UNIVERSIDAD DE VALLADOLID

E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS ESPECÍFICAS DE TELECOMUNICACIÓN, MENCIÓN EN TELEMÁTICA

Desarrollo de una aplicación Android de hepatología configurable desde Web

Autor:
D. Óscar Valderrama García
Tutor:
Dra. Dña. Míriam Antón Rodríguez
Valladolid, 30 de agosto de 2018

TÍTULO:	**Desarrollo de una aplicación Android de hepatología configurable desde Web.**
AUTOR:	**D. Óscar Valderrama García**
TUTOR:	**Dra. Dña. Míriam Antón Rodríguez**
DEPARTAMENTO:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	**Míriam Antón Rodríguez**
VOCAL:	**David González Ortega**
SECRETARIO	**Mario Martínez Zarzuela**
SUPLENTE	**Francisco Javier Díaz Pernas**
SUPLENTE	**Mª Ángeles Pérez Juárez**

FECHA:	**Septiembre, 2018**
CALIFICACIÓN:	

Resumen de TFG

En este Trabajo Fin de Grado se ha llevado a cabo el desarrollo de una aplicación móvil que se configura desde una aplicación Web externa. Tras analizar todas las opciones disponibles tanto para el desarrollo de la aplicación móvil como para la aplicación Web, se decide llevar a cabo el desarrollo de una aplicación nativa en Android para posibilitar el aprendizaje de la asignatura universitaria *Hepatología* con la metodología educativa *Flipped Classroom*, para lograr, entre otros, aspectos como la motivación y la autonomía en el aprendizaje.

La aplicación se caracteriza por su gran flexibilidad de modo que se pueden integrar los contenidos sin necesidad de re-compilación, según se hallen habilitados externamente desde la aplicación Web desarrollada con PHP. De este modo, se puede habilitar diferente documentación, enlaces externos, algoritmos... de forma sencilla, pero también considerando las implicaciones en la seguridad. Así pues, la aplicación implementada servirá como apoyo a la docencia, pero también para la práctica clínica de hepatología pues cuenta unas calculadoras que permiten el apoyo al diagnóstico de los pacientes a partir de los resultados que con estas se obtienen.

Palabras clave

Hepatología, aplicación móvil, Android, Web, servicios Web.

Abstract

In this end-of-degree project has been carried out the development of a mobile application that is configured from an external Web application. After analyzing all the available options for the development of the mobile application and for the Web application, it was decided to carry out the development of a native application on Android for learning the university subject *Hepatology* with the *Flipped Classroom* educational methodology, for achieve, among others, aspects such as motivation and autonomy.

The application has a great flexibility so that contents can be integrated without the need for re-compilation, as they are externally enabled from the Web application developed with PHP. In this way, you can enable different documentation, external links, algorithms ... in a simple way, but also considering the implications for security. The application will serve to support teaching, but also for the clinical practice of hepatology because it has calculators that allow the support to the diagnosis of patients based on the results obtained with these.

Keywords

Hepatology, mobile application, Android, Web, Web services.

A tener en cuenta por el lector: En esta versión del tomo del trabajo de fin de grado, se han eliminado las imágenes de algunas figuras por temas de restricciones dado el carácter público de este documento. Lo mismo ocurre con algunos detalles de implementación.

Tabla de contenido

Capítulo 1: Introducción y contexto

En la actualidad el uso de la tecnología se ha extendido a todos los sectores y ámbitos de la sociedad donde se presenta como una herramienta para las diferentes actividades profesionales, pero también en el día a día cotidiano. Esto se debe a la expansión de los dispositivos móviles y al acceso generalizado a Internet con las tarifas de telefonía móvil disponibles. Se puede decir entonces, que a través de Internet se puede acceder a una cantidad ilimitada de contenidos e información (es más la información que hay, que la que se puede consumir) desde cualquier parte y en cualquier momento. Esto ha motivado la creación de numerosas aplicaciones de todo tipo para explotar las posibilidades que tantos dispositivos móviles conectados ofrecen. Alguno de esos tipos de desarrollos, son la aplicación de las nuevas tecnologías en el panorama educativo y sanitario.

Esto es lo que ha inspirado al denominado *e-learning*, para el cual se presentan a continuación algunas definiciones (Cabrero Almenara, 2006). Se conoce como *e-learning* a la formación que utiliza la red como tecnología de distribución de la información, sea esta red abierta (Internet) o cerrada (intranet). También se define como el desarrollo del proceso de formación a distancia (reglada o no reglada), basado en el uso de las tecnologías de la información y las telecomunicaciones, que posibilitan un aprendizaje interactivo, flexible y accesible, a cualquier receptor potencial. Una última definición es que *e-learning* es una enseñanza a distancia, abierta, flexible e interactiva basada en el uso de las nuevas tecnologías de la información y de la comunicación, y de las comunicaciones, y sobre todo aprovechando los medios que ofrece la red Internet.

La introducción del *e-learning* en la educación se puede apreciar con las nuevas metodologías pedagógicas seguidas cada vez en más centros educativos. Comenzando por las universidades y acabando por los centros de formación secundaria y de formación profesional, se puede comprobar que actualmente la mayoría cuentan con plataformas específicas desde las que los alumnos acceden a todos los contenidos de la asignatura proporcionados por los profesores que las imparten. Ya no sólo se trata de los propios conocimientos en forma de capítulos, sino de la posibilidad de disponer de diferentes actividades interactivas y de la colaboración entre los alumnos. De hecho, existen universidades con presencia en varios países cuya oferta educativa es completamente en línea y carece de la opción presencial.

Dentro del *e-learning* se encuentra la metodología *Flipped Classroom* que en parte ha inspirado al desarrollo del trabajo que se ha realizado. Como en el caso de *e-learning* se presentan algunas definiciones de la metodología *Flipped Classroom* (Bishop & Verleger, 2013). Invertir el aula significa que los eventos que tradicionalmente han tenido lugar dentro del aula ahora tiene lugar fuera del aula y viceversa. Sin embargo, esta sencilla definición no engloba correctamente al uso que los expertos hacen de este concepto. La etiqueta *Flipped Classroom* es a menudo utilizada para referirse a cursos que usan actividades que consisten en lecturas en video asincrónicas basadas en la web y problemas o cuestionarios cerrados. Finalmente, la definición que parece más completa define *Flipped Classroom* como la técnica educativa que consta de dos partes: actividades interactivas de aprendizaje en grupo dentro del aula, e instrucción individual directa basada en computadora fuera del aula.

Por otro lado, como se comentaba antes, el acceso a la información está abierto para todos gracias a Internet. Sin embargo, está presente el problema de la fiabilidad, rigor y calidad de la información pues la misma tecnología que proporciona acceso a la información también posibilita que cualquiera pueda escribirla. Esto, lleva en muchos casos a la confusión de los usuarios quienes más frecuentemente de lo que parece acaban dando por válida información errónea o sin calidad. Así pues, Internet ofrece un acceso sencillo a la información, pero cuando la que se busca es mínimamente especializada o debe contar con un mínimo de fiabilidad el usuario debe llevar a cabo a veces una muy tediosa tarea de contrastación y búsqueda de fuentes fiables de información. Esta es otra motivación para la creación del proyecto que aquí se describe, pues busca centralizar desde una misma aplicación toda la información de calidad necesaria para lograr el eficaz y correcto aprendizaje de sus usuarios.

Finalmente, como última motivación se encuentra el del apoyo al diagnóstico de pacientes para personal profesional del ámbito de la salud no especializado en hepatología, aunque también puede ser utilizado por profesionales especializados y estudiantes.

1.1 Objetivos

La finalidad de este trabajo de fin de grado consiste en el desarrollo de un sistema basado en Web y una aplicación móvil que permita el aprendizaje de los conocimientos de una asignatura universitaria sobre hepatología de forma que los contenidos que se muestran sean configurables de forma externa, y también, constituir una herramienta para su uso real por los profesionales de la salud para el apoyo al diagnóstico de pacientes.

