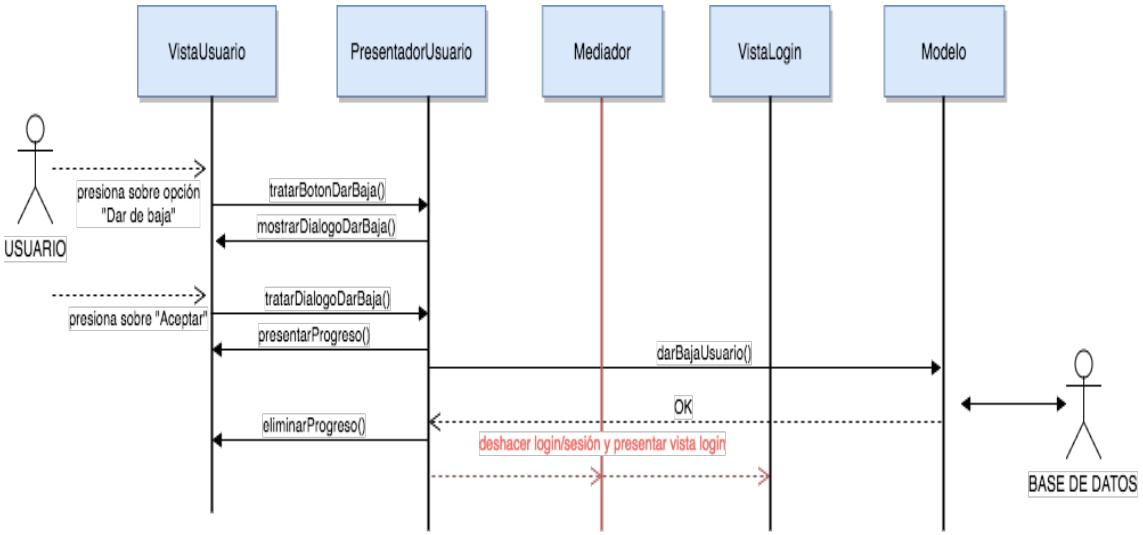

Figura 3.32: Diagrama de secuencia relacionado con la baja de la cuenta de usuario

y que se describen en la siguiente sesión. Asimismo, y debido al almacenamiento de los datos en la nube, aparecen una serie de clases para el manejo de la información, que se explicarán posteriormente.

3.3.1 Clase Modelo

Es la clase encargada de realizar consultas, así como insertar y modificar datos de la base de datos, en respuesta a peticiones del presentador correspondiente. Esta clase implementa la interfaz *IModelo*, que define todos los métodos necesarios, para que los distintos presentadores puedan interactuar con los datos de la base de datos. Los métodos definidos en la interfaz han sido "descubiertos" a partir de los diagramas de secuencia de la sección 3.2. La clase *Modelo* implementará estos métodos para darles la funcionalidad requerida y todos ellos realizarán algún proceso sobre la base de datos. Una vez finalizado este proceso, los métodos notificarán, de alguna forma, el resultado de la operación realizada, al observador que esté operando por ellos (un presentador), tal y como se indicó en la sección 3.1 (como se puede observar, todos los métodos son *void*, es decir, no devuelven nada). A continuación, se presenta una descripción detallada de estos métodos (que se encuentran definidos en la interfaz *IModelo*), así como de sus parámetros y la información que notificarán a los observadores. Los métodos son:

- **comprobarLogin(datos: Object): void**. Comprueba la validez de los datos de identificación proporcionados por el usuario al entrar en la aplicación. Estos datos son el *correo electrónico* y la *contraseña* los cuales están en el parámetro *datos*. Esta comprobación es notificada por el método.

- **comprobarCorreo(email: String): void**. Comprueba si el correo electrónico (parámetro **email**) existe para algún usuario. Este método notifica el resultado de la comprobación.

- **obtenerListaEventos(dia: Object): void**. Método que notifica los eventos de un día determinado, que se le pasa por parámetros.

- **obtenerEventosProximos(nombrePaciente: String): void**. Método que notifica los eventos próximos de un paciente determinado, cuyo nombre se hace llegar por parámetros. Se entiende por eventos próximos aquellos eventos que ocurren en el día actual (inclusive) en adelante.

- **obtenerListaPacientes(idClinica: String): void**. Método que notifica los pacientes de una clínica determinada. El identificador de la clínica se pasa por parámetros.

- **obtenerInfoPaciente(idPaciente: String): void**. Este método notifica la información de un determinado paciente en función del identificador que se le entrega por parámetros, *idPaciente*.

- **obtenerImagenesPaciente(idPaciente: String): void**. Método que, a partir de un identificador de un determinado paciente (parámetro *idPaciente*), notifica las imágenes pertenecientes a este paciente.

- **obtenerNotasPaciente(idPaciente: String): void**. Método que, a partir de un identificador de un determinado paciente, es decir, el parámetro *idPaciente*, notificará las notas relacionadas con este paciente.

- **agregarPaciente(informacion: Object): void**. Este método se encarga de añadir un nuevo paciente a una clínica determinada. La información necesaria (paciente, identificador de la clínica, etc.) estará contenida en el parámetro *informacion*. El método notifica la finalización de la operación.

- **actualizarPaciente(informacion: Object): void**. Método que actualiza, en la base de datos, la información de un determinado paciente. La información a modificar y los datos de este paciente, entre otros, se encuentran en el parámetro *datos*. El método notifica la finalización de la operación.

- **eliminarPaciente(idPaciente: String): void**. Este método se encarga de eliminar un determinado paciente, cuyo identificador es el parámetro *idPaciente*. El método se encarga de notificar el final de la operación.

- **agregarImagen(informacion: Object): void**. Este método se encarga de agregar una imagen a un paciente determinado. La información necesaria (imagen, identificador del paciente, etc.) estará contenida en el parámetro *informacion*. El método notifica la finalización de la operación.

- **obtenerImagen(idImagen: String): void**. Método que notifica una determinada imagen cuyo identificador coincida con el parámetro *idImagen*.

- **eliminarImagen(idImagen: String): void**. Este método se encarga de eliminar una determinada imagen cuyo identificador coincide con el parámetro *idImagen*. La finalización de esta operación la notifica el método.
- **agregarNota(informacion: Object): void**. Método que se encarga de añadir una nueva nota para un determinado paciente. La información necesaria (nota, identificador del paciente, etc.) estará contenida en el parámetro *informacion*. El método notifica la finalización de la operación.
- **actualizarNota(datos: Object): void**. Método que actualiza, en la base de datos, una nota determinada. La información a modificar y los datos de esta nota, entre otros, se encuentran en el parámetro *datos*. El método notifica la finalización de la operación.
- **eliminarNota(idNota: String): void**. Este método se encarga de eliminar una determinada nota, la cual se reconoce a través del identificador de la nota, que corresponde con el parámetro *idNota*. El método notifica la finalización de la operación.
- **agregarEvento(informacion: Object): void**. Este método se encarga de agregar un nuevo evento para una determinada clínica. La información necesaria (evento, identificador de la clínica, etc.) estará contenida en el parámetro *informacion*. El método notifica la finalización de la operación.
- **actualizarEvento(datos: Object): void**. Método que actualiza en la base de datos un evento determinado. La información a modificar y los datos de este evento se encuentran en el parámetro *datos*. El método notifica la finalización de la operación.
- **eliminarEvento(idEvento: String): void**. Este método se encarga de eliminar un determinado evento cuyo identificador coincide con el parámetro *idEvento*. El método notifica la finalización de la operación.
- **obtenerListaTareas(idClinica: String): void**. Método que notifica las tareas de una clínica determinada, cuyo identificador coincide con el parámetro *idClinica*.
- **agregarTarea(informacion: Object): void**. Este método se encarga de agregar una nueva tarea a una clínica determinada. La información necesaria (tarea, identificador de la clínica, etc.) estará contenida en el parámetro *informacion*. El método notifica la finalización de la operación.
- **finalizarTarea(idTarea: String): void**. Este método se encarga de finalizar (eliminar, en definitiva) una determinada tarea, cuyo identificador coincide con el parámetro *idTarea*. El método notifica la finalización de la operación.
- **obtenerListaRealizados(idClinica: String): void**. Método que notifica los pagos realizados de una clínica determinada, cuyo identificador coincide con el parámetro *idClinica*.

- **obtenerListaPendientes(idClinica: String): void**. Método que notifica los pagos pendientes de una clínica determinada, cuyo identificador coincide con el parámetro *idClinica*.

- **agregarPago(informacion: Object): void**. Este método se encarga de agregar un nuevo pago para una clínica determinada. La información necesaria (pago, identificador de la clínica, identificador del paciente, etc.) estará contenida en el parámetro *informacion*. El método notifica la finalización de la operación.

- **setPagoRealizado(idPago: String): void**. Este método se encarga de marcar como *realizado* un determinado pago pendiente, cuyo identificador coincide con el parámetro *idPago*. El método notifica la finalización de la operación.

- **obtenerInfoUsuario(idUsuario: String): void**. Este método notifica la información de un determinado usuario, es decir, un fisioterapeuta, cuyo identificador coincide con el parámetro *idUsuario*.

- **actualizarInfoUsuario(datos: Object): void**. Método que actualiza, en la base de datos, la información de un usuario determinado, es decir, un fisioterapeuta. La información a modificar y los datos de este fisioterapeuta se encuentran en el parámetro *datos*. El método notifica la finalización de la operación.

- **actualizarPassword(datos: Object): void**. Método que actualiza en la base de datos la contraseña de un determinado usuario, es decir, un fisioterapeuta. La información a modificar y los datos de este fisioterapeuta se encuentran en el parámetro *datos*. El método notifica la finalización de la operación.

- **darBajaUsuario(idFisioterapeuta: String): void**. Este método se encarga de dar de baja (eliminar, en definitiva) un determinado usuario, es decir, un fisioterapeuta, cuyo identificador coincide con el parámetro *idFisioterapeuta*.

3.3.2 Descripción del modelo de datos

Si bien los diagramas de secuencia dan una idea inicial de cómo debe ser el modelo de datos (en función de los mensajes enviados entre objetos), es necesario completar este modelo haciendo un estudio exhaustivo de la información utilizada en el aplicación (de cómo se estructurará ésta, de cómo se debe almacenar, de cómo se debe acceder, entre otras cosas). Para ello hay que realizar un diseño detallado del almacenamiento de los datos para que el acceso (escritura y lectura) sea lo más eficiente posible.

3.3.2.1. Diseño de la base de datos

En el caso de la aplicación *FisioClinicApp*, la información se almacenará en una base de datos externa al terminal (la nube) ya que deberá estar disponible para los fisioterapeutas de la clínica y deberá ser mantenida de una forma segura y el cien por cien del tiempo. Para ello se utilizará *Firebase* [12], una plataforma web y móvil en la nube de Google, que ofrece una serie de servicios como: base de datos en tiempo real, sistema de autenticación, almacenamiento, entre otros. Implementar aplicaciones usando *Firebase* es rápido y cómodo, gracias a las API intuitivas incluídas en un solo SDK (tanto para plataformas iOS como para Android).

En la base de datos *Firebase Realtime Database* [20], los datos se almacenan en forma de árboles JSON, por lo que en esta base de datos *NoSQL* [19], no existen tablas ni registros. Al agregar un dato, éstos se convierten en un nodo más en la estructura JSON ya existente.

Sin embargo, y como se conoce el tipo de información a almacenar, se representará los distintos nodos que pueden haber en la base de datos de la aplicación, en formato de tablas (por claridad), para poder explicar los campos que van a tener cada uno de estos nodos (o tablas). Así, en el caso de la aplicación que nos ocupa, existirán ocho tablas diferentes, las cuales se explicarán a continuación.

- **Tabla clinicas**. Tabla encargada de almacenar la información de las clínicas que están registradas para hacer uso de esta aplicación. Será una tabla básica para la aplicación, puesto que el uso de la aplicación y la información con la que se trabaje en ella depende de la clínica del fisioterapeuta, que ha iniciado sesión. Los campos de esta tabla son los siguientes:
  - **idClinica: String**. Identificador único de la clínica.
  - **nombre: String**. Nombre de la clínica.
  - **rutaLogo: String**. Ruta específica donde se encuentra el archivo del logo de la clínica.
- **Tabla fisioterapeutas**. Tabla que contendrá la información de los fisioterapeutas de cada clínica, los cuales son los usuarios como tal de la aplicación. Por tanto, esta tabla será utilizada para realizar las comprobaciones necesarias para iniciar sesión dentro de la aplicación, además de tener otro tipo de información, como su correo electrónico, que puede ser modificado en la misma aplicación. Estos son los campos de la tabla:
  - **idFisioterapeuta: String**. Identificador único de cada fisioterapeuta.
  - **nombre: String**. Nombre del fisioterapeuta.
  - **password: String**. Contraseña de acceso para la cuenta de usuario del fisioterapeuta.

- **correo: String**. Dirección de correo electrónico del fisioterapeuta, que servirá como identificador para iniciar sesión en la aplicación.
- **telefono: String**. Número de teléfono del fisioterapeuta.
- **idClinica: String**. Identificador para relacionar el fisioterapeuta con la clínica a la que pertenece.

Sin embargo, la plataforma *Firebase*, además de su servicio de base de datos en la nube (*Firebase Realtime Database*), permite hacer uso de un potente sistema de autenticación, llamado *Firebase Authentication* [21]. Este servicio se va a encargar de la administración de los usuarios de forma segura, es decir, tareas como el inicio de sesión, cambios en la información utilizada para el acceso (contraseña y correo electrónico), revocar acceso a usuarios, entre otros. Son numerosas las virtudes que proporciona este sistema, por lo que, la aplicación diseñada va a delegar el tratamiento de la información sensible, en *Firebase*.

Así, cada usuario registrado en la aplicación, que corresponde con cada fisioterapeuta, tendrá su información de acceso esencial (correo electrónico y contraseña) e información importante, pero no tan esencial, como: nombre, teléfono e identificador de la clínica en la que trabaja. Debido a esto, a la forma segura de trabajar con *Firebase* y a la mejora del rendimiento en ciertos usos de la aplicación, se plantea una modificación de la tabla *fisioterapeutas*, de forma que almacene los siguientes campos:

- **idFisioterapeuta: String**. Identificador único de cada fisioterapeuta.
- **nombre: String**. Nombre del fisioterapeuta. Este campo será encriptado.
- **idClinica: String**. Identificador para relacionar el fisioterapeuta con la clínica a la que pertenece.

Por otro lado, la información del fisioterapeuta que falta por almacenar (y que es la más sensible), se hará de forma segura usando *Firebase Authentication*. Así, este servicio almacena la información de la siguiente forma:

- **Email:** Correo electrónico del usuario. Es un elemento clave para el inicio de sesión del usuario en la aplicación.
- **Password:** Contraseña del usuario, por tanto importante cuando se desea iniciar sesión por parte de un determinado usuario.
- **Display Name:** Propiedad complementaria a la información de acceso del usuario. En este campo se almacenará, de manera concatenada, el nombre del fisioterapeuta, su teléfono y el identificador de su clínica. Y se hará utilizando el siguiente formato (en una cadena de caracteres): "nombre#telefono#idClinica".

- **Tabla pacientes**. Tabla que almacenará la información de los pacientes de cada clínica. Esta tabla será esencial, sobre todo, para obtener la información de los pacientes de la clínica del fisioterapeuta que ha iniciado sesión, aunque también se harán accesos para escribir en ella, como pueda ser: añadir un

nuevo paciente o eliminar alguno ya existente. Los campos de esta tabla son los siguientes:

- **idPaciente: String**. Identificador único de cada paciente.
- **nombre: String**. Nombre del paciente. Este campo será encriptado.
- **correo: String**. Dirección de correo electrónico del paciente. Este campo será encriptado.
- **telefono: String**. Número de teléfono del paciente. Este campo será encriptado.
- **historial: String**. Historial clínico del paciente. Este campo será encriptado.
- **idClinica: String**. Identificador para relacionar el paciente con la clínica a la que pertenece.

- **Tabla imagenes**. Tabla que contendrá un registro de las imágenes que pertenecen a cada paciente. Se accederá a ella principalmente para obtener las imágenes y mostrarlas en el listado de cada paciente, pero también habrán ciertos accesos para el manejo de las mismas, como puede ser: añadir alguna nueva o eliminarla. Los campos de esta tabla son los siguientes:

  - **idImagen: String**. Identificador único de cada imagen.
  - **ruta: String**. Contiene la ruta específica en la que se encuentra el archivo de la imagen como tal.
  - **descripcion: String**. Descripción de la imagen. Este campo será encriptado.
  - **idPaciente: String**. Identificador para relacionar la imagen con el paciente al que pertenece dicha imagen.

- **Tabla notas**. Tabla que almacenará todas las notas de los pacientes de cada clínica. Sobre esta tabla se realizarán las consultas oportunas para poder listar las notas de cada paciente, pero también para realizar modificaciones en ella, como pueda ser: añadir una nueva o eliminar alguna ya existente. Los campos de esta tabla son los siguientes:

  - **idNota: String**. Identificador único de cada nota.
  - **titulo: String**. Título de la nota. Este campo será encriptado.
  - **cuerpo: String**. Cuerpo de la nota, es decir, la redacción de la nota como tal. Este campo será encriptado.
  - **idPaciente: String**. Identificador para relacionar la nota con el paciente al que pertenece dicha nota.

- **Tabla eventos**. Tabla que contendrá todos los eventos que se registran para las clínicas de fisioterapia, que hagan uso de la aplicación, siendo la gestión de este tipo de información una de las tareas principales, ya que es un pilar básico para la organización de la clínica. Los campos de esta tabla son los siguientes:

- **idEvento: String**. Identificador único del evento.
- **fecha: Object**. Fecha en la que ocurre el evento registrado.
- **horaComienzo: Object**. Hora de comienzo del evento registrado.
- **horaFinalizacion: Object**. Hora de finalización del evento registrado.
- **tipo: String**. Tipo de evento. Este campo será encriptado.
- **descripcion: String**. Descripción del evento. Este campo será encriptado.
- **nombrePaciente: String**. Nombre del paciente para el que se registra el evento. Este campo será encriptado.
- **nombreFisioterapeuta: String**. Nombre del fisioterapeuta relacionado con el evento. Este campo será encriptado.
- **idClinica: String**. Identificador de la clínica relacionada con el evento.

- **Tabla tareas**. Tabla que almacenará la información de todas las tareas existentes para una clínica de fisioterapia en concreto. Se accederá a esta tabla para realizar consultas y obtener dichas tareas, pero también para añadir nuevas y eliminar alguna ya existente cuando ésta haya sido finalizada. Los campos de esta tabla son los siguientes:

  - **idTarea: String**. Identificador único de la tarea.
  - **titulo: String**. Título de la tarea. Este campo será encriptado.
  - **idClinica: String**. Identificador de la clínica a la que pertenece la tarea en cuestión.

- **Tabla pagos**. Tabla que almacenará un registro constante de pagos que se van realizando cada día en la clínica de fisioterapia, y de pagos pendientes. Además de acceder a esta tabla para obtener los pagos para poder presentarlos, también se accederá a ella para añadir nuevos pagos, e incluso modificar algunos pagos como realizados. Los campos de esta tabla son los siguientes:

  - **idPago: String**. Identificador único del pago.
  - **montante: double**. Montante del pago en cuestión.
  - **fecha: Object**. Fecha a la que corresponde el pago.
  - **realizado: boolean**. Indica si el pago ha sido realizado (*true*) o está pendiente de realizar (*false*).
  - **nombrePaciente: String**. Nombre del paciente con el que se relaciona el pago. Este campo será encriptado.
  - **idClinica: String**. Identificador de la clínica con la que se relaciona el pago.

3.3.2.2. Diseño de las nuevas clases del modelo

En esta sección se presentan cada una de las clases relacionadas con las tablas de la base de datos propuestas en el apartado anterior, así como los métodos que incluyen cada una de ellas, para el manejo de los campos de las tablas. Así, la tabla x tendrá relacionadas dos clases: una clase simple que representa a un objeto almacenado en la tabla (se llamará clase X o similar) y una clase adaptador, que manejará los campos de esa tabla (se llamada *BDAdaptadorX* o similar). Las nuevas clases son:

- **Clase Clinica**. Es la clase que define a la entidad *clinicas*. Esta clase incluye el constructor y los métodos *getters* y *setters* relacionados con los atributos de la misma. Todos ellos se muestran a continuación:
  - **Clinica(idClinica: String, nombre: String, rutaLogo: String)**. Constructor de la clase *Clinica*.
  - **getIdClinica(): String**. Método *getter* del atributo *idClinica*.
  - **getNombre(): String**. Método *getter* del atributo *nombre*.
  - **getRutaLogo(): String**. Método *getter* del atributo *rutaLogo*.
  - **setIdClinica(idClinica: String): void**. Método *setter* del atributo *idClinica*.
  - **setNombre(nombre: String): void**. Método *setter* del atributo *nombre*.
  - **setRutaLogo(rutaLogo: String): void**. Método *setter* del atributo *rutaLogo*.
- **Clase BDAdaptadorClinica**. Es la clase relacionada con la tabla *clinicas* y funciona como *clase puente* entre el modelo y la base de datos. No está previsto inicialmente que tenga algún método, porque la aplicación desarrollada se centra en la parte del trabajo del fisioterapeuta (usuario).
- **Clase Fisioterapeuta**. Es la clase que define a la entidad *fisioterapeutas*. Esta clase no estará únicamente relacionada a la tabla *fisioterapeutas* explicada previamente, sino que servirá también para poder manejar los usuarios que provienen del servicio *Firebase Authentication*, y su información. A ojos de la aplicación, los usuarios son los fisioterapeutas de cada clínica, por lo que la información que proviene de los usuarios debe ser tratada para el funcionamiento correcto de la aplicación, y de ahí la estructura que esta clase tiene, y que se presenta seguidamente.

  Esta clase incluye el constructor y los métodos *getters* y *setters* relacionados con los atributos de la misma, excepto con la contraseña del usuario, que sólo se guarda de forma segura en *Firebase*. Todos ellos se muestran a continuación:

- **Fisioterapeuta(idFisioterapeuta: String, nombre: String, correo: String, telefono: String, idClinica: String)**. Constructor de la clase *Fisioterapeuta*.
- **getIdFisioterapeuta(): String**. Método *getter* del atributo *idFisioterapeuta*.
- **getNombre(): String**. Método *getter* del atributo *nombre*.
- **getCorreo(): String**. Método *getter* del atributo *correo*.
- **getIdClinica(): String**. Método *getter* del atributo *idClinica*.
- **setIdFisioterapeuta(idFisioterapeuta: String): void**. Método *setter* del atributo *idFisioterapeuta*.
- **setNombre(nombre: String): void**. Método *setter* del atributo *nombre*.
- **setCorreo(correo: String): void**. Método *setter* del atributo *correo*.
- **setIdClinica(idClinica: String): void**. Método *setter* del atributo *idClinica*.

- **Clase BDAdaptadorFisioterapeuta**. Es la clase relacionada con la tabla *fisioterapeutas* y funciona como *clase puente* entre el modelo y la base de datos. Inicialmente está previsto que esta clase no tenga ningún método, teniendo en cuenta que todo lo referente a la gestión de los usuarios de la aplicación (los fisioterapeutas, en definitiva) se encarga la clase *Modelo*, ya que es la que se comunica con el servicio *Firebase Auth*.

- **Clase Paciente**. Es la clase que define a la entidad *pacientes*. Esta clase incluye el constructor y los métodos *getters* y *setters* relacionados con los atributos de la misma. Todos ellos se muestran a continuación:

  - **Paciente(idPaciente: String, nombre: String, correo: String, telefono: String, historial: String, idClinica: String)**. Constructor de la clase *Paciente*.
  - **getIdPaciente(): String**. Método *getter* del atributo *idPaciente*.
  - **getNombre(): String**. Método *getter* del atributo *nombre*.
  - **getCorreo(): String**. Método *getter* del atributo *correo*.
  - **getTelefono(): String**. Método *getter* del atributo *telefono*.
  - **getHistorial(): String**. Método *getter* del atributo *historial*.
  - **getIdClinica(): String**. Método *getter* del atributo *idClinica*.
  - **setIdPaciente(idPaciente: String): void**. Método *setter* del atributo *idPaciente*.
  - **setNombre(nombre: String): void**. Método *setter* del atributo *nombre*.
  - **setCorreo(correo: String): void**. Método *setter* del atributo *correo*.

  - **setTelefono(telefono: String): void**. Método *setter* del atributo *telefono*.
  - **setHistorial(historial: String): void**. Método *setter* del atributo *historial*.
  - **setIdClinica(idClinica: String): void**. Método *setter* del atributo *idClinica*.

- **Clase BDAdaptadorPaciente**. Es la clase relacionada con la tabla *pacientes* y funciona como *clase puente* entre el modelo y la base de datos. Incluye los siguientes métodos:

  - **obtenerListaPacientes(idClinica: String): void**. Busca, en la base de datos, aquellos pacientes que pertenecen a una clínica determinada, cuyo identificador coincide con el parámetro *idClinica*. El método notifica la lista con los pacientes encontrados.
  - **obtenerPaciente(idPaciente: String): void**. Realiza una consulta en la tabla *pacientes* con el objetivo de encontrar el paciente que concuerde con el parámetro *idPaciente*. El método notifica la información del paciente encontrado.
  - **agregarPaciente(informacion: Object): void**. Añade un nuevo paciente a una clínica determinada que se indica a través del parámetro *informacion*. Ese parámetro, entre otros datos, almacenará: el paciente a añadir, el identificador de la clínica a la que se va a añadir, etc. El método notifica el resultado de la operación.
  - **actualizarPaciente(datos: Object): void**. Con este método se modifica la información de un determinado paciente. La información a modificar y los datos del paciente se encuentran en el parámetro *datos*. El método notifica el resultado de la actualización.
  - **eliminarPaciente(idPaciente: String): void**. Este método se encarga de eliminar un determinado paciente, cuyo identificador coincide con el parámetro *idPaciente*. El método notifica el resultado final de la operación.

- **Clase Imagen**. Es la clase que define a la entidad *imagenes*. Esta clase incluye el constructor y los métodos *getters* relacionados con los atributos de la misma. Todos ellos se muestran a continuación:

  - **Imagen(idImagen: String, ruta: String, descripcion: String, idPaciente: String)**. Constructor de la clase *Imagen*.
  - **getIdImagen(): String**. Método *getter* del atributo *idImagen*.
  - **getRuta(): String**. Método *getter* del atributo *ruta*.
  - **getDescripcion(): String**. Método *getter* del atributo *descripcion*.
  - **getIdPaciente(): String**. Método *getter* del atributo *idPaciente*.
  - **setIdImagen(idImagen: String): void**. Método *setter* del atributo *idImagen*.

- **setRuta(ruta: String): void**. Método *setter* del atributo *ruta*.
- **setDescripcion(descripcion: String): void**. Método *setter* del atributo *descripcion*.
- **setIdPaciente(idPaciente: String): void**. Método *setter* del atributo *idPaciente*.