A continuación, se listan los objetivos que se buscan con el desarrollo de este sistema:

- La organización de los contenidos para posibilitar la aplicación de la metodología pedagógica *Flipped Classroom* en el aprendizaje de la asignatura universitaria.
- Estudiar las tecnologías existentes para el desarrollo del sistema y seleccionar las aquellas más convenientes y que más beneficios aporten.
- El desarrollo del sistema asegurando la compatibilidad y la accesibilidad desde los dispositivos móviles de todos los potenciales usuarios.
- La incorporación de una colección de calculadoras para la obtención de resultados a partir de valores médicos introducidos.
- Garantizar que el sistema sea estable, de uso intuitivo y con contenidos que se actualizan de forma dinámica.
- Elaboración de una completa documentación sobre el uso del sistema y los aspectos técnicos relevantes para el usuario.

1.2 Estructura del documento

El documento se ha organizado en nueve capítulos que muestran de forma ordenada el proyecto HepApp y su desarrollo.

Con el primero se ubica, contextualiza e introduce el trabajo realizado, así como se explican las motivaciones que han originado al proyecto.

El siguiente capítulo estudia y analiza todas las tecnologías disponibles para el desarrollo de aplicaciones para los dispositivos móviles mostrando los aspectos favorables y negativos del uso de cada una de ellas.

El tercer capítulo, de todas las tecnologías disponibles analizadas, justifica el uso de algunas de ellas concretizando las versiones que se van a utilizar para construir el sistema.

El cuarto capítulo explica la teoría médica necesaria para la comprensión de los algoritmos médicos para el cálculo de resultados con los que cuenta el proyecto HepApp.

Los capítulos quinto y séptimo son los manuales de usuario de los dos sistemas que componen el proyecto HepApp en los que se muestran todos los casos de uso y funcionalidades apoyándose en el uso de imágenes que son capturas de pantalla de los sistemas en funcionamiento.

Los capítulos sexto y octavo tratan todas las consideraciones y aspectos técnicos del diseño e implementación de los sistemas. En ellos se describen todos los entresijos y el funcionamiento completo y detallado de estos.

Finalmente, en el capítulo noveno se muestra un estudio financiero en el que de forma justificada se explica una estimación del coste del desarrollo del sistema y también se muestra una lista de posibles mejoras futuras para el proyecto. En este, el autor del documento proporciona su opinión personal y las competencias adquiridas tras realizar el trabajo.

Capítulo 2: Tecnologías disponibles

En el presente capítulo se va a hacer un repaso de las tecnologías Web y las tecnologías móviles más conocidas, utilizadas y expandidas.

El uso de los dispositivos móviles no ha dejado de crecer en los últimos años. De hecho, en la actualidad, es mayor el número de dispositivos móviles en uso que el número de dispositivos de escritorio o sobremesa (StatCounter, 2018). Observando los datos de uso de cada tipo de dispositivo presentados en la figura Figura 1, desde hace unos años atrás se puede comprobar que la tendencia es el crecimiento de los dispositivos móviles y la caída de los dispositivos de escritorio.

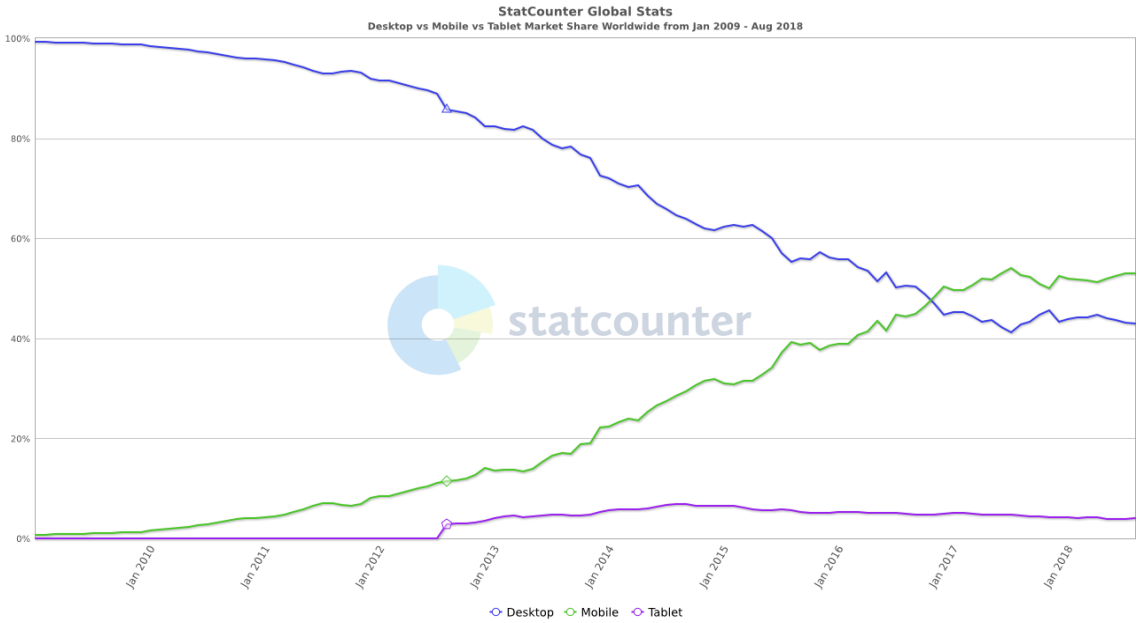

Figura 1. Cuota de mercado de los tipos de dispositivos.

Dentro del mercado de los dispositivos móviles, la existencia de varias plataformas móviles algunas con una cuota de mercado relevante, a la hora de desarrollar una aplicación, conlleva al problema de necesitar adaptar las aplicaciones a varias de estas para cubrir y llegar al mayor número de usuarios posible. Sin embargo, desarrollar y adaptar una aplicación a las diferentes plataformas disponibles puede llegar a suponer un alto coste. Como se puede ver en la tabla Tabla 1, el mercado de los dispositivos móviles se encuentra dominado por las plataformas Android (85.9%) y iOS (14.1%) (Costello & Hippold, 2018). Es por ello que el proyecto *HepApp* cuenta con una versión para Android que se trata en este documento y una versión para iOS.

Worldwide Smartphone Sales to End Users by Operating System in 1Q18 (Thousands of Units)

Operating System	1Q18 Units	1Q18 Market Share (%)	1Q17 Units	1Q17 Market Share (%)
Android	329,313.9	85.9	325,900.9	86.1
iOS	54,058.9	14.1	51,992.5	13.7
Other OS	131.1	0.0	607.3	0.2
Total	**383,503.9**	**100.0**	**378,500.6**	**100.0**

Tabla 1. Cuota de mercado de los sistemas operativos de los dispositivos móviles.

En el desarrollo de sistemas software, además de las plataformas móviles ya mencionadas (aplicaciones nativas), otras soluciones o alternativas son las aplicaciones Web y las aplicaciones híbridas. Por otro lado, también es frecuente la combinación de estas alternativas como es el caso de una aplicación Web que puede proporcionar un servicio que puede ser utilizado y consumido por una aplicación nativa. Todas estas alternativas se ven con más detalle en los subapartados que vienen a continuación.

2.1 Aplicaciones Web

Se trata de aplicaciones que se ejecutan en máquinas servidoras remotas y a las cuales se accede en muchos casos mediante un navegador Web, en ocasiones, también denominado agente de usuario. Estas aplicaciones se definen realmente listando y describiendo sus características que a veces son ventajas y otras inconvenientes (Sin et al, 2012; Delía et al, 2013; Charland & LeRoux, 2011; Luján-Mora, 2002; Luján-Mora, 2001.

Estas aplicaciones no requieren ser instaladas (por el cliente) y por ello también ahorra la instalación de todo el software del que pudiera depender la aplicación. Relacionado con esto, estas aplicaciones se actualizan de forma “transparente” al usuario (es tarea del desarrollador gestionarlo) quién siempre que accede a la aplicación puede disfrutar la última versión disponible. Esto se debe a que el cliente actúa sólo como un mero visualizador pues la aplicación se ejecuta en máquinas remotas.

Por otro lado, las tecnologías Web con las que se desarrollan estas aplicaciones son estándares y esto evita la necesidad del desarrollador de tener que adaptar la aplicación para cada plataforma y sistema operativo (son multiplataforma). En referencia a esto, se encuentra el problema de la compatibilidad de los navegadores con los estándares Web y por ello la necesidad de adaptar el desarrollo de estas aplicaciones no a los sistemas operativos, pero sí a los diferentes y principales clientes Web (navegadores). También, es habitual que su desarrollo se adapte para los diferentes anchos de pantalla de los dispositivos que ejecutan los clientes Web que acceden a la aplicación. Para esto se utiliza lo que se llama el diseño *responsive* (uso específico de la tecnología *CSS*), que permite que la visualización de la aplicación se adapte de forma automática al ancho de la pantalla disponible.

Estas aplicaciones no son distribuidas mediante ninguna tienda de aplicaciones por lo que, aunque se puede perder la capacidad de marketing y promoción que estas a veces ofrecen, a cambio no cuentan con ningún tipo de restricción o no deben pasar ningún proceso de revisión y auditoría para ser publicadas.

Son las aplicaciones que ofrecen el menor rendimiento de entre todas las alternativas. Los usuarios pueden perder interés si la aplicación por costumbre no cuenta con la capacidad adecuada que se traduce en un rendimiento mínimo esperable. El rendimiento de estas aplicaciones ya no sólo depende de la capacidad de las máquinas servidoras en las que se ejecuta la aplicación que ofrecen, sino que depende en gran medida de la conectividad. Se puede decir que este es un factor delicado en el uso de estas aplicaciones por la fuerte dependencia con dicho factor. Es por ello que la conectividad también debe ser la adecuada (en términos de rendimiento) por la parte que corresponde a las máquinas servidoras. Un problema con la conectividad puede mermar el rendimiento de la aplicación o en caso de corte de la conectividad quedar la aplicación inaccesible. Se agrava más el problema cuando los problemas de conectividad proceden por la parte del usuario ya que esto es algo no controlable para el desarrollador. En la práctica, por suerte para las aplicaciones Web, es cada vez más habitual el uso de máquinas servidoras bastante capaces, la conectividad a través de Internet de alto rendimiento y no son tan habituales las pérdidas de conectividad que impiden el acceso a la aplicación. Así pues, el rendimiento ofrecido por estas aplicaciones, aunque siempre será inferior, cada vez es más habitual que sea alto y la diferencia con los otros tipos de aplicaciones se reduzca, al menos lo suficiente como para que compense con creces por disfrutar todas las ventajas propias de estas aplicaciones. A parte, a favor frente a otros tipos de aplicaciones, puede ocurrir que los otros tipos de aplicaciones en ocasiones puedan estar tan ligadas a la conectividad (con todas sus consecuencias) como las aplicaciones Web.

Las aplicaciones Web cuentan con limitaciones en lo que al uso del hardware disponible de los dispositivos de los clientes se refiere. Estas aplicaciones sólo pueden hacer uso de las funcionalidades que ofrece el cliente Web o navegador. Por suerte, son cada vez más las funcionalidades que ofrecen los navegadores que acceden a nuevo hardware. En cualquier caso, el uso de hardware como el GPS, cámara y acelerómetro entre otros sigue siendo inaccesible o parcialmente restringido para estas aplicaciones. De nuevo, a favor de las aplicaciones Web, muchas de estas aplicaciones no necesitan hacer uso de hardware (de los dispositivos de los clientes) no disponible.

Estas aplicaciones no consumen los recursos de los dispositivos de los clientes, sobre todo en cuanto a capacidad de cálculo, uso de memoria, consumo de energía y espacio en disco se refiere.

Es también importante que estas aplicaciones son accesibles desde cualquier dispositivo, es decir, se pueden considerar portables o independientes del dispositivo desde el que se accede. Como se ha dicho, la única restricción es la disponibilidad de conexión a Internet.

Las aplicaciones suelen contar con una correcta disponibilidad al añadir redundancia de máquinas servidoras y sus accesos a Internet de forma que el servicio se ofrece desde múltiples localizaciones distintas. Como se ha indicado, esto sigue sin resolver el problema de la pérdida de acceso a Internet por parte del cliente.

Por el hecho de estar centralizado el acceso a la aplicación por el cliente, esto proporciona la posibilidad del uso colaborativo entre los usuarios. Un par de ejemplos son los calendarios o agendas en línea y los repositorios en el desarrollo software.

Desde el punto de vista de la seguridad, al ser tarea del desarrollador la instalación, configuración, gestión y mantenimiento de la aplicación y los equipos que la sirven, el uso de estas aplicaciones en ocasiones puede ser más seguro porque está evitando que el cliente, que podría no contar con todos los conocimientos técnicos necesarios y la suficiente concienciación en materia de seguridad, realice una mala instalación, realice una combinación errónea de configuraciones o no se encargue de instalar las actualizaciones lo que pudiera suponer un problema de seguridad. Se puede decir que se están restringiendo posibilidades al cliente que, aunque puede perjudicar a algunos usuarios más expertos, por lo general ofrece una mayor seguridad general a la mayoría de los usuarios. También se evita el problema de que para un usuario concreto la seguridad de la aplicación sea tan buena como la del uso que dicho cliente haga del sistema operativo del dispositivo que usa para ejecutar la aplicación. En definitiva, estas aplicaciones no se ven tan afectadas por el malware que pueda existir en los dispositivos de los clientes. También, en referente a este tema, por el contrario para las aplicaciones Web, el hecho de que supongan un servicio centralizado para muchos usuarios, puede aumentar el interés de los usuarios malintencionados que pueden internar dar con una vulnerabilidad del sistema para explotarla después en respuesta a sus intereses.

Finalmente, las tecnologías más habituales para el desarrollo de estas aplicaciones son *HTML*, *CSS* y *JavaScript* para la parte cliente (*frontend*), y *Java (JSP)*, *Python*, *ASP*, *PHP*, y servidores y sistemas gestores de bases de datos (*MySQL*, *Oracle*, *Microsoft SQL Server*, etc.) para la parte servidora (*backend*). También es frecuente el uso de *framework* basados en estas tecnologías en ambos lados (cliente y servidor) y el uso de otras tecnologías (más bien cierto uso concreto de las ya existentes) como el diseño *responsive* y *AJAX*.

2.2 Aplicaciones móviles nativas

Las aplicaciones nativas son aquellas que son específicas de una plataforma o propias para un sistema operativo. Frente a las aplicaciones Web, estas son las que ofrecen el máximo rendimiento, cuentan acceso a todo el hardware disponible en el dispositivo del cliente en el que se ejecutan y la dependencia de la conectividad a Internet puede ser opcional (según el tipo de aplicación). También, pueden hacer uso de los elementos gráficos propios de la plataforma para la que se han hecho y pueden proporcionar una mejor experiencia de uso (como consecuencia también del buen rendimiento) (Delía et al, 2013; Charland & LeRoux, 2011).

Su distribución (al ser aplicaciones móviles) se realiza a través de la tienda de aplicaciones de la plataforma para la que se han desarrollado que, si bien puede ser algo positivo por permitir explotar las posibilidades de marketing y promoción que estas ofrecen, también se está limitando la libertad de publicación ya que deben pasar un proceso de revisión o auditoría para poder ser publicadas en estas tiendas. Además, ligado al uso de las tiendas de aplicaciones puede existir un coste de licencia de uso de las mismas para la publicación y distribución de las

aplicaciones desarrolladas. El problema de las libertades de publicación mediante las tiendas de aplicaciones se puede ver reducido si la aplicación cuenta con un sitio Web oficial desde el que se puede descargar el fichero de instalación de la aplicación de forma directa. Esta práctica considerada bastante insegura para los clientes no conlleva ningún riesgo siempre que la aplicación en caso de obtenerse desde fuera de las tiendas de aplicaciones se haga desde los sitios Web oficiales. El motivo por el que esto es considerado bastante inseguro es debido a que el cliente puede no ser capaz de distinguir el sitio Web oficial y acabar descargando la aplicación premeditadamente modificada, lo que es a efectos prácticos malware.