- **Clase BDAdaptadorImagen**. Es la clase relacionada con la tabla *imagenes* y funciona como *clase puente* entre el modelo y la base de datos. Incluye los siguientes métodos:

  - **obtenerImagenesPaciente(idPaciente: String): void**. Consulta en la tabla *imagenes* cuáles son las imágenes que pertenecen al paciente cuyo identificador coincide con el parámetro *idPaciente*. El método notifica la información de las imágenes del paciente encontrado.
  - **obtenerImagen(idImagen: String): void**. Consulta en la tabla *imagenes*, qué imagen contiene el identificador que se ha pasado por parámetros (*idImagen*) para notificarla al observador.
  - **agregarImagen(informacion: Object): void**. Añade una nueva imagen al paciente que se indique a través del parámetro *informacion*. Ese parámetro, entre otros datos, almacenará: la imagen a añadir, el identificador del paciente al cual se le va a añadir, etc. El método notifica si se pudo o no agregar.
  - **eliminarImagen(idImagen: String): void**. Este método se encarga de eliminar una determinada imagen de la base de datos. La imagen a eliminar debe tener un identificador que coincida con el parámetro *idImagen*. El método notifica si se pudo o no eliminar.

- **Clase Nota**. Es la clase que define a la entidad *notas*. Esta clase incluye el constructor y los métodos *getters* relacionados con los atributos de la misma. Todos ellos se muestran a continuación:

  - **Nota(idNota: String, titulo: String, cuerpo: String, idPaciente: String)**. Constructor de la clase *Nota*.
  - **getIdNota(): String**. Método *getter* del atributo *idNota*.
  - **getTitulo(): String**. Método *getter* del atributo *titulo*.
  - **getCuerpo(): String**. Método *getter* del atributo *cuerpo*.
  - **getIdPaciente(): String**. Método *getter* del atributo *idPaciente*.
  - **setIdNota(idNota: String): void**. Método *setter* del atributo *idNota*.
  - **setTitulo(titulo: String): void**. Método *setter* del atributo *titulo*.
  - **setCuerpo(cuerpo: String): void**. Método *setter* del atributo *cuerpo*.
  - **setIdPaciente(idPaciente: String): void**. Método *setter* del atributo *idPaciente*.

- **Clase BDAdaptadorNota**. Es la clase relacionada con la tabla *notas* y funciona como *clase puente* entre el modelo y la base de datos. Incluye los siguientes métodos:
  - **obtenerNotasPaciente(idPaciente: String): void**. Consulta en la tabla *notas* cuáles son las notas que pertenecen al paciente que tiene el identificador que se pasa por parámetros (*idPaciente*). El método notifica la información de las notas del paciente encontrado.
  - **agregarNota(informacion: Object): void**. Este método se encarga de añadir una nueva nota al paciente que se indique a través del parámetro *informacion*. Este parámetro, entre otros datos, almacenará: la nota a añadir, el identificador del paciente al cual se va a añadir, etc. El método notifica si se pudo o no agregar.
  - **actualizarNota(datos: Object): void**. Con este método se modifica los datos de un determinado paciente de la base de datos. La información a modificar y los datos de este paciente se encuentran en el parámetro *datos*. El método notifica si se pudo o no actualizar.
  - **eliminarNota(idNota: String): void**. Este método se encarga de eliminar una determinada nota de la base de datos, cuyo identificador coincide con el parámetro *idNota*. El método notifica si se pudo o no eliminar.
- **Clase Evento**. Es la clase que define a la entidad *eventos*. Esta clase incluye el constructor y los métodos *getters* relacionados con los atributos de la misma. Todos ellos se muestran a continuación:
  - **Evento(idEvento: String, fecha: Object, horaComienzo: Object, horaFinalizacion: Object, tipo: String, descripcion: String, nombrePaciente: String, nombreFisioterapeuta: String, idClinica: String)**. Constructor de la clase *Evento*.
  - **getIdEvento(): String**. Método *getter* del atributo *idEvento*.
  - **getFecha(): Object**. Método *getter* del atributo *fecha*.
  - **getHoraComienzo(): Object**. Método *getter* del atributo *horaComienzo*.
  - **getHoraFinalizacion(): Object**. Método *getter* del atributo *horaFinalizacion*.
  - **getTipo(): String**. Método *getter* del atributo *tipo*.
  - **getDescripcion(): String**. Método *getter* del atributo *descripcion*.
  - **getNombrePaciente(): String**. Método *getter* del atributo *nombrePaciente*.
  - **getNombreFisioterapeuta(): String**. Método *getter* del atributo *nombreFisioterapeuta*.
  - **getIdClinica(): String**. Método *getter* del atributo *idClinica*.
  - **setIdEvento(idEvento: String): void**. Método *setter* del atributo *idEvento*.

- **setFecha(fecha: Object): void**. Método *setter* del atributo *fecha*.
- **setHoraComienzo(horaComienzo: Object): void**. Método *setter* del atributo *horaComienzo*.
- **setHoraFinalizacion(horaFinalizacion: Object): void**. Método *setter* del atributo *horaFinalizacion*.
- **setTipo(tipo: String): void**. Método *setter* del atributo *tipo*.
- **setDescripcion(descripcion: String): void**. Método *setter* del atributo *descripcion*.
- **setNombrePaciente(nombrePaciente: String): void**. Método *setter* del atributo *nombrePaciente*.
- **setNombreFisioterapeuta(nombreFisioterapeuta: String): void**. Método *setter* del atributo *nombreFisioterapeuta*.
- **setIdClinica(idClinica: String): void**. Método *setter* del atributo *idClinica*.

- **Clase BDAdaptadorEvento**. Es la clase relacionada con la tabla *eventos* y funciona como *clase puente* entre el modelo y la base de datos. Incluye los siguientes métodos:

  - **obtenerListaEventos(datos: Object): void**. Busca, en la base de datos, todos los eventos de una clínica determinada para un día específico. Tanto el identificador de la clínica como el día específico se hacen llegar a través del parámetro *datos*. El método notifica la lista con los eventos encontrados.
  - **obtenerEventosProximos(nombrePaciente: String): void**. Notifica la lista de eventos próximos que tiene un determinado paciente, cuyo nombre se hace llegar por el parámetro *nombrePaciente*. Se entiende por eventos próximos los que ocurren en el día actual y en adelante.
  - **agregarEvento(informacion: Object): void**. Añade un nuevo evento a una clínica determinada. El identificador de esta clínica se incluye en el parámetro *informacion*, que además lleva más información, como el evento a añadir, entre otros. El método notifica si se pudo o no agregar.
  - **actualizarEvento(datos: Object): void**. Con este método se modifican los datos de un evento de la base de datos. La información a modificar y los datos del evento se encuentran en el parámetro *datos*. El método notifica el resultado de la operación.
  - **eliminarEvento(idEvento: String): void**. Este método se encarga de eliminar un determinado evento de la base de datos, cuyo identificador coincide con el parámetro *idEvento*. El método notifica si la eliminación se realizó correctamente o hubo algún problema.

- **Clase Tarea**. Es la clase que define a la entidad *tareas*. Esta clase incluye el constructor y los métodos *getters* relacionados con los atributos de la misma. Todos ellos se muestran a continuación:

- **Tarea(idTarea: String, titulo: String, idClinica: String)**. Constructor de la clase *Tarea*.
- **getIdTarea(): String**. Método *getter* del atributo *idTarea*.
- **getTitulo(): String**. Método *getter* del atributo *titulo*.
- **getIdClinica(): String**. Método *getter* del atributo *idClinica*.
- **setIdTarea(idTarea: String): void**. Método *setter* del atributo *idTarea*.
- **setTitulo(titulo: String): void**. Método *setter* del atributo *titulo*.
- **setIdClinica(idClinica: String): void**. Método *setter* del atributo *idClinica*.

- **Clase BDAdaptadorTarea**. Es la clase relacionada con la tabla *tareas* y funciona como *clase puente* entre el modelo y la base de datos. Incluye los siguientes métodos:
  - **obtenerListaTareas(idClinica: String): void**. Busca, en la base de datos, todas las tareas que pertenecen a una determinada clínica, cuyo identificador coincide con el parámetro *idClinica*. El método notifica la lista con las tareas encontrados.
  - **agregarTarea(informacion: Object): void**. Añade una nueva tarea a una clínica determinada, cuyo identificador se encuentra en el parámetro *informacion*. Además de este identificador, este parámetro también almacena otra información como la tarea a añadir, entre otros. El método notifica si se pudo o no agregar.
  - **eliminarTarea(idTarea: String): void**. Este método se encarga de eliminar una determinada tarea de la base de datos, cuyo identificador coincide con el parámetro *idTarea*. El método notifica si se pudo o no eliminar.

- **Clase Pago**. Es la clase que define a la entidad *pagos*. Esta clase incluye el constructor y los métodos *getters* relacionados con los atributos de la misma. Todos ellos se muestran a continuación:
  - **Pago(idPago: String, montante: double, realizado: boolean, fecha: Object, nombrePaciente: String, idClinica: String)**. Constructor de la clase *Pago*.
  - **getIdPago(): String**. Método *getter* del atributo *idPago*.
  - **getMontante(): double**. Método *getter* del atributo *montante*.
  - **getRealizado(): boolean**. Método *getter* del atributo *realizado*.
  - **getFecha(): Object**. Método *getter* del atributo *fecha*.
  - **getNombrePaciente(): String**. Método *getter* del atributo *nombrePaciente*.
  - **getIdClinica(): String**. Método *getter* del atributo *idClinica*.

- **setIdPago(idPago: String): void**. Método *setter* del atributo *idPago*.
- **setMontante(montante: double): void**. Método *setter* del atributo *montante*.
- **setRealizado(realizado: boolean): void**. Método *setter* del atributo *realizado*.
- **setFecha(fecha: Object): void**. Método *setter* del atributo *fecha*.
- **setNombrePaciente(nombrePaciente: String): void**. Método *setter* del atributo *nombrePaciente*.
- **setIdClinica(idClinica: String): void**. Método *setter* del atributo *idClinica*.

- **Clase BDAdaptadorPago**. Es la clase relacionada con la tabla *pagos* y funciona como *clase puente* entre el modelo y la base de datos. Incluye los siguientes métodos:

  - **obtenerListaPagos(datos: Object): void**. Busca, en la base de datos, los pagos que pertenecen a una determinada clínica, cuyo identificador se encuentra incluido en el parámetro *datos*. Para obtener esta lista, además, también se incluye otro dato en este parámetro, y va asociado al atributo *realizado* de un pago, puesto que en función de qué valor tome este parámetro, se filtrará la lista por pagos realizados o por pagos pendientes. El método notifica la lista con los pagos encontrados.
  - **agregarPago(informacion: Object): void**. Añade un nuevo pago a una clínica determinada, cuyo identificador se incluye en el parámetro *informacion*. Este parámetro, entre otros datos, almacenará: el pago, el identificador de la clínica a la que se va a añadir, etc. El método notifica si se pudo o no agregar.
  - **setPagoRealizado(idPago: String): void**. Este método se encarga de actualizar el atributo *realizado* de un determinado pago, cuyo identificador coincide con el parámetro *idPago*. El método notifica si se pudo o no realizar la acción.

3.4 Diseño de las clases e interfaces de la vista

Partiendo de los diagramas de secuencia generados en el apartado 3.2, las interfaces de la vista están compuestas por una serie de métodos que se pasarán a describir a continuación.

3.4.1 Clase VistaLogin

Es la vista donde los usuarios pueden iniciar sesión en la aplicación y, por tanto, comenzar a hacer uso de la funcionalidad de la misma. Esta clase implementa la interfaz *IVistaLogin*, que está compuesta de los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.
- **mostrarMensajeAclaratorio(): void**. Muestra un mensaje al usuario para explicarle que la aplicación tiene un sistema de registro cerrado, por lo que tendrá que pedir permiso al encargado de su clínica que le genere una cuenta de usuario.

3.4.2 Clase VistaRecordarPassword

Es la vista donde los usuarios pueden solicitar que se les recuerde su contraseña. Esta clase implementa la interfaz *IVistaRecordarPassword*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.

3.4.3 Clase VistaInicio

Es la vista inicial de la aplicación, a la que se accede, además, a través del menú de navegación (ítem "Inicio"). El principal contenido, de esta pantalla, es la lista de eventos que tiene la clínica del fisioterapeuta que ha iniciado sesión (en ese día específico). Esta clase implementa la interfaz *IVistaInicio*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.
- **setListaEventos(listaEventos: Evento[]): void**. Muestra en pantalla la lista de eventos que llegan por parámetros.

3.4.4 Clase VistaPacientes

Vista a la que se accede a través del menú de navegación (ítem "Pacientes") y en ella se muestra la lista de pacientes que tiene la clínica del fisioterapeuta que ha iniciado sesión. Esta clase implementa la interfaz *IVistaPacientes*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.
- **setListaPacientes(listaPacientes: Paciente[]): void**. Muestra en pantalla la lista de pacientes que llega por parámetros.

3.4.5 Clase VistaFichaPaciente

Vista que muestra la ficha de un paciente determinado, que ha sido seleccionado previamente. En ella, se aprecian tres partes bien diferenciadas, gracias al uso de solapas: información del paciente, imágenes del paciente y, por último, sus notas. Esta clase implementa la interfaz *IVistaFichaPaciente*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.
- **setInfoPaciente(paciente: Paciente): void**. Se encarga de mostrar en pantalla la información del paciente que se recibe por parámetros.
- **setEventosProximosPaciente(eventosProximos: Evento[]): void**. Muestra en pantalla la lista de eventos próximos del paciente, cuya lista se recibe por parámetros.
- **setImagenesPaciente(imagenes: Imagen[]): void**. Se encarga de mostrar en pantalla las imágenes del paciente que se reciben por parámetros.
- **setNotasPaciente(notas: Nota[]): void**. Se encarga de mostrar en pantalla las notas del paciente que se reciben por parámetros.
- **mostrarDialogoEliminar(): void**. Muestra un diálogo al usuario, requiriéndole la confirmación de si quiere o no continuar con eliminar el paciente cuya ficha está visualizando.
- **mostrarDialogoEliminarNota(): void**. Muestra un diálogo al usuario, requiriéndole la confirmación de si quiere o no continuar con la eliminación de la nota que ha seleccionado.

3.4.6 Clase VistaAgregarEditarPaciente

Vista que permite agregar o editar un paciente, en función desde dónde se accede a esta vista. Esta clase implementa la interfaz *IVistaAgregarEditarPaciente*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.

3.4.7 Clase VistaAgregarImagen

Vista que permite agregar una nueva imagen para el paciente cuya ficha ha sido previamente desplegada. Esta clase implementa la interfaz *IVistaAgregarImagen*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.

3.4.8 Clase VistaDetalleImagen

Vista que permite ver al detalle una imagen, que ha sido previamente seleccionada del archivo de imágenes del paciente. Esta clase implementa la interfaz *IVistaDetalleImagen*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.
- **setImagen(imagen: Imagen): void**. Muestra en pantalla la imagen que llega por parámetros, además de su correspondiente descripción.
- **mostrarDialogoEliminar(): void**. Muestra un diálogo al usuario, requiriéndole la confirmación de si quiere o no continuar con la eliminación de la imagen.

3.4.9 Clase VistaAgregarEditarNota

Vista que permite agregar o editar una nota, en función desde dónde se accede a esta vista. Esta clase implementa la interfaz *IVistaAgregarEditarNota*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.

3.4.10 Clase VistaCalendario

Vista que se muestra a través del menú de navegación (ítem "Calendario") y en ella se presenta un calendario, con el que el usuario podrá interactuar, para poder seleccionar un día específico. Esta clase implementa la interfaz *IVistaCalendario*, que inicialmente no tiene métodos.

3.4.11 Clase VistaEventos

Vista que permite visualizar la lista de eventos, que existen para un determinado día, el cual ha sido seleccionado previamente en la vista del calendario. Esta clase implementa la interfaz *IVistaEventos*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.
- **setListaEventos(listaEventos: Evento[]): void**. Se encarga de mostrar en pantalla la lista de eventos que llega por parámetros.
- **mostrarDialogoEliminar(): void**. Muestra un diálogo al usuario, requiriéndole la confirmación de si quiere o no continuar con la eliminación del evento.

3.4.12 Clase VistaAgregarEditarEvento

Vista que permite agregar o editar un evento, en función desde dónde se accede a esta vista. Esta clase implementa la interfaz *IVistaAgregarEditarEvento*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.

3.4.13 Clase VistaTareas

Vista a la que se accede a través del menú de navegación (ítem "Tareas") y en ella se muestra una lista de tareas para hacer, las cuales pertenecen a la clínica del fisioterapeuta que ha iniciado sesión. Esta clase implementa la interfaz *IVistaTareas*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.
- **setListaTareas(listaTareas: Tarea[]): void**. Muestra en pantalla la lista de tareas que llega por parámetros.
- **mostrarDialogoAgregar(): void**. Muestra un diálogo con un formulario, que tendrá que rellenar el usuario para poder agregar una tarea.
- **mostrarDialogoFinalizar(): void**. Muestra un diálogo al usuario, requiriéndole la confirmación de si quiere o no continuar con la finalización de la tarea.

3.4.14 Clase VistaPagos

Vista a la que se accede a través del menú de navegación (ítem "Pagos") y en ella se muestra una lista de pagos realizados o pendientes de realizar. Esta clase implementa la interfaz *IVistaPagos*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.
- **setListaPagos(listaPagos: Pago[]): void**. Muestra en pantalla la lista de pagos que llega por parámetros.
- **mostrarDialogoAgregar(): void**. Muestra un diálogo con un formulario, que tendrá que rellenar el usuario para poder agregar un pago.
- **mostrarDialogoRealizado(): void**. Muestra un diálogo al usuario, requiriéndole la confirmación de si está seguro o no de dar el pago seleccionado por realizado.

3.4.15 Clase VistaUsuario

Vista a la que se accede a través del menú de navegación (ítem "Mi usuario") y en ella se presenta la información del usuario (fisioterapeuta) que está utilizando la aplicación. Esta clase implementa la interfaz *IVistaUsuario*, que define los siguientes métodos:

- **presentarProgreso(): void**. Muestra una barra de progreso para indicar al usuario que se está realizando alguna acción en *background*.
- **eliminarProgreso(): void**. Elimina la barra de progreso para indicar al usuario que se terminó de realizar la acción en *background*.
- **setInfoUsuario(usuario: Fisioterapeuta): void**. Muestra en pantalla la información del usuario que le llega por parámetros.
- **mostrarDialogoEditar(): void**. Muestra un diálogo al usuario, requiriéndole la confirmación de si desea continuar con la edición o, por el contrario, desea cancelarla.
- **mostrarDialogoCambiarPassword(): void**. Muestra un diálogo con un formulario, que el usuario tendrá que rellenar para poder cambiar la contraseña de su cuenta.
- **mostrarDialogoDarBaja(): void**. Muestra un diálogo al usuario, requiriéndole la confirmación de si desea continuar con el proceso de dar de baja a su cuenta.

3.5 Diseño de las clases e interfaces del presentador

Partiendo de los diagramas de secuencia generados en el apartado 3.2, las interfaces del presentador están compuestas por una serie de métodos que se pasarán a describir a continuación.

3.5.1 Clase PresentadorLogin

Presentador de la vista *VistaLogin* que implementa la interfaz *IPresentadorLogin*, que define los siguientes métodos:

- **tratarBotonEntrar(correoElectronico: String, password: String): void**. Encargado de comunicarse con el modelo, para realizar la comprobación oportuna de la información insertada por el usuario, para realizar el inicio de sesión en la aplicación.
- **tratarRecordarPassword(): void**. Notifica al presentador que el texto "¿Te olvidaste de la contraseña?" de la vista ha sido presionado. Este método hará que el presentador solicite la navegación a la vista *VistaRecordarPassword*.
- **tratarSinCuenta(): void**. Notifica al presentador que el texto "¿No tienes cuenta?" de la vista ha sido presionado. Este método se encargará de solicitar a la vista que muestre un determinado mensaje aclaratorio definido en la misma.

3.5.2 Clase PresentadorRecordarPassword

Presentador de la vista *VistaRecordarPassword* que implementa la interfaz *IPresentadorRecordarPassword*, que define los siguientes métodos:

- **tratarBotonRecordar(correoElectronico: String): void**. Notifica al presentador que el botón "Recordar" de la vista ha sido presionado. Este método se encargará de comunicarse con el modelo para pasarle el correo electrónico que ha insertado el usuario en la vista, con el objetivo de que haga las comprobaciones oportunas.

3.5.3 Clase PresentadorInicio

Presentador de la vista *VistaInicio* que implementa la interfaz *IPresentadorInicio*, que define los siguientes métodos:

- **tratarItemInicio(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaInicio*.
- **tratarItemPacientes(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPacientes*.
- **tratarItemCalendario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaCalendario*.
- **tratarItemTareas(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaTareas*.
- **tratarItemPagos(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPagos*.
- **tratarItemMiUsuario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaUsuario*.
- **obtenerListaEventos(): void**. Método que se comunica con el modelo para obtener la lista de eventos, que la clínica del fisioterapeuta que ha iniciado sesión, tiene el día actual.

3.5.4 Clase PresentadorPacientes

Presentador de la vista *VistaPacientes* que implementa la interfaz *IPresentadorPacientes*, que define los siguientes métodos:

- **tratarItemInicio(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaInicio*.
- **tratarItemPacientes(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPacientes*.
- **tratarItemCalendario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaCalendario*.
- **tratarItemTareas(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaTareas*.
- **tratarItemPagos(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPagos*.

- **tratarItemMiUsuario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaUsuario*.
- **obtenerListaPacientes(idClinica: String): void**. Método que se comunica con el modelo para obtener la lista de pacientes de la clínica del fisioterapeuta que ha iniciado sesión. Para conocer la clínica en cuestión, se hace llegar por parámetros el identificador de la misma.
- **tratarPaciente(idPaciente: String): void**. Notifica al presentador que un determinado paciente de la lista ha sido presionado. Este método solicita la navegación a la vista *VistaFichaPaciente*, haciéndole llegar a la misma el identificador del paciente, que será clave para el funcionamiento de esta nueva vista.
- **tratarBotonAgregar(): void**. Notifica al presentador que el botón "agregar" de la vista ha sido presionado. Este método solicita la navegación a la vista *VistaAgregarEditarPaciente*.

3.5.5 Clase PresentadorFichaPaciente

Presentador de la vista *VistaFichaPaciente* que implementa la interfaz *IPresentadorFichaPaciente*, que define los siguientes métodos:

- **obtenerInfoPaciente(idPaciente: String): void**. Método que se comunica con el modelo para obtener la información del paciente con identificador dado por parámetros.
- **obtenerEventosProximos(nombrePaciente: String): void**. Método que se comunica con el modelo para obtener la lista de eventos próximos del paciente, que se identifica a partir parámetro *nombrePaciente*.
- **obtenerImagenesPaciente(idPaciente: String): void**. Método que se comunica con el modelo para obtener las imágenes existentes del paciente con identificador dado por parámetro.
- **obtenerNotasPaciente(idPaciente: String): void**. Método que se comunica con el modelo para obtener las notas del paciente con identificador dado por parámetro.
- **tratarBotonCalendario(): void**. Notifica al presentador que se ha presionado el botón "calendario" habilitado en la solapa "INFO", como acceso rápido a añadir un nuevo evento. Este método solicita la navegación a la vista *VistaAgregarEditarEvento*.
- **tratarBotonEditar(idPaciente: String): void**. Notifica al presentador que se ha presionado el icono *Lápiz* que se encuentra en el cabecero de la pantalla, en la solapa "INFO". Este método solicita la navegación a la vista *VistaAgregarEditarPaciente*, haciéndole llegar el identificador del paciente a editar.

- **tratarBotonEliminar(): void**. Notifica al presentador que ha sido presionado el icono *Papelera* que se encuentra en la barra cabecera de la pantalla, en la solapa "INFO". Este método se encargará de interactuar con la vista para solicitarle que muestre el diálogo personalizado para esta opción.

- **tratarDialogoEliminar(idPaciente: String): void**. Notifica al presentador que el botón "Eliminar" del diálogo dedicado a eliminar un paciente ha sido presionado. Este método se encargará de comunicarse con el modelo para indicarle, mediante el identificador que entra por parámetros, qué paciente ha de eliminar.

- **tratarBotonAgregarImagen(idPaciente: String): void**. Notifica al presentador que se ha presionado el botón "+" que se encuentra expuesto en la pantalla, en la solapa "IMÁGENES". Este método solicita la navegación a la vista *VistaAgregarImagen*, indicándole el identificador del paciente para el cual se pretende añadir esta imagen.

- **tratarImagen(idImagen: String): void**. Notifica al presentador que se ha presionado una determinada imagen del archivo de imágenes del paciente. Este método solicita la navegación a la vista *VistaDetalleImagen*, haciéndole llegar el identificador de la imagen que ha sido presionada.

- **tratarBotonAgregarNota(idPaciente: String): void**. Notifica al presentador que se ha presionado el botón "+" que se encuentra expuesto en la pantalla, en la solapa "NOTAS". Este método solicita la navegación a la vista *VistaAgregarEditarNota*, indicándole el identificador del paciente para el cual se pretende añadir la nota.

- **tratarBotonEditarNota(): void**. Notifica al presentador que se ha presionado en el item de menú "Editar", que se encuentra en el menú contextual de cada nota, en la solapa "NOTAS". Este método solicita la navegación a la vista *VistaAgregarEditarNota*, haciéndole llegar el identificador de la nota a editar.

- **tratarBotonEliminarNota(): void**. Notifica al presentador que ha sido presionado el item de menú "Eliminar", que se encuentra en el menú contextual de cada nota, en la solapa "NOTAS". Este método se encargará de interactuar con la vista para solicitarle que muestre el diálogo personalizado para esta opción.

- **tratarDialogoEliminarNota(idNota: String): void**. Notifica al presentador que el botón "Eliminar" del diálogo, dedicado a eliminar una nota, ha sido presionado. Este método se encargará de comunicarse con el modelo para indicarle, mediante el identificador que entra por parámetros, qué nota ha de eliminar.

3.5.6 Clase PresentadorAgregarEditarPaciente

Presentador de la vista *VistaAgregarEditarPaciente* que implementa la interfaz *IPresentadorAgregarEditarPaciente*, que define los siguientes métodos:

- **tratarBotonFinalizar(paciente: Paciente): void**. Notifica al presentador que el botón "check" de la barra cabecera de la vista ha sido presionado. Este método se encargará de comunicarse con el modelo para pasarle el paciente, bien sea porque es nuevo o ha sido modificado, en función de para qué ha sido habilitada la vista, lo cual también se le indica al modelo, para que sepa qué operación aplicar.

3.5.7 Clase PresentadorAgregarImagen

Presentador de la vista *VistaAgregarImagen* que implementa la interfaz *IPresentadorAgregarImagen*, que define los siguientes métodos:

- **tratarBotonFinalizar(imagen: Imagen): void**. Notifica al presentador que el botón "check" de la barra cabecera de la vista ha sido presionado. Este método se encargará de comunicarse con el modelo para pasarle la imagen que el usuario pretende añadir.

3.5.8 Clase PresentadorDetalleImagen

Presentador de la vista *VistaDetalleImagen* que implementa la interfaz *IPresentadorDetalleImagen*, que define los siguientes métodos:

- **obtenerImagen(idImagen: String): void**. Método que solicita al modelo que le proporcione la imagen para la cual se está cargando la vista detalle, proporcionándole el identificador de la imagen.
- **tratarBotonEliminar(): void**. Notifica al presentador que el icono "Papelera" de la barra cabecera de la vista ha sido presionado. Este método se encargará de comunicarse con la vista para que muestre el diálogo específico que solicite la confirmación al usuario, al respecto de continuar o no con la eliminación de la imagen que se detalla.
- **tratarDialogoEliminar(idImagen: String): void**. Notifica al presentador que el botón "Eliminar" del diálogo dedicado a eliminar una imagen ha sido presionado. Este método se encargará de comunicarse con el modelo para indicarle, mediante el identificador que entra por parámetros, qué imagen ha de eliminar.