Por otro lado, otras características que definen a estas aplicaciones son la necesidad de la compilación del código fuente para la plataforma para la que se ha desarrollado la aplicación como si de aplicaciones para equipos de escritorio se tratara. En relación con el desarrollo, la gran desventaja es el enorme coste que tiene porque este se debe realizar una vez por cada plataforma, ya que cada una cuenta con su propio lenguaje de programación. Como consecuencia de los lenguajes propios de cada plataforma, un problema añadido del desarrollador es que este seguramente conozca y se haya especializado más en uno de ellos en lugar de conocer con la suficiente destreza todos. El perfil de un desarrollador suficientemente especializado en cada uno de los lenguajes de las diferentes plataformas para las que se quiere lanzar la aplicación es escaso y es necesario a veces tener que recurrir a más de un desarrollador. En el caso de las tecnologías Web este problema, aunque presente, está mucho más reducido porque el uso de estas tecnologías está mucho más extendido y es más habitual que sean conocidos por los desarrolladores. Para reducir este problema de los diferentes lenguajes de programación en el desarrollo de aplicaciones nativas existe *Xamarin* que propone que se haga el desarrollo una única vez en un lenguaje determinado y que después a partir de este se puedan generar instalables para varias plataformas (Xamarin, 2018). El uso de herramientas como *Xamarin* o *PencilCase* permiten crear lo que se conoce como aplicaciones generadas o falsas nativas. Las aplicaciones generadas con este tipo de herramientas presentan unas ventajas e inconvenientes, pero esto no es objeto de estudio en el presente documento.

Realmente ya no es sólo el coste de desarrollo inicial sino también el del posterior mantenimiento y actualización de lo desarrollado.

Desde el enfoque de la seguridad y en contrapartida a lo explicado antes sobre las aplicaciones Web, la seguridad puede ser un inconveniente porque los usuarios pueden no preocuparse de instalar todas las actualizaciones disponibles tanto del sistema operativo como de las aplicaciones instaladas.

A continuación, se tratan las dos plataformas que en base a sus cuotas de mercado del mercado de los dispositivos móviles se consideran suficientemente relevantes como para estudiarse.

2.2.1 Android

Es un sistema operativo de código abierto dirigido a dispositivos móviles basado en el núcleo Linux modificado que en la actualidad es desarrollado y mantenido por Google (*Alphabeat*).

Para el desarrollo en Android se cuenta con el *Software Development Kit* (*SDK*) y el *Native Development Kit* (*NDK*) (Android Developers, 2018). Lo habitual y la recomendación oficial es utilizar siempre el *SDK* pues aparte de proporcionar el *Entorno de Desarrollo Integrado* (*IDE*) llamado *Android Studio* junto al *Emulador Android* para utilizar este se utiliza el lenguaje de programación *Java* o el recientemente nuevo lenguaje con soporte oficial para Android *Kotlin*. En ejecución, las aplicaciones desarrolladas empleado el *SDK* se ejecutan sobre una máquina virtual *Java* denominada *ART* (la máquina virtual *Dalvik* fue su predecesora en versiones ya bastante antiguas de Android). Con *SDK* se puede hacer uso de las API (*Application Programming Interface*) que ofrecen todas las funcionalidades necesarias a las aplicaciones. Debido a que el código desarrollado con lenguajes como *Java* o *Kotlin* se ejecuta sobre una máquina virtual *Java*, se obtienen algunas ventajas como la portabilidad a pesar de la arquitectura del microprocesador, la gestión automática de memoria y la posibilidad de uso de un extenso conjunto de bibliotecas disponibles. Dentro del desarrollo con *SDK*, se puede emplear *Java* y *Kotlin*. La ventaja con *Java* es que al disponer de más recorrido es actualmente ampliamente conocido y utilizado por muchos desarrolladores. Sin embargo, *Kotlin* se presenta como una alternativa suficientemente beneficiosa como para que se espere que los desarrolladores acaben molestándose en conocer su sintaxis y cambien su preferencia de desarrollo de *Java* a este lenguaje.

Algunas de las ventajas de *Kotlin* frente a *Java* son que es conciso a la vez que expresivo, cuenta con características que lo hacen más seguro de forma que permite el desarrollo de aplicaciones más sanas y de mayor rendimiento por defecto, permite el uso de expresiones lambda (punteros a funciones), permite el uso de valores por defecto para los argumentos de los métodos, permite escribir el código sin utilizar tanto código no necesario y sin código repetitivo, permite la extensión de funcionalidades sin implicar esto el uso del mecanismo de la herencia y es totalmente interoperable con *Java* de forma que se puede llamar código escrito en un lenguaje desde el otro.

Respecto al uso de *NDK*, este está pensado para poder portar a Android desarrollos ya hechos en lenguajes como *C/C++* incorporando este código a la aplicación Android a partir de la *Java Native Interface* (*JNI*). El código desarrollado con *NDK* se ejecutará proporcionando un mayor rendimiento ya que en lugar de ejecutarse a través d una máquina virtual *Java* se ejecuta directamente en el procesador. El uso de *NDK* sólo debe efectuarse en situaciones verdaderamente necesarias y justificadas ya que el desarrollo de la aplicación puede adquirir una gran y excesiva complejidad y verse limitado en funcionalidades. La parte positiva del desarrollo mediante *NDK* es la relativa facilidad para el desarrollo de una aplicación multiplataforma pues en esencia las aplicaciones para iOS y Windows se desarrollan con *C++* y este código se puede portar sin demasiada complejidad. Lo importante es que el uso de *SDK* se puede complementar con el uso de *NDK* ya que algunas aplicaciones podrían hacer puntualmente uso de *NDK* para el desarrollo de una funcionalidad muy específica. Un ejemplo del uso de *NDK* podría ser para tareas de cálculo intensivo como ocurre en videojuegos, simulaciones o el procesado de señales. También, el uso de *NDK* puede ser útil cuando se cuenta

con una biblioteca ya desarrollada en *C++* que no existe en *Java* y se desea utilizar en el desarrollo de la aplicación. Resumiendo, el uso de *NDK* permite crear funcionalidades específicas por necesidades de rendimiento o de reutilización de código de forma que complementa en ocasiones al desarrollo con *SDK*.

Por otro lado, existe una amplia y muy buena documentación oficial para el desarrollo de aplicaciones en Android (utilizando *SDK*) que entre otros explica de forma detallada toda la API Android y proporciona en ocasiones guías y ejemplos muy útiles.

En esta plataforma la distribución de las aplicaciones se realiza mediante su tienda propia de aplicaciones oficial llamada *Google Play*. Existen otras tiendas de aplicaciones, algunas propias de un fabricante de dispositivos y otras altamente no recomendables pues pueden contener abundante malware. Como ya se ha dicho, a cuenta y riesgo propio, un usuario siempre puede instalar aplicaciones obtenidas de fuera de la tienda de aplicaciones a partir del fichero en formato *APK* de la aplicación. Esto no es un problema siempre que el fichero se obtenga de una fuente fiable, pero generalmente esta práctica no se recomienda dado que es la forma más habitual de infectar un dispositivo con malware. En la figura Figura 2 se puede observar la evolución de la cantidad de aplicaciones publicadas en los últimos años en la tienda de aplicaciones oficial y en la figura Figura 3 la evolución de descargas de aplicaciones desde esta (Statista, 2018).

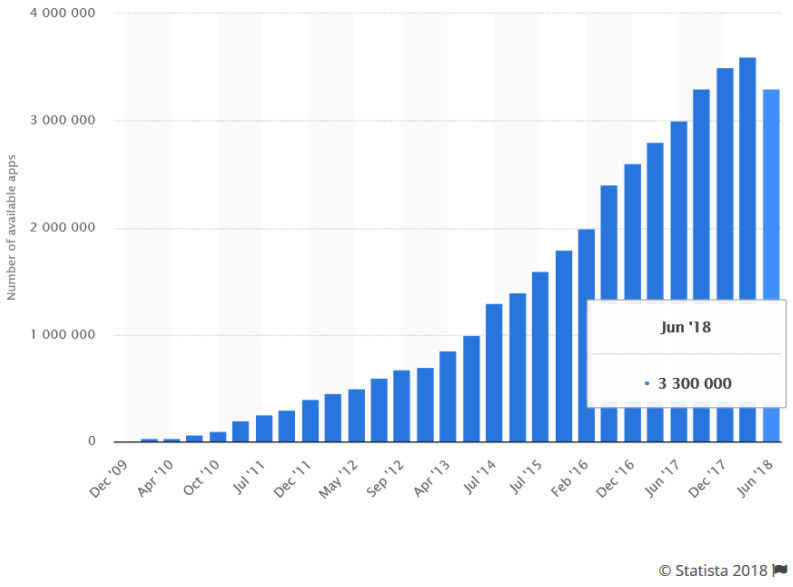

Figura 2. Evolución del número de aplicaciones publicadas en Google Play.

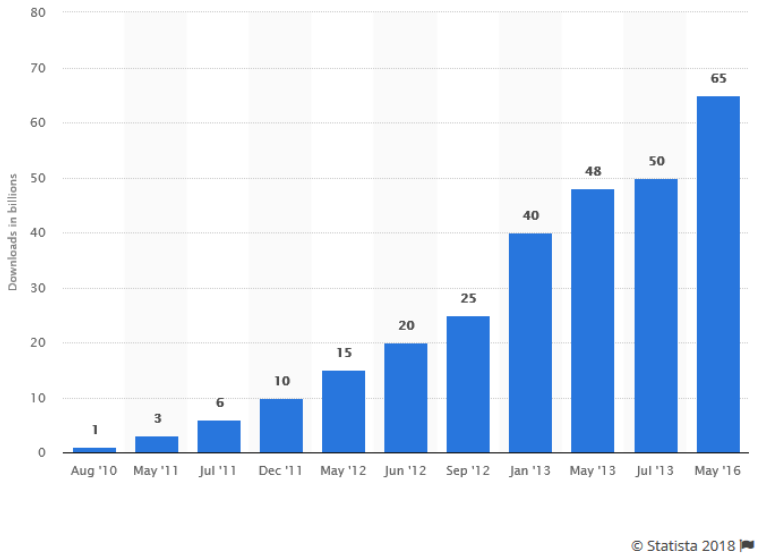

Figura 3. Evolución del número de descargas de aplicaciones desde Google Play.

Para la publicación de una aplicación en *Google Play* lo único que se necesita es adquirir una licencia de desarrollador realizando un pago único (licencia vitalicia) de 25$. Al solicitar publicar una aplicación en *Google Play* la aplicación pasa por un proceso de revisión y auditoría para asegurar la seguridad y calidad de la aplicación que se pretende publicar. Esto no impide que alguna aplicación publicada en esta tienda de aplicaciones pueda contener malware, pero si reduce enormemente el riesgo y es mucho más rápidamente detectado y eliminado. El sistema *Google Play Protect* se trata de un servicio integrado que encarga de analizar grandes cantidades de aplicaciones con la intención de detectar aquellas que sean maliciosas para poder eliminarlas de la tienda de aplicaciones. Utiliza algoritmos de aprendizaje automático (inteligencia artificial) para mejorar constantemente.

Uno de los principales problemas que Android arrastra de forma histórica es el de la fragmentación. La fragmentación es la existencia de cantidades relevantes de dispositivos en uso con versiones antiguas e incluso obsoletas de Android que no pueden ser actualizados porque carecen del soporte del fabricante para la adaptación de las nuevas versiones de Android para todos esos dispositivos. En la figura Figura 4 se muestra el estado de la fragmentación a finales de julio del año 2018. Los fabricantes, tras cierto tiempo desde el lanzamiento de un dispositivo deciden dejar de darle soporte para lanzar nuevos modelos de dispositivos actualizados a las nuevas necesidades. Consideran costoso y un lastre mantener una gran familia de dispositivos. La fragmentación si como efecto adverso bien impide poder disfrutar de las nuevas funcionalidades de las nuevas versiones y con el tiempo impide el uso de varias aplicaciones que requieren de una mayor versión para funcionar, sobre todo son un gran problema de seguridad dada la cantidad de parches de vulnerabilidades que se distribuye desde hace unos años de forma mensual y por no poder contar con los cambios de arquitectura y nuevos sistemas de seguridad implementados en cada una de las versiones más recientes.

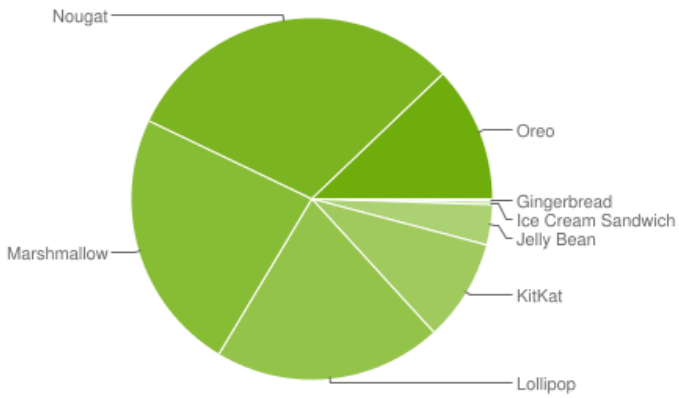

Figura 4. Fragmentación presente en Android a finales de julio del 2018.

Algunos de los intentos, se ha de decir que cada vez más acertados, para solucionar la fragmentación en Android son el sistema de distribución de los parches de seguridad mensuales, la extracción como aplicaciones individuales de características ofrecidas hasta el momento directamente por el sistema operativo de forma que se puedan actualizar desde *Google Play* y el proyecto *Treble* introducido con *Android Oreo* que rediseña la arquitectura del *famework* del sistema operativo Android de forma que ayuda a separar la implementación del fabricante del *framework* de Android a través de una nueva interfaz (AOSP, 2018). En la figura Figura 5 se muestra a la izquierda el proceso de actualización para versiones de Android igual o inferiores a la siete, y a la derecha el proceso de actualización para versiones de Android igual o superiores a la ocho.

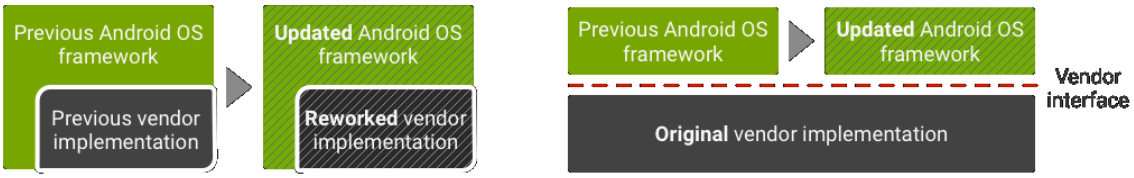

Figura 5. Entorno de actualización de Android antes y después del proyecto Treble.

Aplicando el proyecto *Treble* se ha logrado que se pueda sustituir (actualizar) el *framework* de Android sin tener que recompilar la capa de abstracción del hardware (*HAL*). Dicho de otro modo, los fabricantes pueden volver a utilizar la implementación que ya hicieron en el momento de actualizar un dispositivo a una nueva versión. De esta forma se espera que el problema de la fragmentación se resuelva en dispositivos que de serie incorporen una versión de Android igual o superior a la versión ocho.