3.5.9 Clase PresentadorAgregarEditarNota

Presentador de la vista *VistaAgregarEditarNota* que implementa la interfaz *IPresentadorAgregarEditarNota*, que define los siguientes métodos:

- **tratarBotonFinalizar(nota: Nota): void**. Notifica al presentador que el botón "check" de la barra cabecera de la vista ha sido presionado. Este método se encargará de comunicarse con el modelo para pasarle la nota, bien sea porque es nueva o ha sido modificada, en función de para qué ha sido habilitada la vista, lo cual también se le indica al modelo, para que sepa que operación realizar.

3.5.10 Clase PresentadorCalendario

Presentador de la vista *VistaCalendario* que implementa la interfaz *IPresentadorCalendario*, que define los siguientes métodos:

- **tratarItemInicio(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaInicio*.
- **tratarItemPacientes(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPacientes*.
- **tratarItemCalendario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaCalendario*.
- **tratarItemTareas(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaTareas*.
- **tratarItemPagos(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPagos*.
- **tratarItemMiUsuario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaUsuario*.
- **tratarDiaCalendario(fecha: Object): void**. Notifica al presentador que un determinado día del calendario ha sido presionado. Este método solicita que se navegue a la vista *VistaEventos*, haciéndole llegar la fecha que ha presionado el usuario en el calendario.
- **tratarBotonAgregar(): void**. Notifica al presentador que el botón "+" de la vista ha sido presionado. Este método solicita la navegación a la vista *VistaAgregarEditarEvento*.

3.5.11 Clase PresentadorEventos

Presentador de la vista *VistaEventos* que implementa la interfaz *IPresentadorEventos*, que define los siguientes métodos:

- **obtenerListaEventos(dia: Object): void**. Método que se comunica con el modelo para obtener la lista de eventos que tiene la clínica, un determinado día que se recibe por parámetros.
- **tratarBotonAgregar(): void**. Notifica al presentador que el botón "+" de la vista ha sido presionado. Este método solicita la navegación a la vista *VistaAgregarEditarEvento*.
- **tratarBotonEditar(idEvento: String): void**. Notifica al presentador que el icono *Lápiz* del menú contextual de algún evento ha sido presionado. Este método solicita la navegación a la vista *VistaAgregarEditarEvento*, haciéndole llegar el identificador del evento que se desea editar.
- **tratarBotonEliminar(): void**. Notifica al presentador que el icono "Papelera" del menú contextual de un determinado evento de la lista ha sido presionado. Este método solicita a la vista *VistaEventos* que muestre el diálogo personalizado para esta opción.
- **tratarDialogoEliminar(idEvento: String): void**. Notifica al presentador que el botón "Eliminar" del diálogo dedicado a eliminar un evento ha sido presionado. Este método se encargará de comunicarse con el modelo para indicarle, mediante el identificador que entra por parámetros, qué evento se desea eliminar.

3.5.12 Clase PresentadorAgregarEditarEvento

Presentador de la vista *VistaAgregarEditarEvento* que implementa la interfaz *IPresentadorAgregarEditarEvento*, que define los siguientes métodos:

- **tratarBotonFinalizar(evento: Evento): void**. Notifica al presentador que el botón "check" de la barra cabecera de la vista ha sido presionado. Este método se encargará de comunicarse con el modelo para pasarle el evento, bien sea porque es nuevo o ha sido modificado, en función de para qué ha sido habilitada la vista, lo cual también se le indica al modelo, para que sepa que operación realizar.

3.5.13 Clase PresentadorTareas

Presentador de la vista *VistaTareas* que implementa la interfaz *IPresentadorTareas*, que define los siguientes métodos:

- **tratarItemInicio(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaInicio*.
- **tratarItemPacientes(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPacientes*.
- **tratarItemCalendario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaCalendario*.
- **tratarItemTareas(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaTareas*.
- **tratarItemPagos(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPagos*.
- **tratarItemMiUsuario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaUsuario*.
- **obtenerListaTareas(idClinica: String): void**. Método que se comunica con el modelo para obtener la lista de tareas por hacer, que tiene la clínica del fisioterapeuta que ha iniciado sesión. Esta clínica se le hace llegar por parámetros mediante el identificador de la misma.
- **tratarBotonAgregar(): void**. Notifica al presentador que el botón "+" de la vista ha sido presionado. Este método se encargará de interactuar con la vista para solicitarle que muestre el diálogo personalizado para agregar una nueva tarea.
- **tratarDialogoAgregar(tarea: Tarea): void**. Notifica al presentador que el botón "Aceptar" del diálogo, dedicado a agregar tareas, ha sido presionado. Este método se encargará de comunicarse con el modelo para entregarle la tarea que el usuario desea añadir.
- **tratarBotonFinalizar(): void**. Notifica al presentador que el botón "check" de una determinada tarea de la lista ha sido presionado. Este método se encargará de interactuar con la vista para solicitarle que muestre el diálogo personalizado para esta ocasión.
- **tratarDialogoFinalizar(idTarea: String): void**. Notifica al presentador que el botón "Finalizar" del diálogo dedicado a finalizar una tarea ha sido presionado. Este método se encargará de comunicarse con el modelo para indicarle, mediante el identificador que entra por parámetros, qué tarea se desea eliminar.

3.5.14 Clase PresentadorPagos

Presentador de la vista *VistaPagos* que implementa la interfaz *IPresentadorPagos*, que define los siguientes métodos:

- **tratarItemInicio(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaInicio*.
- **tratarItemPacientes(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPacientes*.
- **tratarItemCalendario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaCalendario*.
- **tratarItemTareas(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaTareas*.
- **tratarItemPagos(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPagos*.
- **tratarItemMiUsuario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaUsuario*.
- **obtenerListaRealizados(idClinica: String): void**. Método que se comunica con el modelo para obtener la lista de pagos realizados, que tiene la clínica del fisioterapeuta que ha iniciado sesión. El identificador de la clínica se recibe por parámetros.
- **obtenerListaPendientes(idClinica: String): void**. Método que se comunica con el modelo para obtener la lista de pagos pendientes, que tiene la clínica del fisioterapeuta que ha iniciado sesión. El identificador de la clínica se recibe por parámetros.
- **tratarBotonAgregar(): void**. Notifica al presentador que el botón "+" de la vista ha sido presionado. Este método se encargará de interactuar con la vista para solicitarle que muestre el diálogo personalizado para agregar un nuevo pago.
- **tratarDialogoAgregar(pago: Pago): void**. Notifica al presentador que el botón "Añadir" del diálogo, dedicado a agregar pagos, ha sido presionado. Este método se encargará de comunicarse con el modelo para entregarle el pago que el usuario desea añadir.
- **tratarBotonRealizado(): void**. Notifica al presentador que el botón "check" de un determinado pago, de la lista de pagos pendientes, ha sido presionado. Este método se encargará de interactuar con la vista para solicitarle que muestre el diálogo personalizado para esta ocasión.

- **tratarDialogoRealizado(idPago: String): void**. Notifica al presentador que el botón "Aceptar" del diálogo, dedicado a finalizar un pago, ha sido presionado. Este método se encargará de comunicarse con el modelo para indicarle, mediante el identificador que entra por parámetros, qué pago se desea eliminar.

3.5.15 Clase PresentadorUsuario

Presentador de la vista *VistaUsuario* que implementa la interfaz *IPresentadorUsuario*, que define los siguientes métodos:

- **tratarItemInicio(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaInicio*.
- **tratarItemPacientes(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPacientes*.
- **tratarItemCalendario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaCalendario*.
- **tratarItemTareas(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaTareas*.
- **tratarItemPagos(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaPagos*.
- **tratarItemMiUsuario(): void**. Método que hará que el presentador solicite la navegación a la vista *VistaUsuario*.
- **obtenerInfoUsuario(idFisioterapeuta: String): void**. Método que se comunica con el modelo para obtener la información del usuario que ha iniciado sesión en la aplicación, a través del identificador que se hace llegar por parámetros.
- **tratarBotonEditar(): void**. Notifica al presentador que el botón *Editar* del menú de la barra cabecera de la vista, ha sido presionado. Este método se encargará de interactuar con la vista para solicitarle que muestre un diálogo personalizado, para continuar con el proceso de edición de la cuenta del usuario.
- **tratarDialogoEditar(usuario: Fisioterapeuta): void**. Notifica al presentador que el botón "Aceptar" del diálogo, dedicado a la edición de la cuenta de usuario, ha sido presionado. Este método se encargará de comunicarse con el modelo para entregarle la información del usuario modificado.
- **tratarBotonCambiarPassword(): void**. Notifica al presentador que el botón "Cambiar contraseña" del menú de la barra cabecera de la vista, ha sido presionado. Este método se encargará de interactuar con la vista para solicitarle que muestre el diálogo personalizado para esta ocasión.

- **tratarDialogoCambiarPassword(idFisioterapeuta: String, password: String): void**. Notifica al presentador, que el botón "Aceptar" del diálogo dedicado al cambio de contraseña, ha sido presionado. Este método se encargará de comunicarse con el modelo para indicarle la nueva contraseña que ha de modificar en un determinado fisioterapeuta, cuyo identificador se entrega por parámetros.

- **tratarBotonDarBaja(): void**. Notifica al presentador que el botón "Dar de baja" del menú de la barra cabecera de la vista, ha sido presionado. Este método se encargará de interactuar con la vista para solicitarle que muestre un diálogo para requerir la confirmación oportuna al usuario.

- **tratarDialogoDarBaja(idFisioterapeuta: String): void**. Notifica al presentador que el botón "Aceptar" del diálogo, dedicado a dar de baja al usuario, ha sido presionado. Este método se encargará de comunicarse con el modelo para indicarle la cuenta de usuario que ha de dar de baja de la aplicación, a través de un identificador que llega por parámetros.

Capítulo **4**

MODELO DE IMPLEMENTACIÓN

En este capítulo se presenta cómo se adapta el modelo de diseño planteado en el capítulo anterior, al lenguaje de programación elegido. Inicialmente, se presenta algunas características de la plataforma Android para, a continuación, indicar las modificaciones del diseño en función de estas características.

4.1 Adecuación a Android

Android es un conjunto de herramientas de software de código abierto para dispositivos móviles que fueron creadas por *Google* y la *Open Handset Alliance*. Android proporciona un *framework* de aplicaciones que permiten a los desarrolladores construir aplicaciones innovadoras escribiendo código usando el lenguaje Java, y controlando el dispositivo a través de bibliotecas Java.

Android proporciona un marco de aplicación adaptable a diferentes dispositivos móviles que le permite proporcionar recursos únicos para diferentes configuraciones. Esto se consigue mediante carpetas específicas en el directorio *res* del proyecto creado (para iconos en las carpetas *drawable*, para los literales, dimensiones, estilos, en las carpetas *values* y para los layouts de las pantallas en las carpetas *layout*).

Las herramientas del SDK de Android compilan el código junto con todos los datos y recursos en un *paquete de Android* (*APK*). Un archivo *APK* contiene todo el contenido de una aplicación para Android y es el archivo que los dispositivos con Android utilizan para instalar la aplicación. Una vez instalado en un dispositivo, cada aplicación Android vive en su propio entorno limitado en seguridad. De esta forma, el sistema Android implementa el principio de *privilegios mínimos*. Cada aplicación sólo tiene acceso a los componentes que necesita para hacer su trabajo y nada más. Esto crea un ambiente muy seguro en el que una aplicación no puede tener acceso a partes del sistema para el cual no se le ha dado permiso (los permisos se solicitan en el archivo *manifiesto* propio de cada proyecto).

Las aplicaciones Android se construyen como una combinación de componentes que pueden ser invocados individualmente. Así, desde un componente se puede iniciar otro, lo que se hace en Android mediante un *Intent*. Los componentes de una aplicación son los componentes esenciales para su construcción. Cada componente

es un punto diferente a través del cual el sistema puede entrar a su aplicación. No todos los componentes son los puntos de entrada reales para el usuario y en algunos casos, unos componentes dependen de otros, pero cada uno existe como una entidad propia y juega un rol específico que ayuda a definir el comportamiento general de la aplicación.

En Android hay cuatro tipos diferentes de componentes. Cada tipo tiene un propósito distinto y tiene un ciclo de vida diferente:

- **Componente *Activity*: actividad**. Un *activity* representa una única pantalla con una interfaz de usuario. Por ejemplo, una aplicación de correo electrónico puede tener una actividad que muestra una lista de los nuevos mensajes de correo electrónico, otra actividad para redactar un correo electrónico, y otra de las actividades para la lectura de mensajes de correo electrónico. Aunque las actividades trabajan juntas para formar una experiencia de usuario coherente en la aplicación de correo electrónico, cada una es independiente de las demás. Una actividad de este estilo se implementa como una subclase de la clase *Activity*.

- **Componente *Service*: servicio**. *Service* es un componente que se ejecuta en un segundo plano para realizar operaciones de larga ejecución o para realizar un trabajo para procesos remotos. Este componente no proporciona una interfaz de usuario. Un ejemplo podría ser a la hora escuchar la radio en segundo plano mientras el usuario está en una aplicación diferente, o cuando obtiene datos sobre la red sin bloquear la interacción del usuario con una actividad. Un componente *service* se implementa como una subclase de la clase *Service*.

- **Componente *Content provider*: proveedor de contenido**. Un *Content provider* gestiona un conjunto compartido de datos de aplicaciones. Puede almacenar los datos en el sistema de archivos, una base de datos, en la web, o cualquier otro lugar de almacenamiento permanente donde la aplicación pueda tener acceso. A través del *content provider*, otras aplicaciones pueden consultar o incluso modificar los datos. Un ejemplo sería cuando el sistema Android ofrece un *content provider* que gestiona la información de contacto del usuario. Un *proveedor de contenido* se implementa como una subclase de la clase *ContentProvider* y debe implementar un conjunto estándar de APIs que permitan a otras aplicaciones realizar transacciones.

- **Componente *Broadcast receivers*: receptores de radiodifusión**. Un *Broadcast receivers* es un componente que responde a transmitir anuncios de todo el sistema. Muchas de estas emisiones se originan en el propio sistema, por ejemplo, a la hora de anunciar que la pantalla se ha apagado, la batería está baja, o una imagen ha sido capturada. Las aplicaciones también pueden iniciar este tipo de emisiones, por ejemplo, cuando algunos datos se han descargado en el dispositivo, la aplicación puede avisar de que está disponible para que los pueda utilizar. Aunque los receptores de radiodifusión no muestran una interfaz de usuario, es posible crear una notificación en la barra de estado

para alertar al usuario cuando se produce un evento de difusión. Un *receptor de radiodifusión* se implementa como una subclase de *BroadcastReceiver* y cada emisión se entrega como un objeto *Intent*.

Un aspecto único del diseño del sistema Android es que cualquier aplicación puede iniciar el componente de otra aplicación. Por ejemplo, si se deseara capturar una fotografía con la cámara del dispositivo, es probable que exista otra aplicación que haga esto y que su aplicación pueda utilizarlo, en lugar de desarrollar una actividad para capturar una fotografía por sí mismo. No es necesario incorporar o incluso enlazar con el código de la aplicación de la cámara. En su lugar, sólo tiene que iniciar la actividad en la aplicación de la cámara que captura una foto. Cuando se haya completado la operación, la foto no regresará a su aplicación, la de la cámara, sino a la que lo llamó para realizar la acción, y para el usuario parece como si la cámara es en realidad una parte de su aplicación.

4.2 La clase AppMediador

Como se comentó en el *Modelo de diseño*, la aplicación utiliza la arquitectura *Modelo-Vista-Presentador* (MVP) adaptada al lenguaje elegido (Java). En esta adaptación, se incorpora una nueva clase, llamada *AppMediador* (introducida en el *Modelo de Diseño* con el nombre *mediador*) que va a funcionar como punto de enlace de las distintas partes de la arquitectura. El *AppMediador* representa a la clase *Application* y se encarga de:

- Definir los presentadores y vistas y los métodos *accessor* de éstos.
- Definir la navegación en la aplicación, es decir, qué vistas (de la interfaz de usuario) se deben lanzar cuando se considere oportuno (normalmente, ante peticiones del usuario). Asimismo, definir qué servicios se lanzan cuando sea necesario.
- Definir las constantes correspondientes a los avisos de notificación, enviadas desde el modelo cuando se haya terminado de realizar una determinada acción.
- Definir las constantes correspondientes a los datos comunicados entre vistas, o recuperados desde el modelo, entre otros, ya que en Android los datos se comunican usando pares *clave, valor*.
- Definir los métodos de manejo de los componentes de Android, correspondientes a: el lanzamiento de actividades, la creación de servicios, los registros de receptores de notificación, los envíos de notificaciones *broadcast*, entre otros.

Es importante comentar que para que la aplicación desarrollada utilice este objeto *AppMediador*, en el archivo *AndroidManifest.xml* hay que indicar que el nombre de la aplicación corresponde con este archivo (indicando el paquete en el que se encuentra el mismo). Para ello, en este archivo hay que añadir, dentro de la etiqueta *application*, lo siguiente:

```
<application
        android:name="tfg.android.jhr.AppMediador"
        ...>
</application>
```

El *AppMediador* contiene una variable de instancia (*private*) por cada una de las interfaces del presentador y la vista de la aplicación desarrollada, además de la variable *singleton* (*private* y *static*) de tipo *AppMediador*, para poder acceder a éste desde cualquier punto.

Por otro lado, se definen unas constantes de petición y notificación (*public* y *static*) que serán las claves de los datos a comunicar (clave, valor) o los avisos de notificación. Las constantes de notificación se utilizarán en los objetos *BroadcastReceiver* y en los métodos *sendBroadcast* para notificar cuándo ha finalizado un evento en la aplicación (por ejemplo, cuando se notifica la inserción de datos en la base de datos). Estos serán utilizados por los presentadores y el modelo.

Además, se implementan los siguientes métodos:

- Los métodos *accessor* de las vistas definidas en el capítulo *Modelo de Diseño*: *getVistaXXX* y *setVistaXXX*, donde *XXX* corresponde con el nombre de cada una de las vistas. Estos métodos son utilizados para obtener o iniciar las variables de las actividades (*Activity* de Android).
- Los métodos de creación y eliminación de los distintos presentadores definidos en el *Modelo de Diseño*: *getPresentadorXXX* y *removePresentadorXXX*, donde *XXX* corresponde con el nombre de cada uno de los presentadores.
- Los métodos de lanzamiento de actividades, servicios y receptores *broadcast*:
  - **launchActivity**: lanza una actividad.
  - **launchActivityForResult**: lanza una actividad y espera un resultado de respuesta.
  - **launchActivityWithFinish**: lanza una actividad y termina la actividad invocadora (sacándola de la pila).
  - **finishActivity**: finaliza una actividad, sacándola de la pila.
  - **launchService**: lanza un servicio.
  - **stopService**: para un servicio previamente lanzado.

  - **registerReceiver**: registra un receptor a la espera de una determinada notificación.
  - **unRegisterReceiver**: des-registra un receptor.
  - **sendBroadcast**: envía una notificación a todos los receptores (sólo aquel que espere la notificación enviada, la atenderá).

- Los métodos de navegación para cada una de las vistas: *getVistaParaXXX*, donde *XXX* corresponde con el nombre de cada una de las vistas definidas en el *Modelo de diseño*. Estos métodos devuelve un objeto de tipo *Class* que representa a la clase de la vista correspondiente.

- Y por último, el método que crea la aplicación: *onCreate*. Este método inicia todas las variables de los presentadores a *null* e indica que la clase *AppMediador* es un *singleton*.

4.3 Paquete modelo

En este apartado se explican los cambios y adaptaciones que se han tenido que realizar en las clases e interfaces del paquete **modelo** con respecto a las definidas en el *Modelo de diseño*. Así, en este paquete, y debido a la necesidad de encriptar información sensible, se ha creado un nuevo paquete, llamado *helpers*, en el que se definen las siguientes clases:

- Clase **AESHelper**. Se trata de una clase que cuenta con una serie de métodos que hacen posible las tareas de encriptación y desencriptación de la información delicada con la que trata la aplicación. Esta clase fue obtenida gracias a un aporte del usuario "varotariya vajsi" en la comunidad *StackOverflow* [22].

- Clase **CryptoProvider**. Esta clase es necesaria para el correcto funcionamiento de la clase anterior. Se trata de una clase que funciona como *proveedor de encriptación* y es necesario crear un objeto manualmente, debido a que, a partir de la API 24 de Android [23], el que se aportaba por parte del SDK de Android, ha quedado obsoleto. Esta clase fue obtenida gracias al aporte del usuario "varotariya vajsi" en la comunidad *StackOverflow* [24].

Asimismo, para facilitar el manejo de las imágenes dentro de la aplicación, y evitar tener que acceder a la nube más de lo necesario, se crea la clase:

- Clase **ContenedorImagen**. Clase con dos atributos: "imagen" de tipo *array de bytes*, que corresponde con la imagen como tal, y "registroImagen", de tipo *Imagen*, que corresponde con el registro de la imagen en la base de datos. De esta manera se consigue tratar con las imágenes pudiendo llevar consigo la imagen y su información. Además, la clase cuenta con los métodos *getter* y *setter* de las variables mencionadas previamente.

A continuación se muestra de forma detallada, cómo se ha modificado cada clase del paquete modelo, que ya fueron definidas en el capítulo *Modelo de diseño*.

4.3.1 Clase Modelo

Los cambios en la clase *Modelo* son los siguientes:

- Métodos **obtenerListaPacientes, obtenerListaTareas, obtenerListaRealizados, obtenerListaPendientes**. Se elimina el parámetro "idClinica" ya que la clase almacena la información del "fisioterapeutaActual", en uno de sus atributos. A través de este atributo se podrá obtener directamente el identificador de la clínica a la que pertenece dicho fisioterapeuta.
- Método **comprobarCorreo**. Su objetivo inicial era comprobar la existencia de un correo electrónico en la base de datos de la aplicación y, en tal caso, entregar la contraseña al usuario que la ha olvidado. Dado que se delega en *Firebase Authentication* la gestión de la autenticación de la aplicación, se replantea el nombre de este método para hacerlo acorde a la nueva realidad. El nombre, por tanto, del nuevo método será *obtenerPassword* (que se ajusta mejor a su funcionalidad).
- Método **obtenerImagen**. Se modifica el parámetro "idImagen" de tipo *String* a tipo *int*, ya que estará referido a una posición dentro de la lista de imágenes que ya ha sido previamente obtenida y se encuentra en la clase *BDAdaptadorImagen*. De esta manera no será necesario volver a la base de datos para obtener la imagen deseada, sino que se toma de la lista ya existente.
- Método **obtenerInfoPaciente**. Se modifica el parámetro "idPaciente" de tipo *String* a tipo *int*, ya que estará referido a una posición dentro de la lista de pacientes que ya ha sido previamente obtenida y se encuentra en la clase *BDAdaptadorPaciente*. De esta manera no será necesario volver a la base de datos para obtener el paciente deseado, sino que se toma de la lista ya existente.
- Métodos **eliminarNota** y **eliminarImagen**. Se añade el parámetro "idPaciente" debido a que ambos métodos lo necesitan para el acceso al nodo específico del paciente en la base de datos.
- Método **obtenerInfoUsuario**. Se elimina parámetro "idUsuario" ya que no será necesario volver a conectar con la nube para obtener el usuario, ya que esta información está almacenada en los atributos de la clase.
- Método **actualizarInfoUsuario.** Se modifican los parámetros, que pasan de "datos" de tipo *Object* a una serie de parámetros, que corresponden con cada uno de los atributos de un fisioterapeuta o usuarios, que se desea actualizar: "password", "nombre", "correoNuevo" y "telefono", todos de tipo *String*.

- Método **darBajaUsuario**. Se sustituye el parámetro "idFisioterapeuta" de tipo *String*,debido a que no es necesario, por el parámetro "password" de tipo *String*, para garantizar que la persona que desea dar de baja la cuenta es el fisioterapeuta dueño de dicha cuenta, además de ser un requerimiento de *Firebase Authentication* para este proceso.
- Método **actualizarPassword**. Se elimina el parámetro "correo", ya que no es necesario, al tener esta información (correo electrónico) guardada en el atributo "fisioterapeutaActual", de tipo *Fisioterapeuta*, que contiene la información del fisioterapeuta que ha iniciado sesión.

Asimismo, a continuación se presentan los nuevos métodos necesarios, en la clase *Modelo*, y que no fueron detectados en el diseño de la aplicación. Estos métodos son:

- Método **obtenerInfoInicial**. Permite la obtención de un contenedor de información importante para la aplicación, a través de la comunicación con el método homónimo que se encuentra en la clase *BDAdaptadorClinica*. Este contenedor, llamado *infoInicial* contiene el nombre del fisioterapeuta que ha iniciado sesión, el nombre de la clínica a la que pertenece y su logotipo, y se trata de información necesaria para diferentes elementos de la vista.
- Método **deslogearUsuario.** Encargado de desconectar el usuario que ha iniciado sesión y se encuentra utilizando la aplicación en dicho momento.
- Método **obtenerListaFisioterapeutas**. Notifica la lista de fisioterapeutas que tiene una clínica de fisioterapia, la cual corresponde con la clínica del fisioterapeuta que ha iniciado sesión en la aplicación.

4.3.2 Clase BDAdaptadorClinica

Inicialmente estaba previsto que esta clase estuviera vacía, pero finalmente ha sido necesario añadir una serie de métodos para la funcionalidad de la aplicación, que no fueron detectados en el diseño. Estos métodos son los siguientes:

- Método **obtenerInfoInicial**. Este método se encarga de notificar la información inicial necesaria tras haber realizado el *login*, que corresponde con el nombre del fisioterapeuta, nombre de la clínica y el logo de la misma. Por ello, tiene como parámetros "nombreFisio" y "idClinica", y con este último será posible obtener el logotipo de la clínica en el almacenamiento en la nube. Esta información, guardada en forma de *array de Object*, permanecerá hasta que el usuario desconecte de la aplicación, consiguiendo así que no sea necesario estar conectando con la nube para conseguir esta información constantemente, sobre todo teniendo en cuenta que esta información permanece prácticamente constante.

- Método **setInfoInicial**. Método *setter* necesario para establecer el atributo "infoInicial" explicado anteriormente, a través del parámetro de mismo nombre que llega por parámetros, de tipo *array de Object*. Se utilizará la primera vez que se conecte con la nube para obtener dicha información, y además, cuando se desconecte el usuario de la aplicación, con el objetivo de limpiar dicha información, por si hay una futura conexión de otro usuario en el mismo uso de la aplicación.

- Método **actualizarNombreFisioInfoInicial**. Este método, cuyo parámetro es "nombreFisio" de tipo *String*, será únicamente necesario en caso de que el usuario haya actualizado su cuenta de usuario el atributo "nombre", y por tanto será necesario una actualización en el contenedor "infoInicial", en el índice correspondiente al nombre del fisioterapeuta. Esto es debido a que, como se explicó previamente, este contenedor no se actualiza constantemente en la nube al sufrir cambios raramente, como es este caso.

4.3.3 Clase BDAdaptadorFisioterapeuta

Inicialmente esta previsto que esta clase estuviera vacía, pero finalmente ha sido necesario añadir un método para la funcionalidad de la aplicación, que no fue detectado en el diseño. Este método es el siguiente:

- **obtenerListaFisioterapeutas(idClinica: String): void**. Realiza una consulta, en la tabla *fisioterapeutas*, con el objetivo de encontrar los fisioterapeutas que pertenecen a una determinada clínica, la cual concuerda con el identificador de la clínica que se pasa por parámetros, *idFisioterapeuta*. El método notifica el envío de la información solicitada.