Otro aspecto relacionado con la seguridad, es que el sistema operativo desde la versión cinco ha ido introduciendo versión tras versión cada vez nuevos y mejores mecanismos de seguridad más avanzados de forma que las últimas versiones disponibles cuentan con un buen nivel de seguridad. Algunos de estos mecanismos son *Verified Boot*, los nuevos esquemas de firmado de aplicaciones, el cifrado de las particiones de datos de los usuarios, el sistema anti robo (mejora de la pantalla de bloqueo o *lockscreen*, el bloqueo del *bootloader* y una partición resistente al restablecimiento de fábrica para alojar las cuentas de Google desde las que se ha efectuado exitosamente el proceso de autenticación) y el aislamiento de la ejecución de las aplicaciones (*sandboxing*) entre otros. Por decirlo de alguna forma, “el precio” de todas estas significativas mejoras en materia de seguridad ha supuesto un poco la pérdida de la libertad

(cierre del sistema) que tanto caracterizó al sistema operativo desde sus inicios. Un síntoma de ello, es la pérdida del interés de los desarrolladores de aplicaciones con funcionalidades avanzadas haciendo uso del usuario *root* (cuenta de usuario con privilegios totales de administrador sobre el sistema operativo) de Android debido a la gran dificultad para obtener procedimientos para desbloquear esta cuenta de usuario en cada uno de los muchos modelos de dispositivos disponibles en el mercado. Otro aspecto positivo de todos estos nuevos sistemas es que han permitido el desarrollo de nuevas aplicaciones al cumplir las requeridas exigencias en materia de seguridad para poder funcionar, como es el caso de las aplicaciones de banca móvil.

Finalmente, respecto al mencionado *IDE* para Android se debe decir que este puede instalarse en cualquier sistema operativo, es decir, no es un *IDE* exclusivo para una plataforma concreta, lo cual permite el desarrollo de las aplicaciones para Android sin ningún tipo de restricción que implique la compra de equipamiento específico. Como alternativa al *IDE Android Studio* también existe el *IDE Eclipse*.

2.2.2 iOS

Es un sistema operativo de código cerrado derivado de *UNIX* (se basa en *Darwin BSD*) dirigido a dispositivos móviles *iPhone* y *iPad* de la empresa *Apple* que en la actualidad es desarrollado y mantenido por dicha empresa. Se lanza cada año una nueva versión de iOS y su versión actual es la versión once.

Para el desarrollo en iOS se cuenta con un *SDK*, el *IDE Xcode* y un simulador de *iPhone* para probar los desarrollos realizados (Apple Developer, 2018). A diferencia de como ocurre con Android, este *IDE* sólo puede instalarse y ejecutarse en equipos de la propia marca por lo que desarrollar aplicaciones para iOS implica disponer de uno de estos equipos (considerando su coste). Para el desarrollo en iOS se utiliza el lenguaje de programación *Swift* que es el recomendado o el lenguaje *Objective-C* que no se considera muy amigable o cómodo de utilizar. El desarrollo en estos lenguajes tiene muchas similitudes o equivalencias con el lenguaje de programación *C*. En el desarrollo se cuenta con una buena comunidad de programadores, lo cual puede ayudar a la entrada de un nuevo desarrollador en esta plataforma. Por otro lado, en este caso también hay disponible abundante documentación para su consulta a la hora del desarrollo.

Respecto a la arquitectura de iOS esta, como se comprueba en la figura Figura 6, se compone de varias capas.

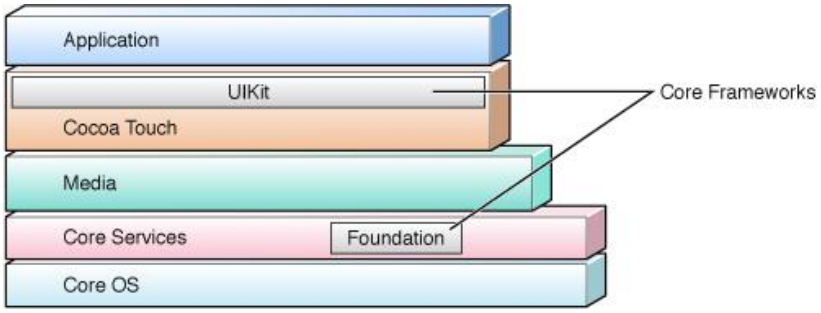

Figura 6. Arquitectura de iOS.

De forma breve se explican algunas de las capas:

- ***Core OS***: Contiene el núcleo, el sistema de ficheros, la infraestructura de red, seguridad, gestión de energía y los *driver*. También incluye las API de nivel de sistema para diferentes servicios.
- ***Core Services***: Proporciona servicios del sistema como la manipulación de cadenas, la gestión de colecciones, utilidades para direcciones URL, manejo de la red, gestión de contactos, preferencias, persistencia y el uso de hardware como GPS y acelerómetro, entre otros.
- ***Media***: Proporcionan servicios relacionados con los gráficos y la multimedia. Algunos de estos servicios son *Core Graphics*, *Core Text*, *OpenGL ES*, *Core Animation*, *AVFoundation* y *Core Audio* entre otros.
- ***Cocoa Touch***: Cuenta con los *framework* necesarios para el desarrollo de aplicaciones nativas en iOS. Cuenta con todos los objetos disponibles para la creación de las interfaces gráficas de las aplicaciones y define estructura para el comportamiento de las aplicaciones que se desarrollen.

Al igual que ocurre con Android, en iOS se dispone de una tienda de aplicaciones propia de nombre *App Store*. En este caso, la licencia para la publicación de aplicaciones, con un coste de 99$, es más cara y se debe renovar anualmente. Los procesos de revisión y auditoría para la publicación de aplicaciones en esta tienda son más estrictos de forma que se asegura una calidad y seguridad de las aplicaciones. En la figura Figura 7 se puede observar la evolución de la publicación de aplicaciones en la tienda *App Store* (Statista, 2018). En la figura Figura 8 se puede ver la evolución de la cantidad de descargas de aplicaciones desde la tienda.

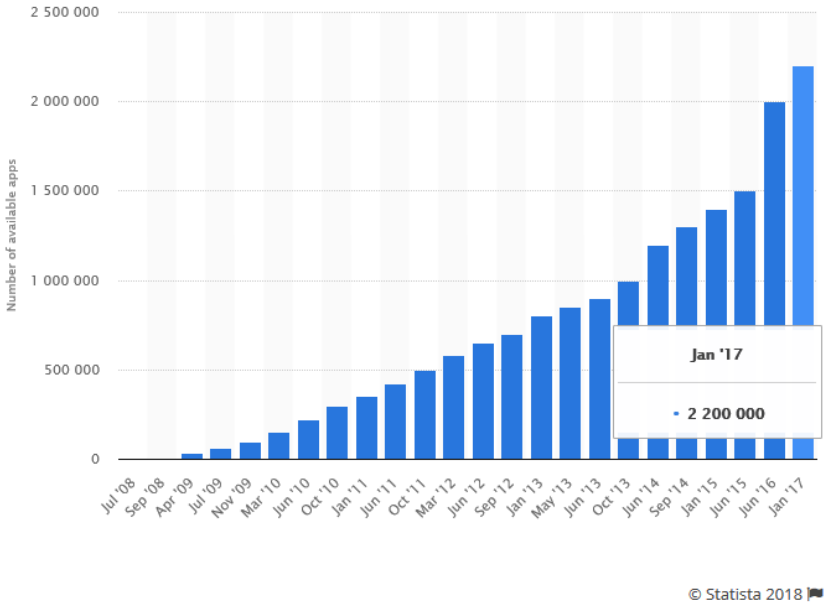

Figura 7. Evolución de la publicación de aplicaciones en la App Store.

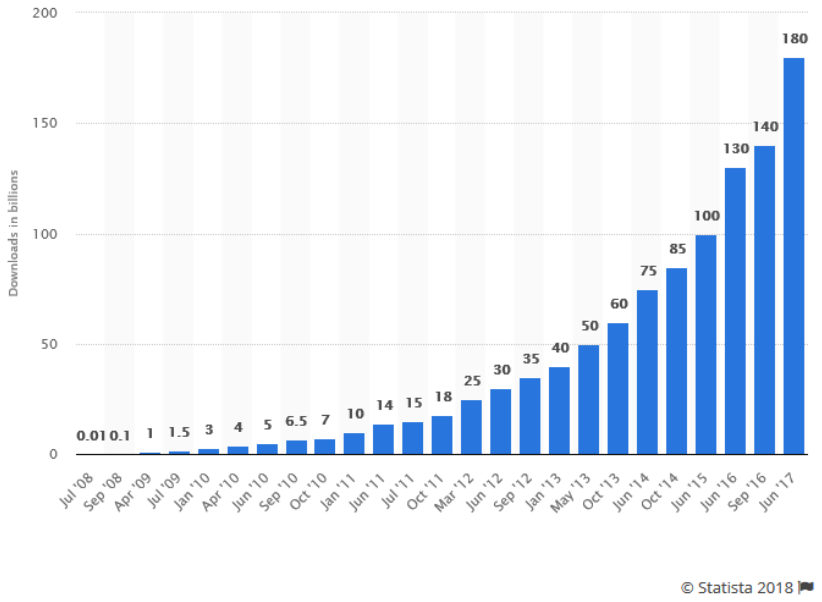

Figura 8. Evolución de las descargas de aplicaciones desde la App Store.