4.3.4 Clase BDAdaptadorPaciente

Los cambios de la clase *BDAdaptadorPaciente* son los siguientes:

- Métodos **obtenerPaciente, eliminarPaciente**. Se añade el parámetro "idClinica" de tipo *String* en ambos métodos, que corresponde con un identificador de clínica, y que será necesario para acceder al nodo (registro) específico del paciente en la nube.

4.3.5 Clase BDAdaptadorImagen

Los cambios de la clase *BDAdaptadorImagen* son los siguientes:

- Método **obtenerImagenesPaciente**. Se añade el parámetro "idClinica" de tipo *String*, que corresponde con un identificador de clínica, ya que será necesario para el acceso al nodo (registro) específico, en la base de datos, de la clínica a la que pertenece el paciente del que se quiere obtener las imágenes.

- Método **obtenerImagen**. Se modifica el parámetro "idImagen" de tipo *String* a tipo *int*, ya que ahora corresponde con un índice o posición de la lista de imágenes, que será un atributo de la clase (la lista ha sido previamente cargada en otro método de la clase).

- Método **eliminarImagen**. Se añaden los parámetros "idPaciente" e "idClinica", ambos de tipo *String*, necesarios para el acceso al nodo (registro) específico, en la base de datos, de la imagen que se desea eliminar, y el acceso a la ruta de la imagen como tal, en el almacenamiento en la nube.

4.3.6 Clase BDAdaptadorNota

Los cambios de la clase *BDAdaptadorNota* son los siguientes:

- Método **obtenerNotasPaciente**. Se añade el parámetro "idClinica" de tipo *String*, que corresponde con un identificador de clínica y es necesario para el acceso al nodo (registro), en la base de datos, específico de las notas de un paciente.

- Método **eliminarNota**. Se añaden los parámetros "idPaciente" e "idClinica", que serán necesarios para el acceso al nodo (registro) específico de la nota que se pretende eliminar.

4.3.7 Clase BDAdaptadorEvento

Los cambios de la clase *BDAdaptadorEvento* son los siguientes:

- Método **obtenerEventosProximos**. Se añade el parámetro "idClinica", que será necesario para acceder al nodo específico de la clínica a la que pertenece el paciente sobre el que se desea obtener la lista de eventos próximos.

- Método **eliminarEvento**. Se añade el parámetro "idClinica", que será necesario para el acceso al nodo (registro) específico del evento que se pretende eliminar.

4.3.8 Clase BDAdaptadorTarea

Los cambios de la clase *BDAdaptadorTarea* son los siguientes:

- Método **eliminarTarea**. Se añade el parámetro "idClinica", que será necesario para el acceso al nodo (registro) específico de la tarea que se pretende eliminar.

4.3.9 Clase BDAdaptadorPago

Los cambios de la clase *BDAdaptadorPago* son los siguientes:

- Método **setPagoRealizado**. Se añade el parámetro "idClinica", que será necesario para el acceso al nodo (registro) específico del pago que se pretende actualizar.

4.4 Paquete vista

En este apartado se explican los cambios y adaptaciones que se han tenido que realizar en las clases e interfaces del paquete **vista**, con respecto a las definidas en el *Modelo de Diseño*.

En primer lugar, teniendo en cuenta la relación evidente de este paquete vista con la interfaz de usuario, se decidió crear un subpaquete denominado *adaptadores*, que contiene cada uno de los adaptadores utilizados en la aplicación. Los adaptadores son elementos proporcionados por Android [25], [26] y son muy importantes al funcionar como puente entre la información y los elementos dinámicos de la interfaz de usuario, como pueden ser las listas o las solapas de una determinada vista. Las clases que se encuentra en este paquete son:

- Clase **AdapterEventos**. Consiste en un adaptador para un elemento de tipo *RecyclerView*. Este adaptador permitirá listar los eventos que se hayan obtenido, pudiendo mostrar en un formato de tarjetas, cada uno de los atributos de cada evento que pertenece a esa lista, siendo cada tarjeta un evento en concreto.
- Clase **AdapterEventosProximos**. Consiste en un adaptador para un elemento de tipo *ListView*. Este adaptador permitirá listar, en un determinado formato, los eventos próximos que tiene un paciente específico, del cual se está visualizando su ficha de paciente.

- Clase **AdapterImagenesPaciente**. Corresponde con un adaptador utilizado en un componente llamado *GridView*. Gracias a él se puede visualizar las distintas imágenes de un determinado paciente en forma de *grid*.

- Clase **AdapterNotas**. Es un adaptador que se utiliza junto a un componente de tipo *RecyclerView*. Con este adaptador se podrá mostrar al usuario una serie de notas obtenidas pertenecientes a un determinado paciente, pudiéndolo hacer en un formato basado en tarjetas. De esta forma, cada tarjeta será para una nota específica, pudiendo ver sus campos diferenciados.

- Clase **AdapterPacientes**. Consiste en un adaptador para un elemento de tipo *ListView*. Este adaptador permitirá listar cada uno de los pacientes con los que cuenta la clínica del fisioterapeuta que ha iniciado sesión, visualizando el nombre de cada uno de ellos.

- Clase **AdapterPagos**. Este adaptador es usado con el objetivo de poder listar cada uno de los pagos con los que cuenta una clínica, tanto realizados como pendientes. Este adaptador funciona con un elemento tipo *ListView*.

- Clase **AdapterTareas**. Corresponde con un adaptador utilizando de manera conjunta con un elemento de tipo *ListView*, consiguiendo así poder listar cada una de las tareas con las que cuenta una determinada clínica.

- Clase **AdapterTabsFichaPaciente**. Este adaptador permite preparar la *VistaFichaPaciente* para que contenga tres solapas ("INFO", "IMÁGENES" y "NOTAS"), las cuales corresponden con un fragmento cada uno, que se explican posteriormente.

- Clase **AdapterTabsPagos**. Consiste en un adaptador que consigue preparar la *VistaPagos* para que cuente con dos solapas ("REALIZADOS" y "PENDIENTES"). Cada una de estas solapas corresponden con un fragmento, que se explican a continuación.

Por otro lado, también se creó un subpaquete denominado *fragmentos*, que contiene cada uno de los fragmentos que se utilizan en la aplicación. Un fragmento [27] representa un comportamiento o una parte de la interfaz de usuario. En el caso de esta aplicación, se combinan varios con el objetivo de dar lugar a una interfaz de usuario multipanel, basada en solapas, donde cada solapa corresponde con un fragmento. De esta manera se puede visualizar diferentes apartados dentro de una misma vista, a partir del gesto de deslizar el dedo en la pantalla. Los fragmentos creados en esta aplicación son:

- Clase **FragmentoInfoPaciente**. Este fragmento se visualizará en la *VistaFichaPaciente* y formará parte de la solapa "INFO", en la que se mostrará la información personal del paciente, además de sus eventos próximos.

- Clase **FragmentoImagenesPaciente**. Este fragmento será visualizado en la *VistaFichaPaciente*, formando parte de la solapa "IMÁGENES", en la que se mostrará las imágenes del paciente.
- Clase **FragmentoNotasPaciente**. Este fragmento será visualizado en la *VistaFichaPaciente* y se muestra en la solapa "NOTAS". En ella se podrá ver cada una de las notas del paciente elegido.
- Clase **FragmentoPagosRealizados**. Este fragmento pertenece a la *VistaPagos* y formará parte de la solapa "REALIZADOS". En ella se podrá visualizar los diferentes pagos realizados, que se han registrado en la clínica del fisioterapeuta, que ha iniciado sesión.
- Clase **FragmentoPagosPendientes**. Este fragmento pertenece a la *VistaPagos*, dando lugar a la solapa "PENDIENTES". En este caso, se permite visualizar los pagos de tipo pendiente de la clínica.

Asimismo, y debido a la programación en Android, las clases de la vista implementan métodos obligatorios, debido a la herencia y a la implementación de ciertas interfaces. Estos métodos son:

- Método **onCreate**. Este método se lanza cuando el sistema crea la actividad.
- Método **onResume**. Este método se invoca cuando la actividad entra en el *primer plano* de la aplicación, tanto al ser primera vez iniciada, tanto como si ha vuelto de un estado de pausa.
- Método **onPause**. Este método se lanza cuando se va a pasar la actividad a un *segundo plano* dentro de la aplicación.
- Método **onClick**. Este método permite manejar los eventos acerca de cuándo se presiona sobre un determinado elemento de la vista.
- Método **onCreateOptionsMenu**. Este método se encarga de la creación de un menú de opciones, típicamente el menú de opciones que se crea en la barra cabecera de algunas actividades.
- Método **onOptionsItemSelected**. Este método trata la selección de un ítem del menú de opciones de la actividad.
- Método **onNavigationItemSelected**. Este método se encarga de manejar el evento cuando un ítem del menú de navegación lateral de la aplicación es presionado.
- Método **onBackPressed**. Este método se encarga de gestionar cuando un usuario ha presionado el botón "atrás", típico de los dispositivos Android.
- Método **onDestroy**. Este método es lanzado cuando la actividad va a ser destruida dentro de la aplicación.

A continuación, se detalla, para cada vista descubierta en el *Modelo de diseño*, los cambios realizados sobre ellas debido a la implementación (modificación de métodos diseñados previamente o explicación de métodos nuevos que no aparecieron en el diseño).

4.4.1 Clase VistaRecordarPassword

Los cambios de la clase *VistaRecordarPassword* son los siguientes:

- Método **mostrarDialogo**. Método nuevo, que muestra un diálogo, con un mensaje que expresa el resultado del envío del correo electrónico específico, para restablecer la contraseña del usuario. Tiene como parámetro "resultado" de tipo *boolean* y en función de su valor, mostrará un mensaje de que ha habido éxito en el envío (valor *true*) o que hubo algún problema (valor *false*).

4.4.2 Clase VistaLogin

Los cambios de la clase *VistaLogin* son los siguientes:

- Método **mostrarDialogoError**. Método nuevo, que muestra un mensaje al usuario, para indicarle que ha habido algún problema con el inicio de sesión en la aplicación y, por tanto, que no se pudo llevar a cabo.

4.4.3 Clase VistaInicio

Los cambios de la clase *VistaInicio* son los siguientes:

- Método **setInfoInicial**. Método nuevo, que muestra por pantalla, la información necesaria a presentar en los diferentes componentes de la vista, como por ejemplo, en el menú de navegación, donde va el logotipo de la clínica y el nombre del fisioterapeuta que ha iniciado sesión, además del nombre de la clínica que se inserta en la barra cabecera de la aplicación en esta vista. Esta información llega a través de los parámetros "nombreFisio" y "nombreClinica" de tipo *String*, y "imagen" de tipo *array de bytes*.
- Método **setListaEventos**. Se actualiza el parámetro "listaEventos", que previamente era de tipo *array de objetos de la clase Evento* y pasa a utilizarse un tipo *ArrayList de objetos de la clase Evento*.

4.4.4 Clase VistaPacientes

Los cambios de la clase *VistaPacientes* son los siguientes:

- Método **setInfoInicial**. Método nuevo, que muestra por pantalla, la información necesaria a presentar en los diferentes componentes de la vista, en concreto, en el menú de navegación, donde va el logotipo de la clínica y el nombre del fisioterapeuta que ha iniciado sesión. Esta información llega a través de los parámetros "nombreFisio" de tipo *String* y "imagen" de tipo *array de bytes*.
- Método **setListaPacientes**. Se actualiza el parámetro "listaPacientes", que previamente era de tipo *array de objetos de la clase Paciente* y pasa a utilizarse un tipo *ArrayList de objetos de la clase Paciente*.

4.4.5 Clase VistaAgregarEditarPaciente

Los cambios de la clase *VistaAgregarEditarPaciente* son los siguientes:

- Método **prepararVista**. Método nuevo, que prepara la vista, en función de si le llega un paciente desde el presentador o no. Así, si llega un paciente significa que la vista debe ser para editar el paciente; en cambio, si no llega ningún paciente, debe prepararse para agregar uno nuevo.
- Método **mostrarDialogo**. Método nuevo, que se encarga de mostrar al usuario, un diálogo informativo. El mensaje a mostrar va en función de la razón que se indique por parámetros, por ejemplo, un error a la hora de agregar o editar un paciente.

4.4.6 Clase VistaFichaPaciente

Los cambios de la clase *VistaFichaPaciente* son los siguientes:

- Método **setImagenesPaciente**. Se actualiza el parámetro "imagenes" de tipo *array de objetos de la clase Imagen* por el parámetro "listaImagenes" de tipo *ArrayList de objetos de la clase Bitmap*, consiguiendo así comunicar menos información de la necesaria, ya que únicamente es necesario para la vista las imágenes como tal (objetos *Bitmap*) para mostrarlos al usuario en la grilla de imágenes.
- Método **setEventosProximos**. Se actualiza el parámetro "eventosProximos" de tipo *array de objetos de la clase Evento*, por el parámetro "listaEventosProximos" de tipo *ArrayList de objetos de la clase Evento*.

- Método **setNotasPaciente**. Se actualiza el parámetro "notas" de tipo *array de objetos de la clase Nota*, por el parámetro "listaNotas" de tipo *ArrayList de objetos de la clase Nota*.

- Método **mostrarDialogoEliminarNota**. Se añade un parámetro denominado "posicion" de tipo *int*, que corresponde con la posición de la nota dentro de la lista de notas mostradas, y que se desea eliminar.

- Método **mostrarDialogo**. Método nuevo, que se encarga de mostrar al usuario, un diálogo informativo. El mensaje a mostrar va en función de la razón por la que se decide mostrarlo, por ejemplo, un error a la hora de eliminar una nota o un paciente. Este motivo se indica a través del parámetro *razon* de tipo *array de Object*.

4.4.7 Clase VistaAgregarImagen

Los cambios de la clase *VistaAgregarImagen* son los siguientes:

- Método **mostrarDialogoResultado**. Método nuevo, que muestra un diálogo al usuario, tras haberse realizado la operación de creación de una nueva imagen. En este diálogo se le indica el resultado de la tarea realizada, que se conoce a través del parámetro que le llega al método, que es "resultado" de tipo lógico.

4.4.8 Clase VistaDetalleImagen

Los cambios de la clase *VistaDetalleImagen* son los siguientes:

- Método **setImagen**. Inicialmente estaba previsto que este método tuviera como parámetro "imagen" de tipo *Imagen*. Finalmente, como se creó una nueva clase llamada *ContenedorImagen* (que almacena un objeto de la clase *Imagen*, que corresponde con el registro de la imagen en la base de datos, y un objeto *Bitmap*, que es la imagen propiamente dicha, obtenida del almacenamiento en la nube) se decide que el nuevo parámetro sea "contenedorImagen" de tipo *ContenedorImagen*.

- Método **mostrarDialogoError**. Método nuevo, que muestra un diálogo al usuario, informándole acerca de un error que se ha producido a la hora de intentar eliminar la imagen detallada.

4.4.9 Clase VistaAgregarEditarNota

Los cambios de la clase *VistaAgregarEditarNota* son los siguientes:

- Método **prepararVista**. Método nuevo, que se encarga de preparar la vista, en función de si le llega una nota desde el presentador o no. Así, si llega una nota significa que la vista debe ser para editar la nota; en cambio, si no llega ninguna nota, debe prepararse para agregar una nueva. Por ello, como parámetro, tiene "nota" de tipo *Nota*.
- Método **mostrarDialogo**. Método nuevo, que se encarga de mostrar al usuario, un diálogo informativo. El mensaje a mostrar va en función de la razón que se indique, por ejemplo, un error a la hora de agregar o editar un paciente. Por ello, para indicar la razón de mostrar este diálogo, se tiene como parámetro "razon" de tipo *array de Object*. El motivo de por qué es un *array* se debe a que almacena dos datos importantes: un objeto *String* que indica la operación realizada y otro objeto lógico, que indica el resultado de dicha operación.

4.4.10 Clase VistaCalendario

Los cambios de la clase *VistaCalendario* son los siguientes:

- Método **setInfoInicial**. Método nuevo, que muestra por pantalla, la información necesaria a establecer en los diferentes componentes de la vista, en concreto, en el menú de navegación, donde va el logotipo de la clínica y el nombre del fisioterapeuta que ha iniciado sesión. Esta información llega a través de los parámetros "nombreFisio" de tipo *String* y "imagen" de tipo *array de bytes*.

4.4.11 Clase VistaEventos

Los cambios de la clase *VistaEventos* son los siguientes:

- Método **setListaEventos**. Se sustituye el tipo del parámetro "listaEventos" previsto, pasando de *array de objetos de la clase Evento* a *ArrayList de objetos de la clase Evento*.
- Método **mostrarDialogoEliminar**. Se añade el parámetro "posicion" de tipo *int* que corresponde con la posición del evento dentro de la lista de eventos mostrada, y que se desea eliminar.

- Método **mostrarDialogoResultadoEliminar**. Método nuevo, que muestra un diálogo al usuario, indicándole el resultado exitoso o fallido de la eliminación del evento que ha elegido borrar de la lista de eventos. Este resultado es conocido por el método a partir del parámetro "resultado" de tipo *boolean*.
- Método **prepararVista**. Método nuevo, que se encarga de presentar ciertos elementos de la vista, como la barra cabecera, mostrando el día del que se pretende obtener la lista de eventos. Como parámetros cuenta con "info" de tipo *array de Object*.

4.4.12 Clase VistaAgregarEditarEvento

Los cambios de la clase *VistaAgregarEditarEvento* son los siguientes:

- Método **prepararVista**. Método nuevo, que se encarga de preparar la vista, en función de si le llega un evento desde el presentador o no. Así, si llega un evento significa que la vista debe ser para editar el evento. En cambio, si no llega ningún evento, debe prepararse para agregar uno nuevo. Por ello, este método tiene como parámetro "evento" de tipo *Evento*, además de otros parámetros como "nombrePaciente" y "dia" de tipo *String* en ambos casos. Estos dos últimos parámetros dependen de si se accede a esta actividad con este tipo de información complementaria a la hora de agregar un evento.
- Método **prepararSpinners**. Método nuevo, que se encarga de preparar los elementos desplegables del formulario, con los nombres de los pacientes y los fisioterapeutas que pertenecen a la clínica del fisioterapeuta que ha iniciado sesión. Entre sus parámetros se encuentran los siguientes: "nombresPacientes" y "nombresFisioterapeutas", ambos de tipo *ArrayList de String*.
- Método **mostrarDialogo**. Método nuevo, que se encarga de mostrar al usuario, un diálogo informativo. El mensaje a mostrar va en función de la razón, por ejemplo, un error a la hora de agregar o editar un evento. Esta razón se le indica al método a través del parámetro "resultado" de tipo lógico.

4.4.13 Clase VistaTareas

Los cambios de la clase *VistaTareas* son los siguientes:

- Método **setInfoInicial**. Método nuevo, que muestra por pantalla la información necesaria a presentrar en los diferentes componentes de la vista, en concreto, en el menú de navegación, donde va el logotipo de la clínica y el nombre del fisioterapeuta que ha iniciado sesión. Esta información llega a través de los parámetros "nombreFisio" de tipo *String* y "imagen" de tipo *array de bytes*.

- Método **mostrarDialogoError**. Método nuevo, que se encarga de mostrar por pantalla, un diálogo que indica al usuario el resultado fallido de una determinada operación, sea de adición o finalización de tareas. La operación realizada se hace llegar al método a través del parámetro "operacion" de tipo *String*.
- Método **setListaTareas**. Se actualiza el parámetro "listaTareas", que previamente era de tipo *array de objetos de la clase Tarea* y pasa a utilizarse un tipo *ArrayList de objetos de la clase Tarea*.
- Método **mostrarDialogoFinalizar**. Se añade el parámetro denominado "posicion" de tipo *int*, que corresponde con la posición de la tarea dentro de la lista de tareas, la cual se desea dar por finalizada.

4.4.14 Clase VistaPagos

Los cambios de la clase *VistaPagos* son los siguientes:

- Método **setInfoInicial**. Método nuevo, que muestra por pantalla, la información necesaria a establecer en los diferentes componentes de la vista, en concreto, en el menú de navegación, donde va el logotipo de la clínica y el nombre del fisioterapeuta que ha iniciado sesión. Esta información llega a través de los parámetros "nombreFisio" de tipo *String* y "imagen" de tipo *array de bytes*.
- Método **mostrarDialogoError**. Método nuevo, que muestra un diálogo al usuario, informándole acerca de un error que se ha producido en alguna operación a realizar, sea de añadir un nuevo pago o de dar por realizado algún pago pendiente. La operación realizada se hace llegar al método por parámetros ("operacion" de tipo *String*).
- Método **setListaPagos**. Se modifica el tipo del parámetro "listaPagos" a tipo *ArrayList de objetos de la clase Pago* y, además, se añade un nuevo parámetro: "realizados" de tipo lógico, que indica el tipo de pago (REALIZADO, en caso de tomar valor *true* o PENDIENTE, en caso de tomar valor *false*) que contiene la lista que se hace llegar por parámetros.
- Método **mostrarDialogoAgregar**. Se añade un parámetro denominado "pacientes" de tipo *ArrayList de String*. Esto es debido a que el diálogo que se pretende mostrar necesita rellenar un elemento desplegable que contiene el nombre de los pacientes de la clínica, y de esta manera se le hace llegar la información al diálogo para que lo prepare debidamente.
- Método **mostrarDialogoRealizado**. Se añade un parámetro a este método denominado "posicion" de tipo *int*. Corresponde con la posición del pago sobre el que se ha presionado, en la lista de pagos pendientes mostrada al usuario. Este dato será necesario posteriormente para dar por realizado dicho pago si el usuario desea continuar con la operación.

4.4.15 Clase VistaUsuario

Los cambios de la clase *VistaUsuario* son los siguientes:

- Método **setInfoInicial**. Método nuevo, que muestra por pantalla, la información necesaria a presentar en los diferentes componentes de la vista, en concreto, en el menú de navegación, donde va el logotipo de la clínica y el nombre del fisioterapeuta que ha iniciado sesión. Esta información llega a través de los parámetros "nombreFisio" de tipo *String* y "imagen" de tipo *array de bytes*.
- Método **mostrarDialogoResultado**. Método nuevo, que muestra un diálogo, que expresa el resultado de la operación que ha solicitado el usuario. Cuenta como parámetro el siguiente: "info" de tipo *array de Object*. Este array contiene información necesaria para el diálogo, tal como la operación realizada, el resultado de la misma, y el error en caso de existir.

4.5 Paquete presentador

En este apartado se explican los cambios y adaptaciones que se han tenido que realizar en las clases e interfaces del paquete **presentador**, con respecto a las definidas en el *Modelo de Diseño* (modificación de métodos diseñados previamente o explicación de métodos nuevos que no aparecieron en el diseño). Los cambios realizados son los siguientes:

4.5.1 Clase PresentadorRecordarPassword

Los cambios de la clase *PresentadorRecordarPassword* son los siguientes:

- Método **tratarDialogo**. Método nuevo, que notifica al presentador, que el botón "Aceptar", del diálogo que se muestra tras haber tenido éxito en la obtención del correo electrónico para restablecer la contraseña, ha sido presionado. Se encargará finalizar la actividad.

4.5.2 Clase PresentadorInicio

Los cambios de la clase *PresentadorInicio* son los siguientes:

- Método **tratarItemDesconectar**. Método nuevo, que se encarga de hacer que el presentador solicite la finalización de la sesión actual del usuario que se encuentra utilizando la aplicación.

- Método **obtenerInfoInicial**. Método nuevo, que se comunica con el modelo, para obtener la información inicial que se necesitan en determinados componentes de la vista (y que ésta pueda presentarlos). En este caso, se obtiene el nombre del fisioterapeuta y el logotipo de la clínica para mostrarlos en el cabecero del menú lateral de navegación, además del nombre de la clínica para mostrarlo en la barra cabecera.

4.5.3 Clase PresentadorPacientes

Los cambios de la clase *PresentadorPacientes* son los siguientes:

- Método **tratarPaciente**. Se actualiza el parámetro anterior "idPaciente" de tipo *String* a "posicion" de tipo *int*. Este valor corresponde con la posición del paciente seleccionado en la lista de pacientes de la vista. Esta posición será enviada a la siguiente actividad, que se encargará de obtener el paciente en cuestión con la información necesaria del mismo.

- Método **obtenerInfoInicial**. Método nuevo, que se comunica con el modelo, para obtener la información inicial que se necesitan en determinados componentes de la vista (y que ésta pueda presentarlos). En este caso, se obtiene el nombre del fisioterapeuta y el logotipo de la clínica para mostrarlos en el cabecero del menú lateral de navegación.

- Método **tratarItemDesconectar**. Método nuevo, que se encarga de hacer que el presentador solicite la finalización de la sesión actual del usuario que se encuentra utilizando la aplicación.

4.5.4 Clase PresentadorAgregarEditarPaciente

Los cambios de la clase *PresentadorAgregarEditarPaciente* son los siguientes:

- Método **tratarBotonFinalizar**. Se sustituye el parámetro "paciente" de tipo *Paciente*, por una serie de parámetros, que corresponden con la información de un paciente y que ha sido insertada por el usuario en la vista: "idPaciente", "nombre", "correo", "telefono" y "historial", todos de tipo *String*.

- Método **obtenerPaciente**. Método nuevo, que se encarga de comprobar si se ha accedido a esta pantalla, con el objetivo de agregar un nuevo paciente o para editar uno ya existente, comprobando si ha llegado algún paciente a esta vista desde otra. En caso de que llegue, se le entrega a la vista indicándole que la actividad es para edición, y en caso contrario, es decir, que no llegue nada, se le indica que es para agregar un nuevo paciente. Como parámetro tiene "datos" de tipo *Object*, que corresponderá con la información, en caso de llegar, de la actividad anterior.
- Método **tratarFinalizarActividad**. Método nuevo, que se encarga de dar por finalizada la actividad, tras haber terminado de editar o agregar un paciente y se desea volver a la vista anterior.

4.5.5 Clase PresentadorFichaPaciente

Los cambios de la clase *PresentadorFichaPaciente* son los siguientes:

- Método **obtenerInfoPaciente**. Se actualiza el parámetro "idPaciente" de tipo *String* a un nuevo parámetro denominado "datos" de tipo *Object*, para dar flexibilidad al método, en cuanto a la información que le llega a través de este parámetro.
- Método **tratarBotonEditar**. Se elimina el parámetro "idPaciente" de tipo *String*, ya que esta información no es necesaria, al ya contar en la clase con la variable "pacienteCargado" de tipo *Paciente*, que será la información que se envíe a la otra actividad.
- Método **tratarDialogoEliminar**. Se elimina el parámetro "idPaciente" de tipo *String*, ya que esta información no es necesaria, ya que se puede obtener este identificador de paciente a través de la variable "pacienteCargado" de tipo *Paciente*, que se encuentra guardada en la clase.
- Método **tratarBotonAgregarImagen**. Se elimina el parámetro "idPaciente" de tipo *String*, ya que esta información no es necesaria, ya que se puede obtener este identificador de paciente a través de la variable "pacienteCargado" de tipo *Paciente*, que se encuentra guardada en la clase.
- Método **tratarImagen**. Se sustituye el parámetro "idImagen" de tipo *String* por un parámetro "posicion" de tipo *entero*, que corresponde con la posición de la imagen en el grid de imágenes mostrado al usuario. Esta posición permitirá obtener la imagen concreta que ha presionado el usuario, y por tanto se podrá obtener esa imagen desde la lista y enviarla a la siguiente actividad.
- Método **tratarBotonAgregarNota**. Se elimina el parámetro "idPaciente" de tipo *String*, ya que esta información no es necesaria, ya que se puede obtener este identificador de paciente a través de la variable "pacienteCargado" de tipo *Paciente*, que se encuentra guardada en la clase.

- Método **tratarBotonEditarNota**. Se añade el parámetro "posicion" de tipo *int*, que corresponde con la posición de la nota, dentro de la lista de notas mostrada. Este valor permitirá obtener la nota indicada por el usuario, pudiendo así comunicarla a la siguiente actividad.
- Método **tratarBotonEliminarNota**. Se añade el parámetro "posicion" de tipo *int*, que corresponde con la posición de la nota, dentro de la lista de notas mostrada, la cual el usuario desea eliminar.
- Método **tratarDialogoEliminarNota**. Se sustituye el parámetro "idNota" de tipo *String* por un parámetro "posicion" de tipo *int*, que corresponde con la posición de la nota en la lista de notas mostrada al usuario. Esta posición permitirá obtener la nota concreta que el usuario desea eliminar, para posteriormente continuar dicha eliminación a través de la comunicación específica al modelo de la aplicación.
- Método **tratarFinalizarActividad**. Método nuevo, que se encarga de dar por finalizada la actividad, tras haber terminado de eliminar un paciente, y se ha de volver a la actividad anterior, es decir, la vista de la lista de pacientes.

4.5.6 Clase PresentadorAgregarImagen

Los cambios de la clase *PresentadorAgregarImagen* son los siguientes:

- Método **tratarBotonFinalizar**. Se sustituye el parámetro original por "imagen" de tipo *Bitmap* y "descripcion" de tipo *String*. Esta es la información que inserta el usuario en la vista, por tanto, se delega en el modelo para que haga lo necesario con dicha información para finalmente agregar la nueva imagen.
- Método **obtenerPaciente**. Método nuevo que recibe, por parte de la vista, el identificador del paciente para el que se pretende agregar la imagen. De esta manera sera posible identificar a qué paciente se esta añadiendo la imagen a la hora de realizar el proceso de subida de la imagen. Este identificador llega a través del parámetro "paciente" de tipo *Object*.
- Método **tratarFinalizarActividad**. Método nuevo que se encarga de finalizar la actividad, tras haber realizada la adición de la nueva imagen, y haber visualizado y tratado el diálogo con el resultado positivo de dicha operación.

4.5.7 Clase PresentadoDetalleImagen

Los cambios de la clase *PresentadorDetalleImagen* son los siguientes:

- Método **obtenerImagen**. Se sustituye el parámetro "idImagen" de tipo *String*, por el parámetro "datos" de tipo *array de Object*. Este *array* almacena la posición de la imagen que ha sido seleccionada, dentro de la lista de imágenes cargada, además del identificador del paciente para el que se ha mostrado la ficha de paciente previamente. Este último dato será útil en caso de querer eliminar la imagen posteriormente.

4.5.8 Clase PresentadorAgregarEditarNota

Los cambios de la clase *PresentadorAgregarEditarNota* son los siguientes:

- Método **tratarBotonFinalizar**. Se sustituye el parámetro por una serie de parámetros, que coinciden con la información que inserta el usuario: "idNota", "titulo" y "cuerpo", todos de tipo *String*.
- Método **obtenerNota**. Método nuevo que se encarga de comprobar si se ha accedido a esta pantalla con el objetivo de agregar una nueva nota o para editar una ya existente. En caso de que llegue una nota, se le entrega a la vista, indicándole que la actividad es para edición, y en caso contrario, es decir, que no llegue nada, se le indica que es para agregar una nueva nota. Para comprobar si llega esta nota o no, existe el parámetro "datos" de tipo *Object*, que lleva consigo la información de la actividad anterior.
- Método **tratarFinalizarActividad**. Método nuevo que se encarga de dar por finalizada la actividad tras haber terminado de editar o agregar una nota y se desea volver a la vista anterior.

4.5.9 Clase PresentadorCalendario

Los cambios de la clase *PresentadorCalendario* son los siguientes:

- Método **tratarDiaCalendario**. Se actualiza el parámetro anterior "fecha" de tipo *Object* a un nuevo parámetro "datos" de tipo *array de Object*. Esto es debido a que se va a utilizar este *array* como un contenedor de dos datos necesarios para presentar la información: la fecha seleccionada por el fisioterapeuta y el nombre del mismo. Esta última información será necesaria para mostrar los eventos correspondientes al fisioterapeuta que utiliza la aplicación, en otro color.
- Método **tratarItemDesconectar**. Método nuevo que se encarga de hacer que el presentador solicite la finalización de la sesión actual del usuario que se encuentra utilizando la aplicación.

- Método **obtenerInfoInicial**. Método nuevo que se comunica con el modelo para obtener la información inicial que se necesitan en determinados componentes de la vista. En este caso, se obtiene el nombre del fisioterapeuta y el logotipo de la clínica para mostrarlos en el cabecero del menú lateral de navegación.

4.5.10 Clase PresentadorEventos

Los cambios de la clase *PresentadorEventos* son los siguientes:

- Método **prepararVista**. Método nuevo que se encarga de obtener la información que es enviada desde la vista anterior y que es necesaria para el funcionamiento de la vista de eventos.
- Método **tratarDialogoResultadoEliminar**. Método nuevo que encarga de tratar la interacción del usuario con el diálogo que indica el resultado de la eliminación. En concreto, este método solo funcionará cuando el diálogo sea exitoso y el usuario presione "Aceptar", ya que se encargará de actualizar la lista de eventos que se le está mostrando, ya con el evento eliminado.
- Método **obtenerListaEventos**. Se actualiza el parámetro "dia" de tipo *Object* a tipo *String*, ya que será el tipo concreto de datos con el que se trabajará las fechas en la aplicación.
- Método **tratarBotonEditar, tratarBotonEliminar, tratarDialogoEliminar**. Estos métodos se ha decidido que tengan como único parámetro "posicion" de tipo *int*. Este parámetro corresponderá con la posición del evento dentro de la lista mostrada al usuario, pudiendo así identificar con qué evento se desea trabajar, sea para edición o para eliminación, dependiendo del método.

4.5.11 Clase PresentadorAgregarEditarEvento

Los cambios de la clase *PresentadorAgregarEditarEvento* son los siguientes:

- Método **tratarBotonFinalizar**. Se sustituye el parámetro por los siguientes (corresponden con cada uno de los campos que inserta el usuario en el formulario de la vista): "idEvento", "fecha", "horaComienzo", "horaFinalizacion", "tipo", "descripcion", "nombrePaciente" y "nombreFisioterapeuta", todos de tipo *String*.

- Método **obtenerEvento**. Método nuevo que se encarga de comprobar si se ha accedido a esta actividad con el objetivo de agregar un nuevo evento o para editar uno ya existente, comprobando si ha llegado algún evento a la vista desde otra. En caso de que llegue, se le entrega a la vista indicándole que la actividad es para edición, y en caso contrario, es decir, que no llegue nada, se le indica que es para agregar un nuevo evento. Como parámetro tiene "datos" de tipo *array de Object*, que puede contener, además del posible evento, información complementaria como el nombre de paciente (en caso de acceder desde la clase *VistaFichaPaciente*) o el día (en caso de acceder desde la clase *VistaEventos*), en caso de querer añadir un evento.

- Método **obtenerInfoSpinners**. Método nuevo que se encarga de obtener la información necesaria para rellenar los elementos desplegables de la vista, es decir, se comunica con el modelo para obtener los nombres de los pacientes y los fisioterapeutas.

- Método **tratarFinalizarActividad**. Método nuevo que se encarga de dar por finalizada la actividad, tras haber terminado de editar o agregar un nuevo evento y se desea volver a la vista anterior.

4.5.12 Clase PresentadorTareas

Los cambios de la clase *PresentadorTareas* son los siguientes:

- Método **tratarItemDesconectar**. Método nuevo que se encarga de hacer que el presentador solicite la finalización de la sesión actual del usuario que se encuentra utilizando la aplicación.

- Método **obtenerInfoInicial**. Método nuevo que se comunica con el modelo para obtener la información inicial que se necesitan en determinados componentes de la vista. En este caso, se obtiene el nombre del fisioterapeuta y el logotipo de la clínica para mostrarlos en el cabecero del menú lateral de navegación.

- Método **obtenerListaTareas**. Se elimina el parámetro "idClinica" de tipo *String*, que correspondía con el identificador único de la clínica en la que se basa la aplicación. Esta decisión se toma porque este dato se puede obtener a partir del atributo "fisioterapeutaActual" guardado en la clase *Modelo* tras haber realizado el inicio de sesión en la aplicación.

- Método **tratarBotonFinalizar**. Se añade el parámetro "posicion" de tipo *int*, que corresponde con la posición de la tarea a finalizar dentro de la lista de tareas mostradas al usuario.

- Método **tratarDialogoFinalizar**. Se cambia el parámetro "idTarea" de tipo *String* a un parámetro llamado "posicion" de tipo *int*, correspondiente a la posición de la tarea que se desea finalizar dentro de la lista de tareas que se muestra al usuario.

- Método **tratarDialogoAgregar**. Se sustituye el antiguo parámetro por el nuevo denominado "titulo" de tipo *String*, que es la información necesaria para la creación de una nueva tarea.

4.5.13 Clase PresentadorPagos

Los cambios de la clase *PresentadorPagos* son los siguientes:

- Método **tratarItemDesconectar**. Método nuevo que se encarga de hacer que el presentador solicite la finalización de la sesión actual del usuario que se encuentra utilizando la aplicación.

- Método **obtenerInfoInicial**. Método nuevo que se comunica con el modelo para obtener la información inicial que se necesitan en determinados componentes de la vista. En este caso, se obtiene el nombre del fisioterapeuta y el logotipo de la clínica para mostrarlos en el cabecero del menú lateral de navegación.

- Métodos **obtenerListaRealizados, obtenerListaPendientes**. En ambos métodos se elimina el parámetro "idClinica" de tipo *String*, ya que esta información (el identificador único de la clínica del fisioterapeuta que ha iniciado sesión) se encuentra en el objeto "fisioterapeutaActual" almacenado en la clase *Modelo* tras haber iniciado sesión en la aplicación.

- Método **tratarDialogoAgregar**. Se sustituye el parámetro inicial por los siguientes: "fecha" de tipo *String*, "montante" de tipo *double*, "realizado" de tipo *boolean* y "nombrePaciente" de tipo *String*.

- Método **tratarBotonRealizado**. Se añade un parámetro llamado "posicion" de tipo *int*, que corresponde con la posición del pago a dar por realizado, dentro de la lista de pagos pendientes.

- Método **tratarDialogoRealizado**. Se sustituye el parámetro que estaba previsto, "idPago" de tipo *String*, por un parámetro "posicion" de tipo *int*. Este parámetro corresponde con la posición del pago que se pretende dar por realizado, y que es de tipo pendiente.

4.5.14 Clase PresentadorUsuario

Los cambios de la clase *PresentadorUsuario* son los siguientes:

- Método **tratarItemDesconectar**. Método nuevo que se encarga de hacer que el presentador solicite la finalización de la sesión actual del usuario que se encuentra utilizando la aplicación.
- Método **obtenerInfoInicial**. Método nuevo que se comunica con el modelo para obtener la información inicial que se necesitan en determinados componentes de la vista. En este caso, se obtiene el nombre del fisioterapeuta y el logotipo de la clínica para mostrarlos en el cabecero del menú lateral de navegación.
- Método **tratarDialogoEditar**. Se sustituye el parámetro inicial por una serie de parámetros que coinciden con cada uno de los campos a modificar de un usuario (a excepción de la contraseña, que es necesaria para realizar una reautenticación del usuario): "password", "nombre", "correo" y "telefono", todos de tipo *String*.
- Método **tratarDialogoCambiarPassword**. Se sustituyen los parámetros que estaban previstos, es decir, "idFisioterapeuta" de tipo *String* y "password" de tipo *String*), por los siguientes: "passwordAntigua" y "passwordNueva", ambos de tipo *String*.
- Método **tratarDialogoDarBaja**. Inicialmente estaba previsto que este método contara con el parámetro "idFisioterapeuta" de tipo *String*, pero finalmente no será necesario al contar con esta información en el objeto "fisioterapeutaActual" que se encuentra en la clase *fisioterapeutaActual*, y representa al fisioterapeuta que ha iniciado sesión. Por tanto, este parámetro será eliminado y se añadirá uno nuevo, "password" de tipo *String*, que corresponde con la contraseña del usuario, necesaria por seguridad, garantizando así que realmente es el fisioterapeuta dueño de su cuenta de usuario el que está dando de baja su cuenta de usuario.

Capítulo **5**

MODELO DE PRUEBAS

5.1 Test de usabilidad de la interfaz de usuario

El propósito del test de usabilidad es saber el grado de dificultad que puede tener la aplicación ante los usuarios a los que va destinado y saber si cumple con su finalidad. Con el test se comprueba su efectividad, eficiencia y grado de satisfacción. El test sólo será un punto de vista de cómo será y cómo podrán interactuar los usuarios con la aplicación mediante su interfaz. La realización de este test de usabilidad permitirá detectar los puntos que podrían ser más críticos del producto y así poder modificarlo y mejorarlo. Para ello, es necesario ver que el usuario se siente cómodo con la aplicación, que sepa qué hacer en todo momento y que le parezca una herramienta útil.

En concreto, la aplicación presentada en esta memoria, está destinada a personas que trabajen en el sector de la fisioterapia, por lo que el test se va a realizar a personas con ciertos conocimientos del sector y a otras no tan expertas, pero con ciertas nociones de lo que tienen que realizar. Algo que sí se requerirá es que sean usuarios habituales de aplicaciones móviles.

5.1.1 Explicación del producto

La aplicación Android para la gestión de una clínica de fisioterapia, *FisioClinicApp*, que se plantea en este Trabajo Fin de Grado, conseguirá facilitar las tareas de gestión que se realizan de manera cotidiana dentro de una clínica de fisioterapia, pudiendo gestionar fichas de los pacientes, eventos, tareas a realizar y pagos. Las principales características que tendrá la aplicación son:

- Diseño claro y simple.
- Permite la consulta de las fichas de cada paciente, donde se tiene información personal de los mismos, un archivo de imágenes específico y un registro de notas.
- Posibilidad de gestionar los eventos de la clínica mediante el uso de un calendario.

- Permite el manejo de una lista de tareas a realizar por y para los empleados de la clínica.
- Posibilita el registro de los pagos de los clientes, tanto realizados como pendientes.
- Existencia de una cuenta de usuario personal para cada empleado de la clínica.

5.1.2 Objetivos del producto

La aplicación Android para la gestión de una clínica de fisioterapia, *FisioClinicApp*, tiene como objetivo ser una herramienta útil y fiable para todas las personas que trabajen en el sector fisioterapéutico, en concreto para aquellas que forman parte de una clínica de fisioterapia. Con esta aplicación, los usuarios podrán unificar en una aplicación todas esas tareas de gestión para las que antes tenían que utilizar diversas aplicaciones, contando con la información las 24 horas del día con total movilidad, sea dentro de la clínica o fuera de ella. En definitiva, será una aplicación que pretende ser imprescindible en el día a día de un profesional de una clínica de este tipo, ya que estas ventajas que aporta permiten una mejora en cuanto a efectividad y productividad en su trabajo, y todo ello, con la comodidad que da un dispositivo móvil.

5.1.3 Usuarios y participantes

Esta aplicación está destinada a profesionales del sector de la fisioterapia, en concreto, a aquellos que forman parte de la plantilla de empleados dentro de una clínica de fisioterapia. Por tanto, serán ellos los usuarios primarios de la aplicación (único tipo de usuarios posibles. La aplicación no cuenta con usuarios secundarios, ni terciarios). El tipo de usuario que realizará el test de usabilidad debe tener, al menos, conocimientos previos sobre el funcionamiento de una clínica de este tipo, además de una cierta familiaridad con el uso de aplicaciones móviles.

5.1.4 Usuarios finales del producto

El perfil de usuario final a quien va dirigido este producto es una persona de cualquier género y que esté entre los 18 y 65 años, es decir, en edad con posibilidad de trabajar y que haya pertenecido o pertenezca a una clínica de fisioterapia.

5.1.4.1. Criterios de selección del grupo de test

Se seguirán los siguientes criterios para seleccionar los usuarios que realizarán el test:

- Persona de lengua castellana con una edad comprendida entre los 18 y 65 años,
- que, además, esté graduado o licenciado en Fisioterapia por una universidad o estén en proceso de ello y se encuentren en fase de prácticas dentro de una clínica de fisioterapia y
- que esté habituada al uso de teléfonos inteligentes (*smartphones*) o tabletas.

5.1.5 Diseño del proceso de test

El test de usabilidad se realizará mediante una demo. A continuación se establece: las características de la evaluación del prototipo, las herramientas necesarias para dicha evaluación, las tareas que debe realizar el usuario, los indicadores de éxito, las respuestas del usuario a esos indicadores y las observaciones realizadas por los usuarios sobre el producto.

El resultado del test se obtendrá observando las dificultades que se encuentren los usuarios a la hora de utilizar la demo y el cuestionario realizado por éstos, según el test de usabilidad planteado. El éxito o fracaso de la demo dependerá de la opinión de los usuarios.

5.1.5.1. Logística

La evaluación se realizará en un lugar tranquilo y cómodo para el usuario, libre de distracciones e interrupciones durante el periodo del test, el cual tendrá una duración aproximada de 30 minutos. Antes de comenzar a realizar el test, al usuario se le dará una breve descripción sobre el propósito de la tarea para poder testear la demo adecuadamente. Durante la evaluación, se entregará al usuario un cuestionario de usabilidad para obtener las evidencias del mismo. En dicho cuestionario se recopilarán datos de usabilidad, utilidad y grado de satisfacción de la aplicación.

5.1.5.2. Instrumentación

Las herramientas a utilizar serán: un ordenador personal con conexión a Internet o, en su defecto, una tableta de diez pulgadas para ejecutar la demo y un prototipo de la aplicación. En caso de no tener conexión a Internet, se requerirá que tenga instalada alguna aplicación software para abrir documentos tipo PDF.

Para acceder al prototipo a través de la red, desde el navegador se lanzará la herramienta de prototipado Balsamiq Mockups, con la que el usuario realizará el test en modo presentación. En el caso de no disponer de conexión a Internet, se presentará el prototipo en la versión PDF.

Además de las herramientas indicadas, el usuario recibirá un documento en papel con el cuestionario a responder.

5.1.5.3. Tareas a realizar

El usuario debe realizar sin ayuda cada una de las tareas que se presentan a continuación (para la correcta realización de las tareas, el usuario debe seguir el orden de las acciones asociadas a cada una de ellas):

- **Tarea 1:** Entrar en la aplicación *FisioClinicApp*.
  Acciones:
  1. Abrir el documento "FisioClinicAppTU.pdf" y, por tanto, la aplicación *FisioClinicApp*
  2. Iniciar sesión. Se supone que ya se dispone de una cuenta de usuario y que la introducción de los datos "Correo electrónico" y "Contraseña" son correctos.
- **Tarea 2:** Añadir un nuevo paciente.
  Acciones:
  1. Acudir a la lista de pacientes de la aplicación.
  2. Añadir un nuevo paciente a la lista de pacientes de la clínica, con todos los campos habilitados.
  3. Guardar el nuevo paciente, sin añadir una cita nueva para el mismo.
- **Tarea 3:** Editar información del paciente Carlos Martínez.
  Acciones:
  1. Acceder al paciente Carlos Martínez.
  2. Editar su información. En concreto: foto de perfil, teléfono y correo electrónico.

3. Guardar la edición del paciente.

- **Tarea 4:** Eliminar imagen del archivo personal de Carlos Martínez.
  Acciones:

  1. Acceder al archivo de imágenes del paciente Carlos Martínez.
  2. Borrar primera imagen.

- **Tarea 5:** Añadir una nueva imagen al archivo personal de Carlos Martínez
  Acciones:

  1. Añadir una nueva imagen al archivo personal de Carlos Martínez
  2. Guardarla, habiendo rellenado también el campo Descripción de la imagen.

- **Tarea 6:** Añadir una nueva nota al archivo personal de Carlos Martínez.
  Acciones:

  1. Acceder al archivo de notas del paciente Carlos Martínez.
  2. Crear una nueva nota, rellenando los campos solicitados.
  3. Guardar dicha nota.

- **Tarea 7:** Editar nota "Sesión 15/01/2017" de Carlos Martínez.
  Acciones:

  1. Editar la nota "Sesión 15/01/2017" de Carlos Martínez. En concreto, el campo "Cuerpo de la nota".
  2. Guardar la edición realizada.

- **Tarea 8:** Eliminar el paciente Carlos Martínez.
  Acciones:

  1. Eliminar el paciente Carlos Martínez del registro de pacientes de la clínica.
  2. Comprobar en la lista de pacientes que se ha eliminado este paciente.

- **Tarea 9:** Añadir un nuevo evento.
  Acciones:

  1. Ir al calendario de la aplicación.
  2. Añadir un nuevo evento, con la siguiente información: La fecha debe ser el 9 de julio. La hora de comienzo es 12:00 y la de finalización las 12:45. El tipo de evento debe ser "Cita", la descripción del evento debe ser "Esguince de tobillo", asociada al fisioterapeuta Juan Castro y al paciente Carmen Nieves Ríos.
  3. Una vez rellenados todos los campos, guardar el evento.
  4. Comprobar que ha sido creado correctamente.

- **Tarea 10:** Editar el evento "Cita" de Miguel Ángel Pérez.
  Acciones:

  1. Editar el evento Cita de Miguel Ángel Pérez de ese día. En concreto, el campo "Descripción del evento".
  2. Guardar la modificación realizada.

- **Tarea 11:** Añadir una nueva tarea.
  Acciones:

  1. Ir a la lista de tareas de la aplicación.
  2. Añadir una nueva tarea llamada "Regar las plantas".

- **Tarea 12:** Dar por finalizada la tarea "Comprar agujas".
  Acciones:

  1. Buscar la tarea "Comprar agujas" y darla por finalizada.

- **Tarea 13:** Eliminar la tarea "Comprar agujas".
  Acciones:

  1. Buscar la tarea "Comprar agujas" y eliminarla de la lista.

- **Tarea 14:** Añadir un nuevo pago realizado.
  Acciones:

  1. Ir a la lista de pagos de la aplicación.
  2. Añadir un nuevo pago, con la siguiente información: Fecha "9 de junio de 2017", paciente "Laura Rodríguez" y con un montante de 25 euros.

- **Tarea 15:** Dar por realizado el pago pendiente de "Lorena Pérez".
  Acciones:

  1. Acudir a la lista de pagos pendientes.
  2. Buscar el pago pendiente de Lorena Pérez y darlo por realizado.
  3. Revisar la lista de pagos realizados posteriormente.

- **Tarea 16:** Cambiar contraseña del usuario de la aplicación.
  Acciones:

  1. Seleccionar el ítem dedicado al usuario de la aplicación.
  2. Cambiar contraseña del mismo.

5.1.5.4. Indicadores de éxito

Los indicadores que avalan el éxito o fracaso de las distintas tareas se presentan a continuación:

- **Tarea 1:** Inicia la aplicación y realiza el *login*, terminando en la pantalla *Inicio* de la aplicación.
- **Tarea 2:** Accede al ítem *Pacientes* y añade un nuevo paciente, que aparece posteriormente en la lista de pacientes.
- **Tarea 3:** Accede al paciente Carlos Martínez, edita la información solicitada y se comprueba dicha edición en la ficha del paciente.
- **Tarea 4:** Accede al ítem "Imágenes" del paciente Carlos Martínez y elimina la imagen solicitada. Al volver al archivo de imágenes se comprueba que ha sido eliminada.
- **Tarea 5:** Accede a la pantalla habilitada para añadir nuevas imágenes y finalmente guarda la nueva imagen correctamente, observando que, finalmente, está esté en el archivo de imágenes.
- **Tarea 6:** Accede al ítem "Notas" del paciente Carlos Martínez y añade una nota a través de la pantalla dedicada a ello.
- **Tarea 7:** Accede a la pantalla de edición de la nota solicitada a través del menú contextual de la nota y es capaz de terminar dicha edición.
- **Tarea 8:** Es capaz de eliminar el paciente acudiendo directamente al ítem "Ficha". Será importante que sepa reconocer que en esa pantalla se podrá eliminar el paciente, y que no navegue innecesariamente por la aplicación, buscando el botón específico.
- **Tarea 9:** Es capaz de salir del ítem *Pacientes* hacia el ítem *Calendario* y añadir un nuevo evento en esta pantalla.
- **Tarea 10:** Es capaz de reconocer el botón indicado para editar el evento solicitado y realiza la edición correctamente.
- **Tarea 11:** Accede al ítem *Tareas* saliendo sin problemas de la pantalla anterior y añade una nueva tarea a la lista.
- **Tarea 12:** Es capaz de reconocer el concepto "Dar por finalizada", diferenciándolo del "Eliminar", utilizando el botón concreto para ello.
- **Tarea 13:** Selecciona el botón habilitado para eliminar la tarea y, por tanto, es capaz de eliminar dicha tarea.
- **Tarea 14:** Accede al ítem *Pagos* y añade un nuevo pago realizado.

- **Tarea 15:** Es capaz de acceder a la pantalla dedicada a los pagos pendientes, y dar por realizado el pago pendiente solicitado. Posteriormente, debe observar si el pago está, por tanto, en la pantalla "Realizados".
- **Tarea 16:** Accede al ítem *Mi usuario* y reconoce el menú de la barra de estado para buscar la acción "Cambiar contraseña". Además, realiza el cambio de contraseña sin problemas.

5.1.5.5. Respuesta de los usuarios a los indicadores

Los usuarios, a la hora de realizar el test, son evaluados también en términos de tiempo de realización de cada tarea, permitiéndole realizar observaciones al terminar cada una de ellas. En la tabla 5.1 se muestran las duraciones de cada una de las tareas de las evaluaciones, y por otro lado en la sección 5.1.5.6 se enumeran las evidencias obtenidas según las respuestas de los usuarios que realizaron este test de usabilidad. Es importante mencionar que, si bien lo ideal es que el test de usabilidad lo realicen al menos 5 usuarios, al autor de este Trabajo Fin de Grado le ha resultado imposible conseguir que más de 2 usuarios hayan respondido al test. Aún teniendo este problema, el autor de este trabajo considera que es importante hacer este estudio y utilizar los comentarios que estos usuarios han hecho para mejorar el producto.