A diferencia de lo que ocurre con Android, en iOS no hay problemas de fragmentación debido a que el único fabricante de los dispositivos es también quién se encarga de desarrollar el sistema operativo. Por ello, el catálogo de dispositivos iOS es inmensamente mucho más reducido que el de Android. Esto permite que todos los dispositivos del catálogo puedan recibir de forma mucho más fácil y rápida las actualizaciones de una nueva versión del sistema operativo que lleva a que no exista fragmentación (existe, pero no se considera al compararlo con Android). Además, desde el punto de vista del desarrollo, el disponer de un número bastante reducido de diferentes dispositivos (en comparación con Android) facilita enormemente el desarrollo de aplicaciones pues no se deben considerar tantas particularidades de diferentes dispositivos. De hecho, se ha mantenido la relación alto-ancho de las pantallas en los diferentes

dispositivos del catálogo lo que simplifica enormemente tener que adaptar las aplicaciones a las diferentes pantallas.

Relacionado con lo anterior y a diferencia con Android, también se cuenta con la ventaja de que se debe desarrollar para una única arquitectura de microprocesador lo cual simplifica las cosas y tiene un ligero impacto positivo en el rendimiento.

Respecto a toda la filosofía que engloba a iOS y sus dispositivos, se trata de una plataforma mucho más cerrada desde el punto de vista de libertad de posibilidades para sus usuarios y los desarrolladores a pesar de la existencia del *JailBreak* que se puede considerar como el análogo al *root* en Android.

Otra de las ventajas sobre otros sistemas móviles es una mayor integración del hardware y el software por encargarse la misma empresa de ambas partes. En iOS tampoco está presente el problema del *bloatware* que en el caso de Android especialmente en los dispositivos de algunos fabricantes podría considerarse un problema serio ya no por la gran cantidad de aplicaciones preinstaladas que el usuario no necesita y consumen los recursos del sistema, sino que por la dificultad y en ocasiones imposibilidad de su inhabilitación o desinstalación. Finalmente, también se puede destacar el tema del soporte técnico que es bastante superior en el caso de iOS. En Android, existen algunos fabricantes que ofrecen un servicio de soporte técnico tan malo como bueno lo ofrecen otros fabricantes distintos.

2.3 Aplicaciones híbridas

Como su propio nombre indica, pretenden tomar los aspectos positivos de las aplicaciones nativas y de las aplicaciones Web. Este tipo de aplicaciones hacen uso de las tecnologías Web de la parte cliente (*frontend*) que son multiplataforma y multipantalla pero al mismo tiempo también puede acceder a gran parte del hardware del sistema del dispositivo sobre el que se ejecutan. Esto se debe a que se desarrollan con las tecnologías Web pero luego se ejecutan de forma nativa en el dispositivo haciendo uso de un contenedor. Es también este mismo contenedor el que en ocasiones permite hacer uso de las características nativas abstrayendo a la aplicación de la plataforma propia del dispositivo (ofrece una API que se utiliza con *JavaScript*) (PhoneGap, 2018; Apache Cordova, 2018; Delía et al, 2013).

Es frecuente que hagan uso de un cliente Web (navegador) incrustado de forma que con un diseño correcto el usuario podría no llegar a percibir qué pantallas se muestran mediante accesos Web y cuáles de forma nativa.

Estas aplicaciones se ejecutan de forma nativa (aunque sea debido al contenedor) y como tal pueden hacer uso de las tiendas de aplicaciones aprovechando las ventajas que estas ofrecen en posibilidades de distribución y promoción.

Sin embargo, se puede ver como una desventaja el no utilizar (aunque sea posible) la apariencia propia de cada plataforma, sino que se utiliza una apariencia común para todas las plataformas un poco al estilo de lo que pasa con las aplicaciones Web precisamente para que la apariencia sea coherente con estas pues pueden realizarse accesos Web.

En el caso de necesitar construir una aplicación compleja, el uso de las soluciones híbridas como *Apache Cordova* o *PhoneGap* pueden impedir contar con un rendimiento adecuado.

Capítulo 3: Descripción del problema y tecnologías utilizadas

En este capítulo primero se va a presentar el problema que se va a resolver y que ha dado origen a este proyecto, y después se van a comentar las tecnologías que se han decidido utilizar para desarrollar el sistema que va a solucionar el problema.

3.1 Descripción del problema

El proyecto HepApp ya existía en el momento del inicio del proyecto que se trata en este documento. Como situación de partida ya se contaba con una implementación nativa hecha para Android y una implementación no nativa hecha con *PencilCase* para iOS. De forma paralela a este trabajo se ha desarrollado una implementación nativa para iOS. Por otro lado, en el caso de Android se cuenta con otra aplicación complementaria llamada *DoctorCase* que implementa unas calculadoras y algoritmos médicos para el apoyo al diagnóstico ya que permiten obtener los resultados a partir de la entrada de los valores de parámetros médicos de un paciente.

Todas las implementaciones existentes en el momento del inicio de este proyecto contaban con el contenido de la aplicación precargado por lo que cualquier cambio sobre el contenido por pequeño que fuera implicaba que la aplicación fuera editada, compilada, publicada en la tienda de aplicaciones y descargada por todos los usuarios. A parte, esto provoca una fuerte dependencia con el desarrollador de las aplicaciones pues este es quien se debe hacer cargo de todo este proceso cada vez que se requiera hacer un cambio del contenido.

Por otro lado, algunas de las implementaciones existentes necesitaban ser actualizadas, revisadas y depuradas para corregir algunos problemas y optimizar el funcionamiento logrando mejorar el rendimiento, uso de memoria, el consumo energético y el consumo de datos.

Finalmente, en el caso de Android era necesario unificar las implementaciones existentes de la aplicación HepApp y *DoctorCase* como una única aplicación llevando las funcionalidades de la segunda a la primera.

3.2 Solución planteada

La solución que se plantea para resolver el problema indicado en el subapartado anterior es el de crear una aplicación Web que va a actuar a modo de servicio Web para una aplicación implementada de forma nativa en Android y que ambas se comunican a través de Internet siguiendo una arquitectura cliente-servidor. El desarrollo de ambos sistemas se ha llevado a cabo comenzando desde cero. En el caso de la aplicación Web porque no existe ya una implementación de partida, y en el caso de la aplicación Android porque para que su contenido se cargue dinámicamente a partir de los datos obtenidos del servicio Web a través de Internet requiere que el diseño de la aplicación sea totalmente distinto al que ya existía pues en esta ocasión la aplicación debe ser muy dinámica y flexible, ya que son muchas más cosas las que se desconocen de antemano.

Llevando a cabo el desarrollo de la aplicación Android desde cero con el suficiente buen diseño e implementación se puede lograr que el funcionamiento y rendimiento sea el adecuado y la aplicación se encuentre ciertamente optimizada.

Aprovechando la implementación de esta nueva aplicación se han unificado todas las funcionalidades en esta de forma que ya no es necesario seguir contando con dos aplicaciones por separado.

Para el desarrollo de la aplicación Web que va a actuar como servicio Web se ha decidido utilizar las tecnologías Web del lado del cliente (*frontend*) *HTML 5, CSS 3* y *JavaScript* junto con el *framework Bootstrap 4* y la biblioteca *JQuery 3.3.1*. Para el lado del servidor (*backend*) se han utilizado las tecnologías *PHP 7.2.6* junto al sistema gestor de bases de datos *MariaDB 10* (derivado de *MySQL*).