	Perfil 1	**Perfil 2**
Tarea1	0:05	0:06
Tarea2	1:44	1:23
Tarea3	1:09	0:51
Tarea4	0:22	0:24
Tarea5	0:41	0:36
Tarea6	0:33	0:31
Tarea7	0:58	0:34
Tarea8	0:13	0:24
Tarea9	2:41	1:46
Tarea10	1:46	1:37
Tarea11	0:27	0:30
Tarea12	0:07	0:11
Tarea13	0:03	0:04
Tarea14	0:33	0:29
Tarea15	0:14	0:11
Tarea16	0:40	0:52

Tabla 5.1: Tiempos de ejecución de las tareas programadas, en el test de usabilidad

5.1.5.6. Observaciones

Como ya se ha comentado previamente, tras la realización de cada una de las tareas, se le permitió a cada uno de los usuarios realizar determinadas observacio-

nes que ellos consideraran adecuadas, sean aspectos positivos o negativos hacia la aplicación. A continuación, se engloban estas observaciones separadas por tareas:

- **Tarea 2:**
  - Todos los usuarios consideran inútil la opción de que cada paciente tenga su foto de perfil y la quitarían de la aplicación, ya que serían molestias de cara al paciente estar solicitándoles una foto.
  - Un usuario aporta que sería interesante añadir un nuevo campo de información para los pacientes, llamado "Historial clínico", que englobe la siguiente información de utilidad para los fisioterapeutas: alergias, medicación, operaciones, antecedentes de lesiones, accidentes de tráfico y embarazos.
- **Tarea 3:**
  - A un usuario le costó inicialmente buscar cómo acceder a la edición del paciente, ya que no se fijaba en la barra superior donde se encuentran los botones de acciones "Editar" y "Borrar", pero tras encontrarla, accedió rápidamente el icono correspondiente para la edición del paciente, por lo que se aprecia que es suficientemente identificativo tal icono.
  - Además, otro usuario consideraría interesante cambiar el nombre de la pestaña "FICHA" por "INFO", ya que cree que la ficha personal como tal sería la unidad-conjunto de los tres apartados: información personal del paciente, sus imágenes y sus notas.
- **Tarea 4:**
  - Todos los usuarios encuentran interesantes el archivo de imágenes de los pacientes. Un usuario, en concreto, fue más allá y consideró que sería interesante la opción de que se pudieran añadir vídeos también, ya que los encontraría de utilidad en su caso.
- **Tarea 6:**
  - A los usuarios las notas les parecen interesantes para añadir información de las sesiones, como la patología por la que acude, el diagnóstico y el tratamiento aplicado.
- **Tarea 7:**
  - Un usuario, para acudir a la acción "Editar", su primera acción es pulsar sobre la tarjeta de la cita (se supone que para elegir luego la acción a realizar mediante un dialogo) y no el menú contextual de la tarjeta, aunque luego lo reconoce.
- **Tarea 8:**

  - Todos los usuarios recuerdan fácilmente que tienen que volver al apartado "FICHA" del paciente para hacer uso del botón de la barra de estado "Eliminar", y por tanto, eliminar el paciente indicado.

- **Tarea 9:**

  - Un usuario, al añadir un evento, navega erróneamente al ítem *Tareas* del menú de navegación, en vez de al ítem *Calendario*. Se encuentra confuso y pide ayuda. Se le indica el ítem correcto, y aprecia su fallo, indicando que *Calendario* le parece más lógico. Además, es interesante ver cómo accede para añadir el evento, ya que no lo hace en la primera pantalla del calendario, sino una vez seleccionado el día solicitado.

- **Tarea 10:**

  - A todos los usuarios les cuesta encontrar el evento solicitado (paciente Miguel Ángel Pérez), y es debido al formato de la tarjeta. Por ello, proponen cambiar el formato, intercambiando posiciones entre el fisioterapeuta y el nombre del paciente, para así darle importancia al paciente.

- **Tarea 13:**

  - Un usuario realmente no cumple lo solicitado. La acción que realiza es "Eliminar" (siguiente tarea), y no "Dar por finalizada". Una vez comentado el error, considera que con "Eliminar" la tarea basta. No considera útil dar por finalizado como paso intermedio para que resto de fisioterapeutas verifiquen la realización de la tarea. En esta idea también se encuentra otro usuario.

- **Tarea 15:**

  - Respecto al ítem *Pagos*, a un usuario le es bastante indiferente. Cree que en su caso al ser una consulta de 1 fisioterapeuta, y tener una única tarifa, y además, que no suele fiar pagos a sus pacientes, no parece que este apartado sea de gran utilidad para él. Sin embargo, es importante que permanezca tal y como está, porque en una clínica de fisioterapia, es normal dejar pendiente los pagos durante un tiempo.

5.1.6 Conclusiones finales

Una vez finalizado el test de usabilidad a los usuarios y analizar los resultados obtenidos, se ha detectado la necesidad de introducir algunos datos, además de modificar y eliminar otros. Las modificaciones oportunas que se han considerado debido a los resultados obtenidos son:

- **Pantalla Inicio**

  - En la lista de eventos para el día actual saldrán los eventos de la clínica de fisioterapia en general, destacando en otro color los eventos que son para el fisioterapeuta que ha iniciado sesión.

- **Pantalla Lista de pacientes**
  - La lista será ordenada alfabéticamente según el nombre.
- **Pantalla Ficha de paciente**
  - Quitar la posibilidad de que los pacientes tengan fotos de perfil.
  - Modificar el apartado/pestaña de esta sección llamado "FICHA" por "INFO".
  - Añadir campo "Historial clínico" a la información personal de cada paciente.
- **Pantalla Lista de eventos**
  - Cambiar el formato de las tarjetas específicas de cada evento, intercambiando las posiciones entre el fisioterapeuta y el nombre del paciente.
- **Pantalla Tareas**
  - Quitar función "Dar por finalizada", dejando únicamente la posibilidad de eliminar tareas.

5.2 Test de verificación del modelo de datos

Los test de verificación del código probarán alguna de las funcionalidades del *Modelo* de la aplicación, ya que ésta es la parte más crítica de la arquitectura *MVP*, descrita en el capítulo *Modelo de diseño*. De todos los posibles tests que se pueden realizar sobre el modelo, se han elegido una serie de ellos, con el fin de no alargar demasiado esta sección. A todo esto, es importante destacar que antes de publicar cualquier aplicación es fundamental realizar la batería de test completa. Llevar a cabo estas pruebas conlleva una técnica dinámica de verificación ya que se requiere de un prototipo ejecutable del sistema. Aunque, en realidad, no es necesario que todo el sistema se encuentre operativo, basta con que aquellas partes del sistema que se están probando estén en funcionamiento, en este caso, las clases del modelo de datos de la aplicación. Así, para testear estas clases se necesita crear clases de test, formadas por métodos de tests que prueben los distintos casos que se vayan a encontrar a la hora de invocar el método del modelo.

En esta sección del **Modelo de pruebas** se van a presentar algunos test, de los realizados, para verificar el grado de funcionamiento de la *FisioClinicApp: Aplicación para la gestión de una clínica de fisioterapia*. Prácticamente todos los métodos con los que cuenta el *Modelo* de la aplicación fueron sometidos a una serie de test

para comprobar dicho funcionamiento, a excepción de algunos, en concreto *actualizarInfoUsuario*, *actualizarPassword* y *darBajaUsuario*, debido a una razón común, y es la complejidad de llevar a cabo los test para estos métodos al necesitar un *login* previo, lo cual complicaba la tarea arduamente. Si bien es cierto que con la realización de los test se verifica el correcto funcionamiento de las sucesivas funcionalidades puestas a prueba, hay que remarcar que nunca se va a poder demostrar que el software está completamente libre de defectos (*bugs*), pero la elaboración de los mismos aporta un paso adelante en el desarrollo final de la aplicación.

A continuación se describen los test que se han realizado a cinco métodos del modelo. Como se dijo anteriormente, si bien sólo se presentan estos cinco ejemplos en la memoria, durante el proceso de verificación del modelo se llegaron a testear la mayoría de los métodos. Estos test pueden ser analizados en la carpeta *test* del código de la aplicación que se encuentra en la documentación adjunta al Trabajo de Fin de Grado.

5.2.1 Tests del método comprobarLogin

En estos test se comprueba el correcto funcionamiento del proceso de autenticación de un usuario, es decir, un fisioterapeuta, en la aplicación, a través de las pruebas realizadas al método *comprobarLogin* de la clase *Modelo*. Así, se requieren como parámetros iniciales el *email* y la *password* (contraseña) del usuario que se desea autenticar, que serán variables predefinidas para cada método (requisitos iniciales).

Dado que hay varios casos que manejar dentro de esta prueba, se han creado tres métodos de test para garantizar su correcto funcionamiento. Estos son:

- **testComprobarLoginCorrecto(): void**. Método en el que se define un *email* y una *password* (contraseña) que coinciden con los de un usuario registrado en la aplicación y, por tanto, almacenados en la nube. En este test se llama al método *comprobarLogin* del *Modelo*, pasándole por parámetros el email y la contraseña anteriores, los cuáles son válidos. Tras esto, se accede la nube, en concreto al servicio *Firebase Authentication*, para comprobar la autentidad de estos datos y obtener el resultado de tal comprobación. En este caso, el resultado esperado es la llegada de un valor lógico *true*. En caso contrario, es decir, que llegara un valor lógico *false*, significa que la validación de la autenticación no ha tenido éxito, y en este caso, se daría el test por fallido. En la figura 5.1 se demuestra el resultado final del test, donde la barrera de color verde indica que se ha ejecutado satisfactoriamente.
- **testComprobarLoginCorreoIncorrecto(): void**. Método en el que se define un *email* inválido, es decir, que no se ajusta al formato de correo electrónico estándar o que no es correcto al no existir en el servicio de autenticación,

Figura 5.1: Resultado del test *testComprobarLoginCorrecto*

además de una *password*. Al igual que en el test anterior, se llama al método *comprobarLogin* de la clase *Modelo*, al que se le pasa por parámetros el email y la contraseña mencionadas previamente. Tras esto, se accede a la nube, en concreto al servicio *Firebase Authentication*, y se comprueba la veracidad de los mismos, esperando la respuesta correspondiente con el resultado de dicha comprobación. En este caso, al estar insertando un email inválido, la respuesta ha de ser negativa, es decir, un valor lógico *false*. En caso de no llegar este valor, se consideraría el test como fallido. En la figura 5.2 se demuestra el resultado final del test, donde la barra de color verde indica que se ha ejecutado satisfactoriamente.

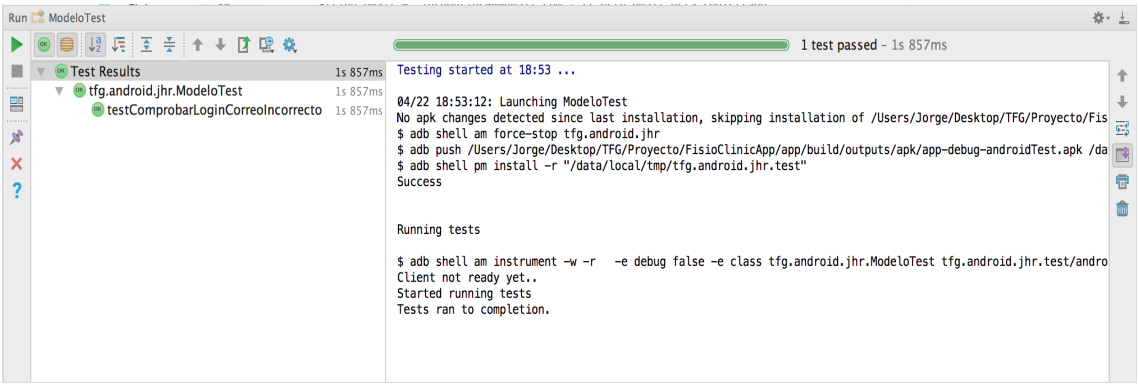

Figura 5.2: Resultado del test *testComprobarLoginCorreoIncorrecto*

- **testComprobarLoginEmailCorrectoPasswordIncorrecto(): void**. Método en el que se define un *email* correspondiente al de un usuario de la aplicación, pero la *password* que se entrega es inválida, es decir, no coincide con la del usuario con el email pasado por parámetros. Al igual que en el test anterior, se llama al método *comprobarLogin* de la clase *Modelo*, al que se le pasa por parámetros el email y la contraseña mencionadas previamente. Tras esto, se accede al servicio de autenticación de *Firebase Authentication* para comprar la veracidad de estos datos, esperando por la respuesta con el resultado dicha

comprobación. Al igual que en el test anterior, para considerar que se ha pasado el test correctamente se ha de recibir un valor lógico *false*, ya que el *login* no es correcto. Cualquier valor de otro tipo hará que el test sea considerado fallido. En la figura 5.3 se demuestra el resultado final del test, donde la barra de color verde indica que se ha ejecutado satisfactoriamente.

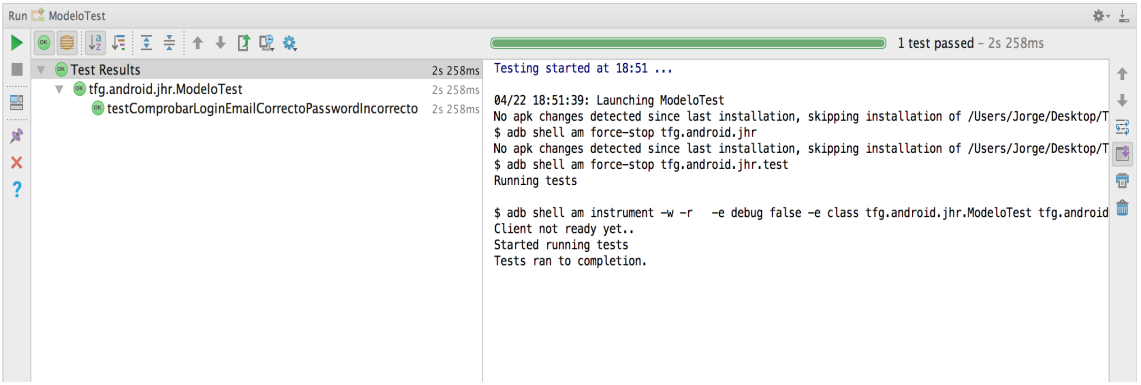

Figura 5.3: Resultado del test *testComprobarLoginEmailCorrectoPasswordIncorrecto*

5.2.2 Tests del método obtenerPassword

En estos test se comprueba el correcto funcionamiento del proceso de recuperación de la contraseña de un usuario que la ha olvidado y no puede acceder a la aplicación, a través del método *obtenerPassword* de la clase *Modelo*. Así, se requieren como parámetros iniciales el correo electrónico del usuario que no recuerda su contraseña. Este correo electrónico se hace llegar al servicio *Firebase Authentication*, el cual se encargará de comprobar la existencia del mismo en sus registros, y en caso positivo continuará con el envío de un correo electrónico a dicha dirección, dando la posibilidad al usuario de poder restablecer su contraseña, es decir, poder elegir una nueva.

En este caso, y al igual que con muchos de los tests que se pueden encontrar en la documentación adjunta a este Trabajo de Fin de Grado, la realización de los tests, concretamente los que implican *escrituras en la base de datos en la nube*, se realizaron de manera conjunta con el *dashboard* que proporciona *Firebase*, en este caso en su servicio *Firebase Database*. En él se puede apreciar en tiempo real las creaciones, modificaciones y eliminaciones de los datos almacenados de forma gráfica. Así, además de comprobar el valor devuelto a la hora de invocar el método a probar, se puede apreciar de forma visual el efecto de los tests sobre la base de datos. En el caso de este método, al estar haciendo uso de una funcionalidad que aporta el servicio *Firebase Authentication*, se recibe un correo electrónico, que sirve como prueba gráfica y complementaria para comprobar el correcto funcionamiento de este método.

Para esta prueba, se realizaron dos métodos de tests, que se comentan a continuación:

- **testObtenerPasswordEmailCorrecto(): void**. Método en el que se define un *email* con el objetivo de que se reciba el correo electrónico correspondiente para el restablecimiento de la contraseña del usuario. En este test se llama al método *obtenerPassword* de la clase *Modelo*, pasándole por parámetros el email mencionado. Tras esto, se hace la solicitud correspondiente al servicio *Firebase Authentication*, que se encargará de comprobar si el correo electrónico está registrado en la aplicación, en cuyo caso continuará con el envío de un correo electrónico a dicha dirección para que el usuario pueda elegir una nueva contraseña de acceso. En este caso, la respuesta debe ser un valor lógico *true*, que indique el resultado exitoso de la operación, considerando así que se ha pasado el test. En caso de obtener una respuesta diferente, el test será considerado como fallido. En la figura 5.4 se demuestra el resultado final del test, donde la barra de color verde indica que se ha ejecutado satisfactoriamente. Si bien con esto podría ser suficiente, como se comentó anteriormente, se aporta también el recibo del correo electrónico que se hace llegar al usuario que solicita esta funcionalidad en la aplicación. Este correo electrónico se puede apreciar en la figura 5.5.

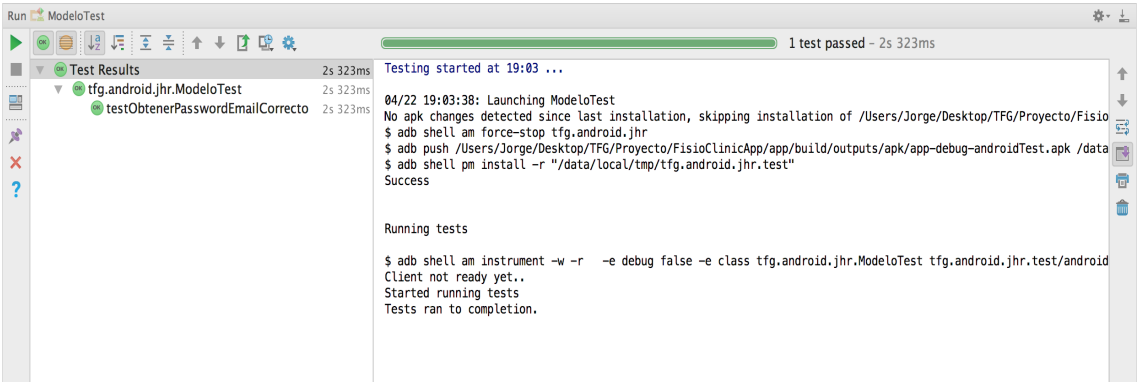

Figura 5.4: Resultado del test *testObtenerPasswordEmailCorrecto*

- **testObtenerPasswordEmailIncorrecto(): void**. Método en el que se define un *email* con el objetivo de que se reciba el correo electrónico correspondiente para el restablecimiento de la contraseña del usuario, pero en este test el email definido es inválido, es decir, tiene un formato incorrecto o no está registrado en la aplicación. En este test se llama al método *obtenerPassword* de la clase *Modelo*, pasándole por parámetros el email mencionado. Tras esto, se hace la solicitud correspondiente al servicio *Firebase Authentication*, que se encargará de comprobar si el correo electrónico está registrado en la aplicación, en cuyo caso continuará con el envío de un correo electrónico a dicha dirección para que el usuario pueda elegir una nueva contraseña de acceso. En este caso, como no se produce envío de correo electrónico al ser un email

Figura 5.5: Correo electrónico recibido tras la realización del test *testObtenerPasswordEmailCorrecto*

incorrecto, la respuesta debe ser un valor lógico *false*, que indique el resultado fallido de la operación, considerando así que se ha pasado el test. La obtención de otra respuesta hará que se considere el test como fallido. En la figura 5.6 se demuestra el resultado final del test, donde la barra de color verde indica que se ha ejecutado satisfactoriamente.

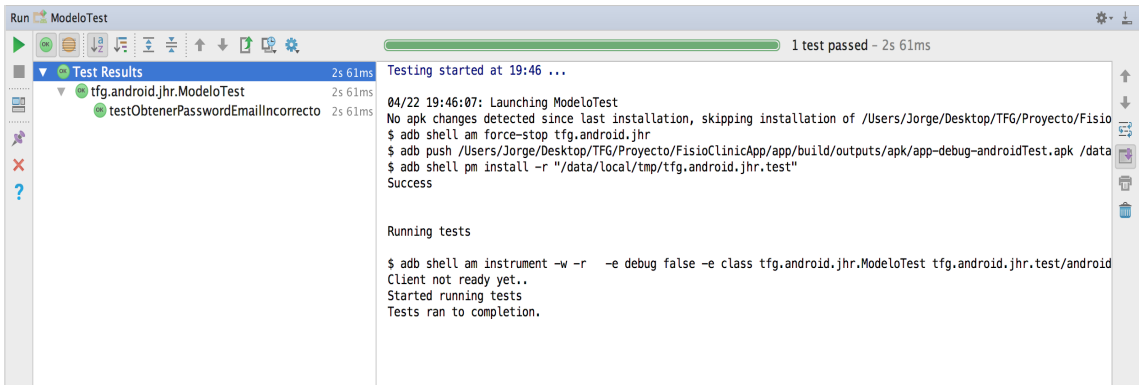

Figura 5.6: Resultado del test *testObtenerPasswordEmailIncorrecto*

5.2.3 Tests del método obtenerListaEventos

En estos test se comprueba el correcto funcionamiento del proceso de obtención de la lista de eventos de una determinada clínica para un día determinado, los cuales se encuentran registrados en la base de datos en la nube. Esta lista de eventos se hace llegar a la aplicación mediante el método *obtenerListaEventos* de la clase *BDAdaptadorEvento*. Así, se requieren como parámetros inicial el *día* del que se desea obtener la lista de eventos y el *identificador de la clínica*, los cuales serán

variablea predefinidas para el método (requisitos iniciales). Para esta prueba se realizó un método de test, que se presenta a continuación al detalle:

- **testObtenerListaEventos(): void**. Método en el que se define un *día* específico con el objetivo de obtener la lista de eventos para tal día. En este test se llama al método *obtenerListaEventos* de la clase *Modelo*, el cual hará a su vez la llamada correspondiente al método homónimo *obtenerListaEventos* de la clase *BDAdaptadorEvento*, que será el encargado de obtener la lista de eventos en cuestión. Además del día deseado, el método de la clase *Modelo* hace llegar un *identificador de clínica* al método de la clase *BDAdaptadorEvento*. Este identificador de la clínica se obtiene a partir del fisioterapeuta que ha iniciado sesión en la aplicación. Para la elaboración de este método, con el objetivo de no complicar su funcionamiento (ya que habría que realizar un *login* previo), se registra manualmente un fisioterapeuta en la clase *Modelo*, pudiendo así obtener un identificador de una clínica con el que trabajar. Con todo esto, el método *obtenerListaEventos* finalmente accederá a la base de datos en la nube y obtendrá los eventos con los parámetros especificados. En este caso, para considerar que se ha tenido éxito con el test, se ha de obtener una lista de eventos con el día solicitado y, además, deben estar ordenados en función de la hora de comienzo del evento. Todo resultado que no sea este mencionado, hará que el test sea considerado como fallido. En la figura 5.7 se demuestra el resultado final del test, donde la barra de color verde indica que se ha ejecutado satisfactoriamente. Además, a efectos de comprobar que se está obteniendo los eventos con los requisitos especificados, se presenta en la figura 5.8 el *log* resultado de la ejecución de este test.

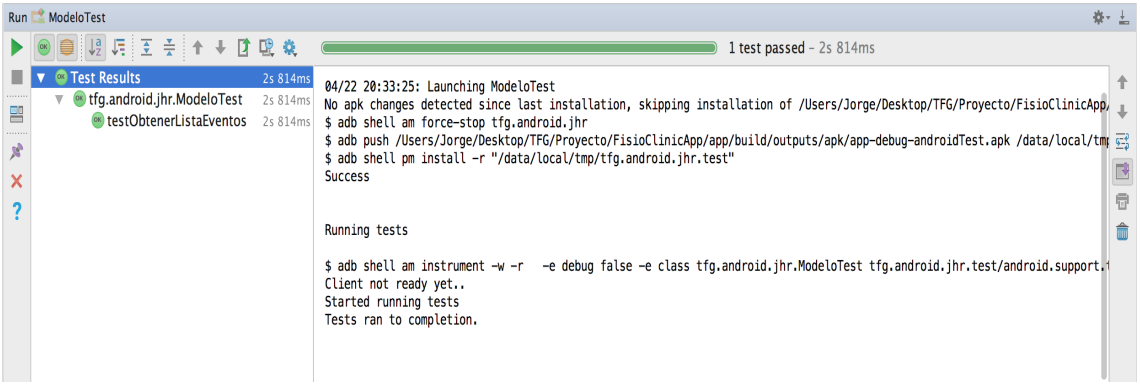

Figura 5.7: Resultado del test *testObtenerListaEventos*

5.2.4 Tests del método obtenerListaPacientes

En estos test se comprueba el correcto funcionamiento del proceso de obtención de la lista de pacientes de una determinada clínica, los cuales se encuentran registrados en la base de datos en la nube. Esta lista de pacientes se hace llegar a la

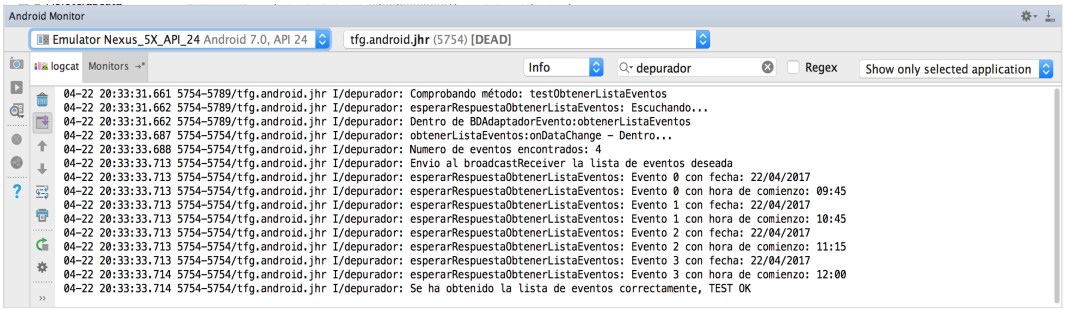

Figura 5.8: *Log* obtenido tras la ejecución del test *testObtenerListaEventos*

aplicación a través del método *obtenerListaPacientes* de la clase *BDAdaptadorPaciente*. Así, se requieren como parámetro inicial el *identificador de la clínica* de la que se desea obtener la lista de pacientes. Este identificador será una variable predefinida para el método (requisito inicial). Para esta prueba se realizó un método de test, que se explica con detalle a continuación:

- **testObtenerListaPacientes(): void**. Método en el que se define un *identificador de clínica* con el objetivo de obtener la lista de pacientes para esa clínica. Este identificador se define a través de la creación de un fisioterapeuta, que se registra manualmente en la clase *Modelo*, con el objetivo de simplificar el proceso de test, evitando la realización de un *login* previo. En este test se llama al método *obtenerListaPacientes* de la clase *Modelo*, que hará a su vez la llamada correspondiente al método homónimo *obtenerListaPacientes* de la clase *BDAdaptadorPaciente*, que será realmente el encargado de hacer llegar a la aplicación dicha lista, comunicándose con la base de datos en la nube. A este último método se le hace llegar el *identificador de la clínica* del fisioterapeuta logeado en la aplicación, que se encuentra registrado en la clase *Modelo*. En este caso, para considerar que se ha tenido éxito con el test se ha de obtener la lista de pacientes que pertenecen a la clínica indicada. Todo resultado que no sea el mencionado, hará que el test sea considerado como fallido. En la figura 5.9 se demuestra el resultado final del test, donde la barra de color verde indica que se ha ejecutado satisfactoriamente. Además, a efectos de comprobar que se está obteniendo los eventos con los requisitos especificados, se presenta en la figura 5.10 el *log* resultado de la ejecución de este test.

5.2.5 Tests del método agregarImagen

En estos test se comprueba el correcto funcionamiento del proceso de adición de una nueva imagen para un determinado paciente de una clínica concreta, a través del test al método *agregarImagen* de la clase *BDAdaptadorImagen*. Así, se requieren como parámetros iniciales los datos de la nueva imagen que se pretende añadir, es decir, la *imagen* como tal y su *descripcion*, además de un *identificador de paciente*

Figura 5.9: Resultado del test *testObtenerListaPacientes*

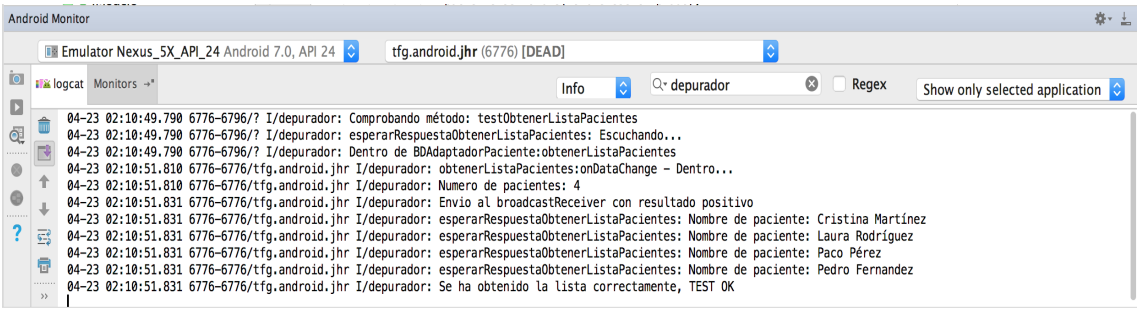

Figura 5.10: *Log* obtenido tras la ejecución del test *testObtenerListaPacientes*

e *identificador de clinica*. Todos estos datos serán variables predefinidas para el método (requisitos iniciales). Para esta prueba, se realizó un método de test, que se comenta enseguida:

- **testAgregarImagen(): void**. Método en el que se define una *imagen*, *descripcion*, *identificador de paciente* e *identificador de clinica*. Este último se define como se ha hecho en tests explicados anteriormente, es decir, a partir de la creación de un fisioterapeuta que se registra en la clase *Modelo*, simulando así que existe un fisioterapeuta/usuario que ha iniciado sesión. El objetivo, tal como ya se ha expresado, es evitar la realización de un *login* previo que pueda complicar la tarea de testeo de este método. En este test se llama al método *agregarImagen* de la clase *Modelo*, que a su vez hará la llamada al método homónimo *agregarImagen* de la clase *BDAdaptadorImagen*, que es el que se encarga realmente de la comunicación con la base de datos en la nube para la adición de la imagen. Además, también se encarga de comunicar a la aplicación cuál ha sido el resultado de la operación, elemento clave para la determinación de si el test ha sido exitoso o no. Así, se considerará que se ha pasado el test si el resultado es un valor lógico *true*. En caso contrario, se dará el test como fallido. En la figura 5.11 se demuestra el resultado final del test, donde la barra de color verde indica que se ha ejecutado satisfactoriamente, obteniendo el resultado esperado.

  Si bien con esto podría ser suficiente, como se comentó anteriormente, se puede utilizar el *dashboard* que ofrece *Firebase* como herramienta de verificación de

tests. Este método tiene dos partes claramente diferenciadas: el registro de la imagen en un paciente determinado en la base de datos, utilizando *Firebase Database*, y la subida de la imagen como tal, utilizando el servicio *Firebase Storage* para el almacenamiento de las mismas. En el caso de la primera parte, se muestra en la figura 5.12 como la imagen ha sido registrada para un determinado paciente. Y por otro lado, en la figura 5.13, se encuentra la imagen propiamente dicha, almacenada ya en la nube. En definitiva, estas dos imágenes nos reafirman en el resultado positivo que ha tenido la ejecución del test.

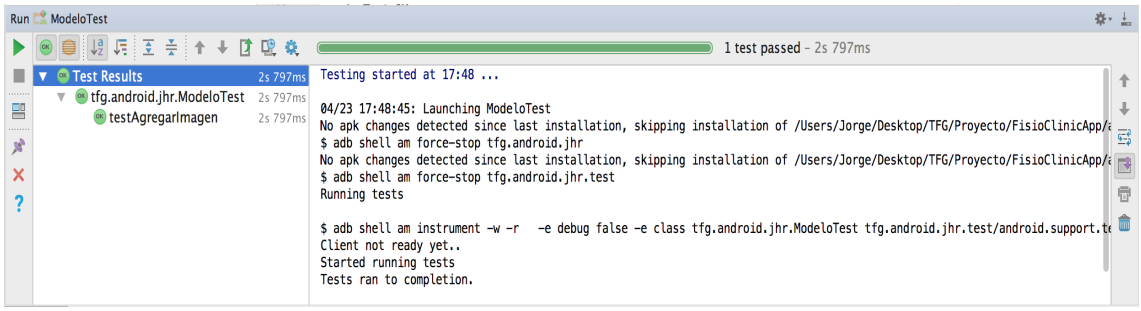

Figura 5.11: Resultado del test *testAgregarImagen*

Figura 5.12: Resultado tras la ejecución del test *testAgregarImagen* desde el *dashboard* de *Firebase Database*

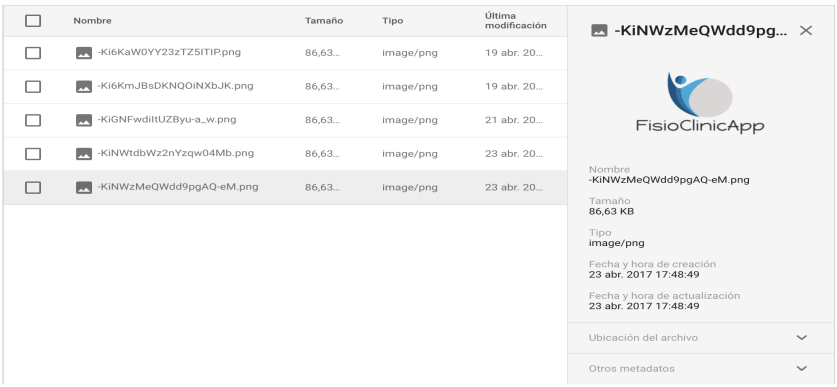

Figura 5.13: Resultado tras la ejecución del test *testAgregarImagen* desde el *dashboard* de *Firebase Storage*

Capítulo **6**

CONCLUSIONES Y MEJORAS

6.1 Conclusiones

La aplicación creada en el Trabajo de Fin de Grado presentado en esta memoria, tiene como principal objetivo permitir a los profesionales del sector de la fisioterapia, la gestión de su día a día, dentro del ambiente de una clínica de fisioterapia, consiguiendo mayor comodidad y productividad en sus tareas.

Para lograr este objetivo, inicialmente se planteó que los usuarios fueran capaces de realizar una serie de tareas como: registrarse y acceder a su propia cuenta en la aplicación, disponer de una ficha personalizada para cada uno de los pacientes de su clínica, donde encontrar su información personal, apuntes de sus sesiones e incluso contar con un archivo de imágenes propio, gestionar sus numerosos eventos del día a día, registrar pagos, anotar tareas a realizar, entre otras funcionalidades.

En la tabla 6.1 se puede analizar, de forma más detallada, el grado de consecución de las diferentes funcionalidades planteadas en el comienzo del Trabajo de Fin de Grado presentado. En esta tabla se indican los objetivos planteados así como los resultados obtenidos y, en caso de que no se llegara a realizar alguno de ellos, se especifica el motivo por el cual no se desarrolló.

Tal y como se puede apreciar en la tabla 6.1, la mayoría de los objetivos planteados inicialmente se han logrado con éxito, siendo uno de ellos eliminado de la aplicación final porque durante el desarrollo de esta se consideró que se trataba de una funcionalidad que no es objeto de esta aplicación, ya que la aplicación se desea que sea únicamente dedicada a la gestión de la clínica de fisioterapia como tal, y no a la gestión interna de la aplicación, delegando esta tarea a otro apartado futuro del producto, tal como se explicará posteriormente.

Por lo tanto, se puede concluir que la aplicación *FisioClinicApp*, objeto del presente Trabajo de Fin de Grado, ha sido desarrollada con éxito, convirtiéndose así en una aplicación real que permitirá a los fisioterapeutas que formen parte de una clínica de fisioterapia que haga uso de la aplicación, encontrar un punto común para sus tareas de gestión que realizan en el día a día, de manera cómoda y sencilla, donde y cuando quieran, gracias a las funcionalidades logradas planteadas en la tabla 6.1, entre otras.

Objetivo	**Resultado**	**Motivo**
Uso de servicio "backend" en la nube para la gestión de cuentas de usuario y almacenamiento de datos y archivos.	Logrado	-
Acceso a un archivo de pacientes en el que poder consultar sus fichas personalizadas, además de poder añadir nuevos pacientes, editarlos y eliminarlos.	Logrado	-
Búsqueda rápida en el archivo de pacientes.	Logrado	-
Visualización de una ficha personal para cada paciente, en la cual se cuente con información personal del paciente, sus próximos eventos, una galería de imágenes propia y un registro de anotaciones.	Logrado	-
Disponer de un calendario, sobre el que basar la gestión de eventos de la clínica de fisioterapia.	Logrado	-
Crear, consultar, editar y eliminar eventos cuando se accede a un día concreto del calendario.	Logrado	-
Poder listar, añadir y eliminar tareas en un apartado dedicado.	Logrado	-
Registrar los pagos que se van realizando en la clínica y también los que están pendientes de ser realizados.	Logrado	-
Acceder a un panel de usuario personal, en el que poder gestionar su cuenta de usuario, pudiendo editar su información personal, cambiar su contraseña e incluso dar de baja su cuenta de usuario.	Logrado	-
En caso de tener rol de administrador dentro de la clínica, se puede eliminar usuarios de la aplicación (en caso de que algún empleado sea despedido o haya abandonado la clínica)	No realizado	Se decidió eliminar de la aplicación final por cuestiones de diseño, prefiriendo dejar esta funcionalidad para otro apartado del producto (explicado posteriormente en el apartado de Mejoras)

Tabla 6.1: Objetivos planteados y logrados del Trabajo de Fin de Grado

En cualquier caso, pese al buen resultado final, la consecución de los objetivos anteriormente planteados ha tenido algunos momentos complicados. Sin embargo,

con esfuerzo, se ha conseguido salir de ellos y no han repercutido en el resultado deseado y final.

Un punto clave del desarrollo de la aplicación fue el aprendizaje del servicio *Firebase*. Este servicio de Google es relativamente nuevo (sufrió en los últimos meses una reconversión total en su manera de trabajar), pero está consiguiendo bastante popularidad en el sector del desarrollo de aplicaciones móviles, e incluso del desarrollo de aplicaciones web. Aunque el autor de este trabajo conocía la existencia de esta plataforma (de ahí que se deseara utilizar como tecnología "en la nube" para esta aplicación), nunca se había trabajado en ella. Por tanto, hubo que poner esfuerzo en el estudio de la misma, aprendiendo sobre cómo usarla en la aplicación, y más teniendo en cuenta que se utilizarían tres servicios de la misma: *Firebase Authentication* para el sistema de autenticación, *Firebase Database* para la base de datos y *Firebase Storage* para el almacenamiento de archivos (imágenes y vídeos). Sin embargo, se puede decir que el esfuerzo mereció la pena, ya que se considera que *Firebase* cuenta con numerosas virtudes y su implementación es realmente cómoda. Además, como se mencionó previamente, se trata de una tecnología en auge, por lo que su aprendizaje ha sido un punto a favor incluso en el desarrollo personal.

Otro de los puntos conflictivos, también en relación al uso de *Firebase*, fue la implementación de la batería de test para el modelo de la aplicación. De hecho, se puede decir que fue el punto más difícil en el desarrollo de la aplicación. Pero no en cuanto a la prueba de los mismos y que éstos fallaran (en este aspecto, el proceso de prueba fue bastante bueno y ningún método en concreto dio problemas grandes), sino en la configuración inicial de esta batería de tests, para que éstos fueran ejecutados y realmente conectaran con *Firebase*, para así poder comprobar el resultado de los mismos. Los problemas se debieron, principalmente, a que las opciones aportadas por Android, que se conocían para realizar tests unitarios, se encontraban obsoletas, por lo que hubo que encontrar otras soluciones. Una vez encontradas, había que comprobar que estos tests, una vez ejecutados, conectaran con *Firebase*, ya que al ser una plataforma en la nube, los tiempos de acceso y respuesta varían. Finalmente, todos los problemas fueron solucionados, consiguiendo una batería de tests extensa, realmente útil y que, en gran medida, permitió que el resultado de esta aplicación fuera exitoso.

Sin embargo, es importante indicar que la aplicación se ha testeado en condiciones totalmente óptimas (un solo usuario simultáneamente, buena conexión de red, etc.), siendo muchas las fuentes que pueden llegar a dar problemas, tanto a nivel externo, como a nivel interno (por ejemplo: cómo la aplicación responderá ante una gran de usuarios utilizándola de manera simultánea, variabilidad en el entorno de red, variedad de dispositivos, entre otros). Debido a esto, es posible que la aplicación no sea completamente funcional en todos los casos de uso existentes. En esta línea, se pretende que las posibles incidencias o errores detectados en el futuro puedan ser resueltas, garantizándose una actualización de la aplicación para todos sus usuarios (generando nuevas versiones de la aplicación), pero también a nivel de la infraestructura *backend* sobre la que se sostiene el uso de la aplicación. Todo con el objetivo

de que la aplicación esté siempre operativa, alcanzando un funcionamiento óptimo para sus usuarios.

Por último indicar, que el diseño de la aplicación aportó la mayoría de las clases y métodos de éstas. Sin embargo, debido a la implementación en Android, se tuvieron que hacer modificaciones de algunos métodos (en lo que tenía que ver con los parámetros pasados), crear nuevos métodos (sobre todo en relación a las vistas de la aplicación) y en algunos casos, crear nuevas clases (por ejemplo, las necesarias para la encriptación y el manejo de tablas y listas).

6.2 Mejoras

Si bien la aplicación cumple con la práctica totalidad de los objetivos planteados inicialmente, es cierto que se pueden realizar cambios, con el objetivo de aumentar sus prestaciones e incluso mejorarla como concepto de *producto* final. Algunas de las posibles mejoras a realizar se proponen a continuación:

- **Crecimiento de *FisioClinicApp* como producto**. Esta mejora consistiría en que *FisioClinicApp* no fuera únicamente en una aplicación Android, sino que también contara con una versión en el sistema operativo iOS y, además, una aplicación Web. De esta manera, se convertiría en una plataforma total. La aplicación Web, además de contar con la funcionalidad de las aplicaciones móviles, serviría también para la administración interna de la aplicación. Aquí residiría la funcionalidad, que estaba dentro de los objetivos de este Trabajo de Fin de Grado, pero que sin embargo no fue llevada a cabo: la gestión de los usuarios que acceden a la aplicación (tal y como se muestra en la tabla 6.1. Esta aplicación Web, que comúnmente se conocen como aplicación tipo *backoffice*, únicamente estaría orientada a usuarios con un rol *administrador* (encargados de las clínicas de fisioterapia que hagan uso del producto), y contarían con un *escritorio*, en el que visualizar estadísticas, además de contar con un apartado para la gestión de usuarios, pudiendo ver éstos, crear nuevos y eliminar ya existentes. De esta manera se conseguiría completar el sistema cerrado de cuentas de usuarios, que se propone en esta aplicación, delegando en el encargado de la clínica la creación de cuentas de usuarios.
- **Sumar nuevos roles a la aplicación**. Además del rol *administrador* que se comentó previamente, y el evidente para los *fisioterapeutas*, sería interesante para aquellas clínicas más grandes, otros roles para otro tipo de empleados, como pueden ser: *asistentes*. Cada tipo de rol se podría generar dentro de la aplicación Web explicada anteriormente, y el *administrador* podría asignar qué apartados de la aplicación puede visualizar. Por ejemplo, un asistente no debería poder acceder a las fichas de los pacientes, ya que éstas pueden contener información confidencial, pero sí poder acceder al calendario para añadir nuevos eventos.

- **Chat interno**. Hoy en día muchas empresas en general cuentan con chats grupales, en aplicaciones de mensajería populares como *WhatsApp* o *Telegram*. El objetivo del chat interno con el que contaría *FisioClinicApp* es focalizarlo a temas únicamente laborales, consiguiendo aislar la vida laboral de la personal de cada uno de los empleados de una clínica de fisioterapia. Existirían tanto conversaciones individuales, contando con un directorio donde encontrar a cada uno de los empleados, y también conversaciones grupales.

- **Galería multimedia**. Una de las ideas que aportó una de las personas entrevistadas en el *test de usabilidad* del producto, explicado en el capítulo *Modelo de pruebas*, es que el archivo de imágenes con el que cuenta cada paciente evolucionará a una galería multimedia, en el que poder también subir vídeos, por ejemplo, de las sesiones. Esta persona consideraba que, en su especialidad dentro de la fisioterapia, sería realmente útil.

- **Replanteamiento del apartado "Tareas"**. Uno de los principales motivos por el que se decidió implantar el apartado "Tareas", es para situaciones en las que, por ejemplo, se tiene que comprar material, como puede ser rollos de papel, agujas, etc. Este se prevé que sea el mayor uso que se le dé a esta opción, por lo que en un futuro se podría dedicar un único apartado a esta gestión de inventario dentro de la clínica, "Inventario", con funcionalidad personalizada. El apartado "Tareas" continuaría, pero para el resto de aspectos que puedan considerarse oportunos.

- **Sistema de notificaciones**. En caso de implementar un chat interno, es evidente que debería añdirse un sistema de notificaciones, para conocer cuándo existen nuevos mensajes en las conversaciones de los usuarios. También se podría usar para agregar más notificaciones en la aplicación, por ejemplo, el fisioterapeuta podría subscribirse a avisos previos al comienzo de cada uno de sus eventos, o a los que él desee en concreto.

- **Mejorar el trato de la información**. En este momento la aplicación no tiene la manera más óptima de manejar la información. La aplicación está preparada para conectar con la nube constantemente para traer la información que requiere en cada pantalla de la aplicación. Esto no es lo deseable, puesto que implica un consumo de recursos en cuanto la base de datos va aumentando, y se hace notar en el rendimiento de la aplicación. Mejorar este aspecto requiere tiempo, y por ello no se ha llevado a cabo en este Trabajo de Fin de Grado, pero en un futuro merecería la pena trabajar en esta situación: añadir persistencia de datos en el dispositivo, añadir bibliotecas para optimizar la carga de imágenes, "cachear" éstas o realizar peticiones a la nube únicamente cuando sea necesario (primera vez que se trae la información y cuando existen actualizaciones en la información, serían algunas de las primeras ideas a implantar).

Existen otras mejoras que se podrían plantear para llevar a cabo, pero se puede decir que éstas son las más importantes e interesantes. Además, pueden realizarse

mejoras en cuanto al diseño gráfico de la aplicación, poner elementos de mejor aspecto visual realizados por algún diseñador gráfico, con el objetivo de dar a la aplicación un aspecto único y especial, o incluso la optimización de código. No obstante, estas mejoras, junto con las descritas anteriormente, requieren de tiempo y experiencia, pero son perfectamente realizables si se dispone de ambas.

BIBLIOGRAFÍA

[1] Contreras, R., *Especial BYOD*. Computing, 2012. Información disponible en: http://www.computing.es/infraestructuras/especiales/1061620001801/especial-byod.1.html
Consultada Junio 2017.

[2] Leo, J., *Uno de cada dos médicos de Familia españoles ya es e-doctor*. Redacción Médica, 2013.
Información disponible en: https://www.redaccionmedica.com/secciones/gestion/uno-de-cada-dos-medicos-de-familia-ya-son-e-doctores-9679
Consultada Junio 2017.

[3] Risco, C. *La nueva 'mHealth': el negocio del doctor en el bolsillo*. TLIFE, 2015.
Información disponible en: http://tlife.guru/movilidad/mhealth-el-negocio-del-doctor-en-el-bolsillo/
Consultada Junio 2017.

[4] *Sitio web de Mobile World Capital Barcelona*.
Información disponible en: http://mobileworldcapital.com/es/
Consultada Junio 2017.

[5] Ditendria, *Informe Mobile en España y en el Mundo 2016*.
Información disponible en: http://www.ditrendia.es/wp-content/uploads/2016/07/Ditrendia-Informe-Mobile-en-España-y-en-el-Mundo-2016-1.pdf
Consultada Junio 2017.

[6] *Sitio Web sobre la Ley de Moore*.
Información disponible en: http://www.mooreslaw.org
Consultada Junio 2017.

[7] *Sitio Web de Android*.
Información disponible en: http://www.android.com
Consultada Junio 2017.

[8] *Sitio Web de iOS*.
Información disponible en: http://www.apple.com/es/ios/
Consultada Junio 2017.

[9] Guenveur, L., *Ventas de smartphones: Huawei y Samsung se disputan el liderazgo en España*. Kantar España, 2017.
Información disponible en: http://es.kantar.com/tech/movil/2017/enero-

2017-cuota-de-mercado-de-smartphones-en-españa/
Consultada Junio 2017.

[10] *Sitio Web de Google Play.*
Información disponible en: https://play.google.com/
Consultada Junio 2017.

[11] *Sitio Web de App Store.*
Información disponible en: http://www.appstore.com
Consultada Junio 2017.

[12] *Sitio Web de Firebase.*
Información disponible en: https://firebase.google.com
Consultada Junio 2017.

[13] *Agencia estatal. Boletín oficial del Estado. Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal.*
Información disponible en: http://www.boe.es/buscar/act.php?id=BOE-A-1999-23750
Consultada Junio 2017.

[14] Nielsen, J., *10 Usability Heuristics for User Interface Design.*
Información disponible en: http://www.nngroup.com/articles/ten-usability-heuristics/
Consultada Junio 2017.

[15] *Sitio Web de Balsamiq Mockups.*
Información disponible en: https://balsamiq.com/products/mockups
Consultada Junio 2017.

[16] *Modelo Objectory.*
Disponible en: http://metodologiasoo.wikispaces.com/Objectory,+por+Ivar+Jacobson
Consultada Junio 2017

[17] Alfredo Weitzenfield.
Ingeniería de software orientada a objeto con UML, Java e Internet.
Editorial Thomson.

[18] Potel, M., *MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java.*
Información disponible en: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf
Consultada Junio 2017.

[19] Acens White Paper, *Bases de datos NoSQL. Qué son y tipos que nos podemos encontrar.*
Información disponible en: https://www.acens.com/wp-content/images/2014/02/-bbdd-nosql-wp-acens.pdf
Consultada Junio 2017.

[20] Google, *Firebase Realtime Database.*
Información disponible en: `https://firebase.google.com/docs/database/-?hl=es-419`
Consultada Junio 2017.

[21] Google, *Firebase Authentication.*
Información disponible en: `https://firebase.google.com/docs/auth/?hl=es-419`
Consultada Junio 2017.

[22] Varotariya, V., *Easy way to Encrypt/Decrypt string in Android.* Comunidad StackOverflow.
Información disponible en: `http://stackoverflow.com/questions/40123319/-easy-way-to-encrypt-decrypt-string-in-android`
Consultada Junio 2017.

[23] Giro, S., *Security "Crypto" provider deprecated in Android N.*
Información disponible en :`https://android-developers.googleblog.com/2016/-06/security-crypto-provider-deprecated-in.html`
Consultada Junio 2017

[24] Varotariya. V., *Security "Crypto" provider deprecated in Android N.* Comunidad StackOverflow.
Información disponible en: `http://stackoverflow.com/questions/39097099/-security-crypto-provider-deprecated-in-android-n`
Consultada Junio 2017.

[25] *Android Developer. Building layouts with an Adapter.*
Información disponible en: `https://developer.android.com/guide/topics/ui/-declaring-layout.html#AdapterViews`
Consultada Junio 2017.

[26] *Android Developer. Creating Lists and Cards.*
Información disponible en: `https://developer.android.com/training/material/-lists-cards.html`
Consultada Junio 2017.

[27] Google, *Fragmentos.* Android Developers Guide.
Información disponible en: `https://developer.android.com/guide/components/-fragments.html?hl=es`
Consultada Junio 2017.

Parte II

Pliego de condiciones

PLIEGO DE CONDICIONES

Introducción

A continuación se especificarán los requisitos técnicos que permitirán la ejecución de la aplicación en terminales móviles con Android. Para ello, se establecerán las diferentes condiciones referentes al hardware y al software, así como la instalación de la aplicación. Por último, se expondrá la licencia de uso del software.

Requisitos

El hardware requerido para el correcto funcionamiento de la aplicación consiste en un terminal *Android* (*smartphone* o tableta) con la versión 5.0 o superior de su sistema operativo, conexión a Internet y una pantalla superior a 4.5 pulgadas. En este sentido, la aplicación ha sido probada en los siguientes dispositivos:

- Teléfono móvil Huawei P8.
- Tableta Xiami MiPad 2.

Para los cuales se garantiza (a través de las pruebas) su correcto funcionamiento.

Los distintos fabricantes de terminales móviles *Android* especifican, a través de las hojas de características, la versión del sistema operativo soportado. Asimismo, el terminal también lo indica, típicamente accediendo a: *Ajustes* → *Acerca del teléfono* → *Info de software* (o similar).

Para el funcionamiento de la parte del servidor de la aplicación, se ha utilizado *Firebase*. Dentro de esta plataforma fueron utilizados tres servicios concretos: *Firebase Authentication* para la gestión de las cuentas de usuario, *Firebase Database* para el almacenamiento de la información en bases de datos, y *Firebase Storage* para el alojamiento de imágenes. Estos servicios corresponden con una parte fundamental para el manejo de los datos de interés de la aplicación, sin los cuales se perdería la práctica totalidad de su funcionamiento. No obstante, la aplicación está desarrollada de tal forma que, en caso de necesitar en un futuro un cambio de servidor o servicio de *backend*, esto se pueda realizar con la mínima modificación del código de la aplicación.

Condiciones de instalación

Para la instalación de la aplicación desarrollada, el archivo a instalar se encuentra en la entrada *Código* del menú del disco compacto adjunto a esta memoria. Así, se procede a copiar el archivo *FisioClinicApp.apk* en la memoria externa (*sdcard* o memoria SD) en una ubicación cualquiera del terminal (se recomienda la carpeta *Aplicaciones* o similar). La copia puede llevarse a cabo mediante USB, *Bluetooth* o *Wi-Fi*. Otra forma más sencilla de recuperar el archivo para su instalación consiste en enviarla por email o accediendo a un enlace en la nube (donde esté almacenado el archivo *apk*). Asimismo, es necesario tener activada la opción de poder instalar aplicaciones que no se hayan descargado desde el *Google Play*. Para ello, hay que activar la opción que hay en *Ajustes* → *Pantalla bloque y seguridad* → *Fuentes desconocidas* (o similar).

Una vez copiado el archivo *apk* en la memoria externa del terminal móvil (primera opción), se procede a la instalación. Para ello es necesario que el usuario haya descargado e instalado previamente, a través del *Google Play* de *Android*, cualquier explorador de archivos, tal como *ES Explorador de Archivos* de *File Manager*, entre otros. Así, se ejecuta el explorador de archivos en el terminal móvil y se entra en la carpeta donde fue transferido el archivo *FisioClinicApp.apk*. Al seleccionar el archivo ha de salir un menú contextual en el que se sugieren varias opciones. Entre ellas, *Instalar*. Si se elige *Instalar*, aparecerá una ventana que explica al usuario los permisos que concede a la aplicación. Si el usuario está de acuerdo, se elige *Aceptar*, con lo que la instalación queda realizada. En el caso de haber utilizado la segunda opción, una vez descargado el archivo por email o mediante un enlace, se selecciona *Instalar*, apareciendo la ventana indicada anteriormente.

Una vez instalada la aplicación, se ha de acceder al listado de aplicaciones del terminal, donde la aplicación objeto de este trabajo vendrá representada por un icono característico (figura 6.1). Mediante la selección del icono será ejecutada la aplicación.

Figura 6.1: Icono de la aplicación *FisioClinicApp*

Licencia del programa

Normas generales

De conformidad con lo dispuesto en la normativa reguladora en materia de Propiedad Intelectual, el TFT se considera una obra en colaboración entre el estudiante y la tutora. Para la explotación industrial del TFT será de aplicación lo establecido en los Estatutos de la Universidad de Las Palmas de Gran Canaria.

Asimismo, el uso de esta aplicación ha de ser bajo la expresa autorización del autor o de la tutora del trabajo o de la Escuela de Ingeniería de Telecomunicación y Electrónica de la Universidad de Las Palmas de Gran Canaria.

Derechos de autor

La aplicación y la documentación están protegidos por la ley de Propiedad Intelectual aplicable, así como por las disposiciones de los tratados internacionales. En consecuencia, el usuario podrá usar copia de esta aplicación, así como del código fuente de programación y de la documentación, siempre bajo la autorización del autor o de la tutora del trabajo o de la Escuela de Ingeniería de Telecomunicación y Electrónica de la Universidad de Las Palmas de Gran Canaria.

Garantía

Se garantiza el correcto funcionamiento de la aplicación en el momento de la instalación de acuerdo con las especificaciones vistas con anterioridad. La instalación de la aplicación no ocasionará la aparición de defectos en los dispositivos que cumplan con las especificaciones técnicas. En este sentido, la aplicación se encuentra, en el momento de la redacción, en su versión inicial (1.0). Esto significa que a medida que se vayan descubriendo posibles incidencias o errores, estos podrán ser detectados y solucionados, llevándose a cabo actualizaciones de la aplicación para garantizar su correcto y óptimo funcionamiento.

Con la única excepción de lo expresamente expuesto en el párrafo anterior, la aplicación ha sido desarrollada sin garantías de ninguna clase. El autor no asegura, garantiza o realiza ninguna declaración respecto al uso o los resultados derivados de la utilización del programa o de la documentación. Tampoco se garantiza que la operación del programa sea interrumpida o sin errores. En ningún caso serán el autor, tutora o la Escuela de Ingeniería de Telecomunicación y Electrónica de la Universidad de Las Palmas de Gran Canaria responsables de los perjuicios directos, indirectos, incidentales, ejemplarios o consiguientes gastos, lucro cesante, pérdida de capital, interrupción de negocios, pérdida de información comercial o de negocio o

cualquier otra pérdida que resulte del uso o de la incapacidad de usar la aplicación o la documentación. El usuario conoce y acepta que los derechos de licencia reflejan esta asignación de riesgo como el resto de cláusulas y restricciones.

Otras consideraciones

La fiabilidad de operación de la aplicación puede estar afectada por factores adversos, a los que se denominan "fallas del sistema". En estos se incluyen errores en el funcionamiento del hardware del dispositivo, sistema operativo o entorno del mismo, compiladores o software de desarrollo usado para realizar la aplicación, software externo para el funcionamiento de la misma, errores de instalación, problemas de compatibilidad del software y hardware, fallos o funcionamiento incorrectos de equipos de control, fallas por uso, errores por parte del usuario de la aplicación o problemas en el acceso de Internet. En el supuesto de que cualquier disposición de esta licencia sea declarada total o parcialmente inválida, la cláusula afectada será modificada convenientemente de manera que sea ejecutable y una vez modificada, plenamente eficaz, permaneciendo el resto de este contrato en plena vigencia. Esta licencia se regirá por las leyes de España. El usuario o licenciatario acepta la jurisdicción exclusiva de los tribunales de este país en relación con cualquier disputa que pudiera derivarse de la presente licencia.

Parte III

Presupuesto

PRESUPUESTO

Introducción

El COGITTCAN (Colegio Oficial de Graduados e Ingenieros Técnicos de Telecomunicación de Canarias) informa, en su página web[1], en base a la nota remitida a todos los colegios profesionales por el Ministerio de Economía y Hacienda, que siguiendo directivas europeas se deben eliminar los baremos orientativos de honorarios. Esto significa que los honorarios se establecen como un libre acuerdo entre el profesional y el cliente. A continuación se detalla la forma en la que el autor de este Trabajo Fin de Grado establece la baremación del proyecto presentado.

Así, el presupuesto presentado se divide en las siguientes partes:

- Tarifa de honorarios por tiempo empleado.
- Amortización de los equipos empleados.
  - Amortización del material hardware.
  - Amortización del material software.
- Coste de acceso a Internet.
- Redacción de la documentación.
- Derechos de visado.
- Gastos de tramitación y envío.

En las siguientes secciones se analizará, en detalle, cada una de estas partes.

[1] http://www.coittcan.es/index.php/servicios/orientacionlaboral

Tarifa de honorarios por tiempo empleado

Este concepto contabiliza los gastos correspondientes a la mano de obra. Para realizar este cálculo, se propone la siguiente fórmula:

$$H = (14,48 \times Hn) + (20,27 \times He)$$

Siendo:

- **H**: honorarios.
- **Hn**: honorarios en jornada laboral normal.
- **He**: honorarios fuera de la jornada laboral normal.

En el trabajo presentado en esta memoria, se ha empleado un periodo de aproximadamente **4** meses, trabajando alrededor de **5** horas diarias, en horario laboral normal (300 horas totales).

Distribución de la temporización del trabajo

El trabajo se divide en cuatro etapas bien diferenciadas, que se pasan a definir:

Definición

Esta etapa está dedicada a la definición de los requisitos de la aplicación, es decir, ¿qué problema se quiere resolver? Para realizar este proceso hay que implementar un prototipo de la aplicación y probar con usuarios reales para detectar las debilidades del diseño planteado.

Diseño

En esta etapa se estudian cada una de los módulos de la aplicación. A medida que se desarrolle esta parte debe distinguirse la actividad que desempeña cada módulo y la interrelación de los mismos (mensajes que se envían). Asimismo, se especifica cómo será el modelo de datos y dónde se debe almacenar la información.

Codificación

En esta etapa se realiza el desarrollo de la aplicación, es decir, se traduce lo establecido en el diseño a código, concretizando desde la generalidad a la particularidad, en función del lenguaje elegido (en este caso, Java para Android).

Verificación

En esta etapa, el desarrollador verifica que el código realizado cumple los requisitos establecidos en el primer punto. En este Trabajo Fin de Grado se verifica, exclusivamente, parte del modelo de datos, que es la parte más crítica de la aplicación. Dentro de la parte de verificación se ha incluido la etapa de *Control y Seguimiento* del TFG, que corresponde con la última tarea del anteproyecto presentado.

Redacción

En esta etapa se lleva a cabo la redacción de la documentación y su posterior revisión. En el caso de este Trabajo Fin de Grado, esta etapa se extiende a lo largo del tiempo dedicado al trabajo, correspondiendo a los entregables definidos en su día, en el diagrama de Gantt presentado en el anteproyecto.

Cálculo de tarifa de honorarios por tiempo empleado

En la tabla 6.1 se detalla el tiempo empleado para cada una de las etapas anteriormente descritas y se realiza el cálculo de los honorarios.

Etapa	**Horas laborales**	**Honorarios**
Definición	52	752,96 €
Diseño	57	825,36 €
Codificación	88	1274,24 €
Verificación	30	434,40 €
Redacción	73	1057,04 €
Total		**4344** €

Tabla 6.1: Honorarios por tiempo empleado

Por lo que sumadas cada una de las etapas, el cálculo de honorarios por tiempo empleado asciende a CUATRO MIL TRESCIENTOS CUARENTA Y CUATRO EUROS.

Amortización de los equipos empleados

Dentro de este concepto se considera tanto la amortización del hardware como del software empleado en la realización del trabajo presentado. De este modo, se estipula el coste de amortización para un período de 3 años, utilizando un sistema de amortización lineal o constante. En este sistema, se supone que el inmovilizado material se deprecia de forma constante a lo largo de su vida útil. La cuota de amortización anual se calcula haciendo uso de la siguiente fórmula:

$$C = \frac{Vad - V}{N}$$

Donde:

- **C**: cuota de amortización anual.
- **Vad**: valor de la adquisición.
- **V**: valor residual.
- **N**: número de años de vida útil de la adquisición.

Siendo el valor residual el valor teórico que se supone tendrá el elemento en cuestión después de su vida útil, teniendo en cuenta los índices de depreciación actual. En el caso del hardware y del software son 3 años (al 33 % de depreciación máximo por año).

Amortización del material hardware

Debido a que el trabajo se ha elaborado en un periodo inferior a 3 años, que es el periodo en que se calcula la amortización de material hardware, se realizará una amortización equiparable al período de duración del mismo (300 horas en 4 meses). Según esto, se obtienen los gastos expuestos en la tabla 6.2.

Por lo tanto, el coste total de hardware asciende a la cantidad de DOSCIENTOS TRECE EUROS CON SESENTA Y SIETE CÉNTIMOS.

Amortización del material software

El coste total del software utilizado es la suma de las herramientas software empleadas para la realización del trabajo (tabla 6.3). De esta manera se describe cada uno de las herramientas utilizadas y el coste supuesto para su utilización usando la fórmula de amortización anteriormente citada.

Por tanto, el coste total de software asciende a CATORCE EUROS CON CINCUENTA Y SEIS CÉNTIMOS.

Descripción	Unid.	Valor de adquisición	Tiempo de uso	Coste anual	Total
Ordenador portátil MacBook Pro Retina 13 pulgadas, Intel Core i5 @ 2.7 GHz, 8 Gb de RAM	1	1454,00 €	4 meses	484,68 €	161,56 €
Teléfono móvil Huawei P8	1	298,00 €	4 meses	99,34 €	33,11 €
Tableta Xiami MiPad 2	1	171,00 €	4 meses	57,00 €	19,00 €
Total (€)					**213,67**

Tabla 6.2: Precios y costes de amortización del hardware

Descripción	Valor de adquisición	Tiempo de uso	Coste anual	Total
Android Studio	0,00 €	4 meses	0,00 €	0,00 €
SDK y ADT Android	0,00 €	4 meses	0,00 €	0,00 €
Balsamiq Mockups (para desarrollo de muestras de la aplicación en formato pdf)	62 €	4 meses	20.67 €	6,89 €
Microsoft Office 365 Personal	69,00 €	4 meses	23,00 €	7,67 €
Total (€)				**14,56**

Tabla 6.3: Precios y costes de amortización del software

Coste de acceso a Internet

Para la conexión a Internet se dispone de una solución de fibra óptica a 300Mbps cuyo coste mensual es de 56,68 euros. Debido a que se ha usado dicha conexión durante cada una de las etapas de creación del mismo (4 meses), el coste de acceso a Internet se traduce a un total de DOSCIENTOS VENTISÉIS EUROS CON SETENTA Y DOS CÉNTIMOS.

Redacción de la documentación

Al coste de redacción obtenido hasta el momento se le deben añadir otros gastos, quedando el importe final de redacción del trabajo como se describe en la tabla 6.4.

Por lo tanto, la redacción del trabajo asciende a un total de MIL OCHENTA EUROS CON OCHENTA Y CUATRO CÉNTIMOS.

Concepto	**Coste**
Redacción del trabajo	1.057,04 €
Coste de impresión con tinta color	15,80 €
Encuadernación	5,00 €
CDs de 700MB	3,00 €
Total	**1.080,84** €

Tabla 6.4: Coste final de redacción del trabajo

Derechos de visado

El COITT establece que los derechos de visado para las aplicaciones de Integración de Sistemas en el año 2016 se calculan de acuerdo con la siguiente ecuación[2]:

$$V = 0,0035 \times P \times C \tag{6.1}$$

Siendo:

- **V**: coste del visado.
- **P**: presupuesto.
- **C**: coeficiente reductor en función del presupuesto.

El valor del coeficiente se obtiene de la tabla 6.5, mientras que el valor de P es el valor total del presupuesto, que se pasa a calcular en la tabla 6.6.

Coste del presupuesto (€)	**Factor de Correlación (C)**
Hasta 30.050	1
Exceso de 30.050 hasta 60.101	0.9
Exceso de 60.101 hasta 90.151	0.8
Exceso de 90.151 hasta 120.202	0.7
...	

Tabla 6.5: Tabla de coeficientes para el cálculo del visado

Por lo tanto, el valor del visado será de:

$$V = 0,0035 \times 4.822,77 \times 1 = 16,88 \tag{6.2}$$

Los costes de derechos de visado del trabajo ascienden a un total de DIECISÉIS EUROS CON OCHENTA Y OCHO CÉNTIMOS.

[2]https://www.coit.es/sites/default/files/upload/151123_tarifas_2017_visado.pdf

Concepto	Coste
Tarifa de honorarios por tiempo empleado	4.344 €
Amortización del material hardware	213,67 €
Amortización del material software	14,56 €
Coste de acceso a Internet	226,74 €
Gastos adicionales de redacción	23,80 €
Total	**4.822,77** €

Tabla 6.6: Cálculo total P para el cálculo del visado (Presupuesto base)

Gastos de tramitación y envío

Los gastos de tramitación y envío según la tarifa asciende a SEIS EUROS por cada documento visado de forma telemática.

Presupuesto antes de impuestos

Sumando todos los conceptos calculados hasta el momento, se obtiene el presupuesto, sin incluir los impuestos, que se muestra en la tabla 6.7.

Concepto	Coste
Presupuesto base	4.822,76 €
Derechos de visado	16,88 €
Gastos de tramitación y envío	6,00 €
Total	**4.845,64** €

Tabla 6.7: Presupuesto total sin impuestos

El presupuesto calculado, antes de incluir los impuestos, asciende a CUATRO MIL OCHOCIENTOS CUARENTA Y CINCO EUROS CON SESENTA Y CUATRO CÉNTIMOS.

Presupuesto incluyendo impuestos

Al presupuesto calculado anteriormente hay que incluirle un 7 % de IGIC obteniendo el coste del presupuesto final (tabla 6.8).

Por tanto, el presupuesto total, incluyendo impuestos, asciende a la cantidad de CINCO MIL CIENTO OCHENTA Y CUATRO EUROS CON OCHENTA Y TRES

Concepto	**Coste**
Total (sin IGIC)	4.845,64 €
IGIC (7 %)	339,19 €
Total	**5.184,83** €

Tabla 6.8: Presupuesto total

CÉNTIMOS.

Las Palmas de Gran Canaria, a 08 de Junio de 2017.

Fdo: Jorge Hernández Ríos

Parte IV

Apéndices

Apéndice **A**

SEGURIDAD Y PRIVACIDAD DE LA INFORMACIÓN

A.1 Introducción

La *Aplicación Android para la gestión de una clínica de fisioterapia* objeto de este Trabajo de Fin de Grado, *FisioClinicApp*, cuenta con la particularidad de que almacena información personal de pacientes y fisioterapeutas, e incluso de las propias clínicas de fisioterapia, que están dadas de alta en la aplicación. Este almacenamiento de la información, además, se realiza en servidores externos, gracias al uso de la plataforma *Firebase*. Esto ya de por sí es un factor muy importante a tener en cuenta a la hora de desarrollar una aplicación móvil, pero se hace más serio aún cuando se está hablando de información delicada y confidencial, relacionada con la salud de los pacientes de una clínica.

Por tanto, toda medida que pueda garantizar la seguridad y confidencialidad de esta información es poca, y por ello, a la hora de elaborar este Trabajo de Fin de Grado, se ha tenido muy en cuenta esta situación, con el objetivo de que no sea un impedimento en su desarrollo.

A.2 Marco jurídico

La *Ley Orgánica 15/1999*, de 13 de diciembre, de Protección de Datos de Carácter Personal[1], expresa que la protección de datos de carácter personal es un derecho fundamental reconocido en la Constitución Española, que atribuye al titular de los datos la facultad de controlar, disponer y decidir sobre sus datos. Las empresas que manejan y tratan datos de este tipo, en calidad de agentes, están obligados a garantizar este derecho fundamental. En el caso de este Trabajo de Fin de Grado, el autor no almacena la información como tal, como se ha mencionado previamente, sino que se almacena en servidores externos en la nube. En este caso, como desarrollador de la aplicación, se tiene la responsabilidad de seleccionar una empresa que cumpla con la normativa, incluso a nivel de la Unión Europea. Como ya se ha mencionado, se

[1] http://www.boe.es/buscar/act.php?id=BOE-A-1999-23750

delega el almacenamiento de la información en la plataforma *Firebase*, perteneciente a la empresa *Google*.

Tras la inhabilitación del acuerdo "Safe Harbor"[2], en julio de 2016 se firmó un nuevo tratado, denominado "Privacy Shield"[3] entre la Unión Europea y el Departamento de Comercio de los EEUU, en el que se somete a empresas americanas, como Google, a cumplir con obligaciones más estrictas a la hora de contar con datos que provienen de países de Europa. De esta manera, cualquier empresa española que quiera comenzar a usar los servicios de una empresa estadounidense, que suponga el tratamiento de datos en aquel país, tan solo tendrá que revisar que dicha empresa americana está en el listado del acuerdo "Privacy Shield". Esto querrá decir que Google, encargada del tratamiento, se ha comprometido a seguir las reglas del acuerdo y someterse a ellas. Por tanto, Google se encuentra al día con las normativas de privacidad a nivel europeo.

Por otro lado, se espera que en 2018 se aplique en España la nueva normativa R.G.P.D.[4], aprobada en el Parlamento Europeo, modificando la anterior L.O.P.D., la cual surge con un objetivo claro de normalizar a los países miembros en materia de protección de datos, y con ello mayores exigencias a las empresas de tratamiento de información. El paso final que, además, se espera de una empresa como Google, es que permita la libre elección de en cuál de sus *CPDs* (Centros de Procesamiento de Datos) se quiere almacenar la información, tema que no se ha adaptado todavía en una plataforma como *Firebase* al ser tan joven.

A.3 Medidas adoptadas

Además de elegir una empresa de tratamiento de información fiable, se decide adoptar otras medidas para garantizar la privacidad y seguridad de esta información:

- **No guardar la sesión del usuario que ha iniciado sesión previamente**. Aunque parezca algo relativamente sencillo, se trata de una medida bastante importante. Hoy en día la mayoría de aplicaciones móviles, con el objetivo de dar comodidad a sus usuarios, deciden guardar la sesión del usuario aunque éste salga de la misma, y de esta manera al volver, no necesitaría insertar su identificador de usuario y contraseña. En el caso de la aplicación *FisioClinicApp*, interesa que la persona que está haciendo uso de la aplicación móvil sea realmente el fisioterapeuta, que es el único que debe tener acceso a la información confidencial que aparece en la aplicación. Por ello, requerir la contraseña a menudo supone cierta incomodidad, pero es una importante ventaja en materia de privacidad de la información.

[2]http://2016.export.gov/safeharbor/
[3]https://www.privacyshield.gov/
[4]https://www.boe.es/doue/2016/119/L00001-00088.pdf

- **Cifrado de la aplicación**. Se trata de la medida más importante adoptada en materia de seguridad y privacidad. En todo momento la información que se almacena en los servidores en la nube y que fluye por la red hasta llegar a la aplicación, se encuentra cifrada utilizando el cifrado AES (Advanced Encryption Standard)[5], que es uno de los más utilizados hoy en día. Tal como se aprecia en la figura A.1, en ningún momento el administrador de la aplicación puede visualizar la información almacenada. En definitiva, se consigue así que ninguna persona externa a los fisioterapeutas de la clínica pueda leer estos datos tan importantes.

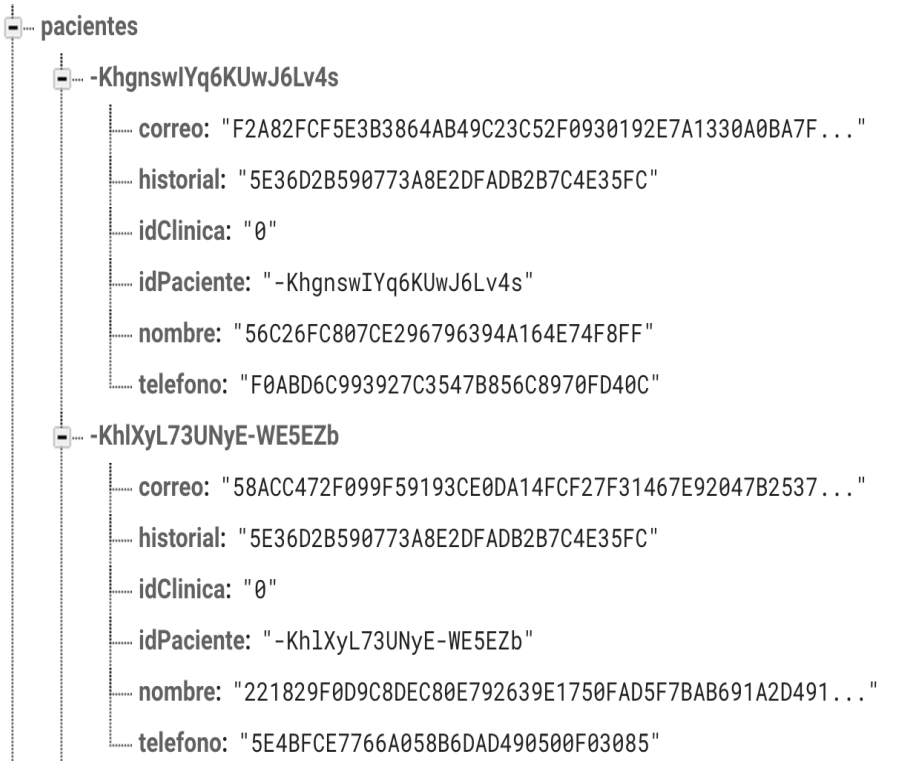

Figura A.1: Ejemplo de almacenamiento de la información relevante de la aplicación en *Firebase*

[5]https://www.boxcryptor.com/es/cifrado

Apéndice **B**

GESTIÓN DE USUARIOS

B.1 Sistema cerrado de usuarios

La aplicación objeto de este Trabajo de Fin de Grado, *FisioClinicApp*, tiene una característica particular y es que no permite registro de usuarios a los que instalan la aplicación, lo cual es un hecho poco común actualmente en las aplicaciones móviles. La aplicación está planteada de manera que, una clínica de fisioterapia que desea contar con los servicios de la aplicación, deba ser dada de alta en la misma por el administrador de la aplicación. Dado que los usuarios de esta aplicación van a ser los fisioterapeutas de cada una de las clínicas dadas de alta, resulta evidente que cualquier persona no puede registrarse libremente. Por ello, cuando un fisioterapeuta quiere hacer uso de la aplicación y, por tanto, tener su propia cuenta de usuario, ésta debe ser solicitada al encargado de su clínica, y en última instancia, éste le hará llegar la solicitud al administrador de la aplicación, para que registre una nueva cuenta de usuario para el fisioterapeuta en cuestión. De esta manera, se controla qué usuarios tiene, cada clínica de fisioterapia, dados de alta.

Por otro lado, cada usuario dispone de la funcionalidad de dar de baja a su cuenta, por ejemplo si abandona la clínica de fisioterapia. Sin embargo, el encargado de cada clínica puede contactar con el administrador de la aplicación para inhabilitar o, incluso, eliminar cualquier cuenta que desee.

B.2 Aplicación para creación de usuarios

Para el desarrollo de este Trabajo de Fin de Grado, se decidió crear de manera paralela una aplicación para la creación de usuarios. Esto es debido a que *Firebase*, en su servicio *Firebase Authentication*, pese a contar con un panel para la gestión de los usuarios (figura B.2), no permite crearlos con las características especiales con las que se cuenta en *FisioClinicApp*. Así, la opción "AGREGAR USUARIO" de la figura B.2 permite crear un usuario, pero sólo deja insertar el correo electrónico y la contraseña.

Sin embargo, los usuarios (fisioterapeutas de la clínica) cuentan con más información, como es su nombre, teléfono y el identificador de la clínica a la que pertenecen.

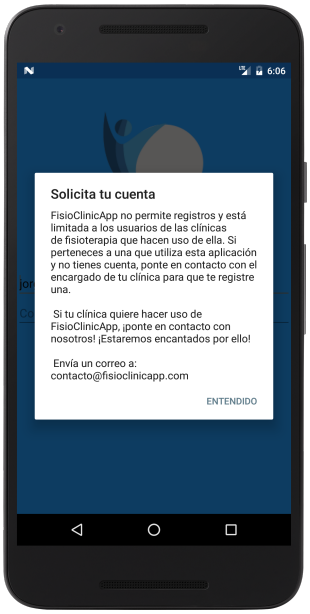

Figura B.1: Diálogo mostrado en la aplicación cuando no se tiene cuenta de usuario

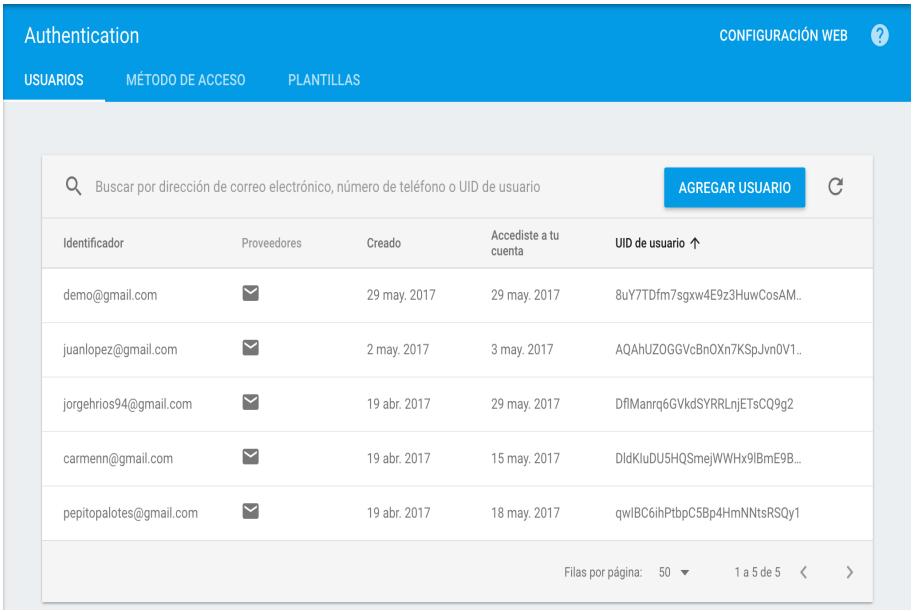

Figura B.2: Panel de gestión de los usuarios en *Firebase*

Para crear los usuarios con las características deseadas, sólo se puede realizar mediante programación, y por ello se crea la aplicación llamada *Gestión Usuarios FCA*, que se puede observar en la figura B.3. En esta aplicación, se rellenan los campos que se muestran para crear un nuevo usuario; y también sirve también para dar de alta una nueva clínica, que se haría, básicamente, creando el primer usuario para dicha clínica, determinando con él el identificador de la clínica que ésta va a adoptar dentro de *FisioClinicApp*.

Figura B.3: Aplicación *Gestión Usuarios FCA*

En el capítulo dedicado a las *Conclusiones y mejoras* de este Trabajo de Fin de Grado se plantea que, en un futuro, se cuente con una aplicación Web que, además de contener la funcionalidad de esta aplicación móvil Android, permita también ciertas tareas de administración interna de las aplicaciones, como es, en este caso, la gestión de los usuarios. El encargado de cada clínica de fisioterapia dada de alta, contaría con un panel donde poder él personalmente ver los usuarios existentes, además de crear nuevos y eliminarlos.