Para el desarrollo de la aplicación se ha optado por Android utilizando el *IDE Android Studio 3* (junto con el *Emulador Android*), utilizando el *SDK* de esta plataforma mediante *Java*. La versión mínima requerida para la aplicación es *Android 5.1 Lollipop*.

A continuación, se describen más detalles y las decisiones de uso de estas tecnologías.

3.3 Tecnologías Web frontend: HTML 5, CSS 3 y JavaScript

En lo que las tecnologías del lado cliente (*frontend*) para el desarrollo de la aplicación Web se refiere, no hay otras alternativas o tecnologías a poder utilizar. Respecto a las versiones concretas de *HTML* y *CSS* se han utilizado estas dado que debido a su recorrido son casi totalmente compatibles con todos los navegadores populares y ampliamente utilizados hoy en día. Se trata de las últimas versiones disponibles de estas tecnologías y simplemente son las que más posibilidades de implementación ofrecen y de la forma más simple. La tecnología *HTML 5* se ha utilizado para el esqueleto de las páginas Web que conforman a la aplicación. La tecnología *CSS 3* es la que permite establecer el estilo y aspecto de todos los elementos definidos en un documento *HTML*. *JavaScript* se ha utilizado para proporcionar la lógica de vista necesaria en la aplicación Web. En este caso no se utiliza para soportar *AJAX*.

3.4 Tecnologías Web frontend: Bootstrap 4 y JQuery 3.3.1

Bootstrap es un *framework* que se construye sobre las tecnologías Web del lado del cliente (*frontend*) que se han tratado en el apartado anterior y que sirve para simplificar y ayudar a construir interfaces gráficas compatibles con varios clientes Web (navegadores) y compatibles con diferentes anchos y tipos de pantallas (diseño *responsive*) (Bootstrap, 2018). Para ello cuenta con sistemas de disposiciones, una gran de variedad de componentes preconstruidos y varios *plugin* que se apoyan en *JQuery*.

En este proyecto se ha utilizado este *framework* para construir las diferentes pantallas y partes de la interfaz gráfica del panel Web empleando su sistema de disposición de elementos en cuadrícula y utilizando algunos de sus componentes y *plugin* para la barra superior de

navegación, los formularios y las tablas entre otros. Todos estos elementos son compatibles con los diferentes clientes Web (navegadores) y se adaptan de forma dinámica al ancho de la pantalla.

En el caso de *JQuery*, se trata de una biblioteca para *JavaScript* que permite simplificar el desarrollo del código *JavaScript* aprovechando las funcionalidades que esta ofrece (jQuery, 2018).

En el proyecto se ha utilizado ya que es una dependencia de *Bootstrap* y aprovechándolo se ha utilizado para simplificar el código *JavaScript* que se ha desarrollado por ejemplo para la funcionalidad de autorrellenado de los campos de los formularios del panel Web a partir de la pulsación de las filas de las tablas que en este se muestran.

Las versiones que se han decidido utilizar en ambos casos se han elegido simplemente por ser las más actuales en el momento del desarrollo del sistema sin que por ello se produzca ninguna incompatibilidad.

3.5 La tecnología Web backend PHP 7.2.6

Se trata de un lenguaje de programación de código abierto bastante popular y ampliamente conocido que se ejecuta en la máquina servidora y se puede integrar con código *HTML* (PHP, 2018). Aunque se diseñó originalmente para el desarrollo Web de contenido dinámico se trata de un lenguaje de propósito general. Así pues, se puede utilizar para generar código *HTML* que después se hace llegar al cliente. Se puede ver que el cliente lo que recibe es el resultado de la ejecución del script *PHP*. Como este resultado se recibe como código *HTML*, el cliente de la aplicación no podrá conocer la estructura del código que ha originado al resultado que ha recibido. Es un lenguaje interpretado y muy flexible.

Se trata de un lenguaje muy simple pero que al mismo tiempo cuenta con muchas características avanzadas que lo hacen válido para desarrollar todo tipo de proyectos.

Los motivos para el uso de esta tecnología son que sirve precisamente para lo que se busca, que es el desarrollo de aplicaciones Web con acceso a la información almacenada en un sistema gestor de bases de datos. Por otro lado, cuenta con una amplia documentación de calidad y completa con bastantes ejemplos para la comprensión del uso de este lenguaje. Es ampliamente conocido por muchos desarrolladores lo que posibilita obtener soporte o poder consultar dudas planteadas anteriormente por otros desarrolladores. Se trata de un lenguaje no tipado, es decir, que no se necesita definir los tipos de las variables. Es compatible tanto con la programación orientada a procedimientos como con la programación orientada a objetos. A pesar de los anteriores motivos, se pueden destacar por encima que la elección es por gusto propio del desarrollador, conocimiento previo y experiencia con este lenguaje, y la posibilidad de total integración con *HTML*.

Respecto a la versión concreta esta se ha elegido por ser la más actual y por sus ventajas ofrecidas en funcionalidades y rendimiento dado que su uso no produce ningún problema de compatibilidad.

En el proyecto se ha utilizado para implementar todas las partes dinámicas del panel Web desde el que se gestiona el contenido de la aplicación Android mostrando en tablas la información procesada que se obtiene de un sistema gestor de bases de datos. También se encarga de procesar la información que se recibe mediante formularios para después almacenarla de forma correcta en las bases de datos. Por último, también se ha utilizado para implementar el servicio Web que va a declarar una *API REST* que la aplicación Android va a poder utilizar para solicitar información que se va a obtener de la base de datos y que tras procesarse se va a proporcionar como respuesta a la petición recibida desde la aplicación.

3.6 La tecnología Web backend MariaDB 10

Se trata de un sistema gestor de bases de datos de código abierto que utiliza el lenguaje *SQL* (*Structured Query Language*) (MariaDB, 2018). Es uno de los sistemas más populares y cuenta con una alta compatibilidad con el sistema *MySQL*. Se ha utilizado este pues ofrece las suficientes características y funcionalidades necesarias para el desarrollo de este proyecto y porque es el que viene con el entorno de pruebas *XAMPP* que se ha utilizado para el desarrollo de forma local del sistema antes de su despliegue en un servicio hosting de terceros.

En el proyecto se ha utilizado para almacenar toda la información del contenido que luego se va a mostrar de forma dinámica en la aplicación Android y que se sirve a través de la *API REST*.

Respecto a la versión utilizada, de nuevo, ya que su uso no provoca ninguna incompatibilidad el único motivo dada la relativa sencillez de la aplicación Web y servicio Web desarrollados es el de utilizar la última versión disponible en el momento del inicio del proyecto.

3.7 Android 5.1 Lollipop

La plataforma Android se compone del propio sistema operativo de igual nombre, un conjunto de bibliotecas, un conjunto de aplicaciones para el usuario y un entorno de ejecución (máquina virtual *Java*) (AOSP, 2018).

Algunas de las novedades que presenta esta versión de Android son la nueva interfaz gráfica basada en *Material Design*, los nuevos ajustes rápidos de la barra de notificaciones con la funcionalidad de linterna, mejoras en la vida útil de la batería con el proyecto Volta, mejora de la pantalla de bloqueo que ya no admite *widget* pero permite mostrar notificaciones, se vuelve a disponer de acceso en cualquier ubicación al almacenamiento externo por parte de las aplicaciones de terceros, se independiza *WebView* como una aplicación propia para que esta pueda por seguridad ser actualizada desde *Google Play* sin tener que depender del lanzamiento de actualizaciones por parte de los fabricantes de dispositivos, se cuenta con una gráfica

detallada del consumo de energía y la estimación del tiempo restante de batería y de carga, mejora de las notificaciones que admiten ahora prioridades, el modo no molestar, notificaciones emergentes, mejoras en la gestión de la batería (consumo), implantación definitiva de la máquina virtual *Java ART*, cifrado predeterminado de las particiones de datos del usuario (aunque finalmente se dejó a elección del fabricante), sistema antirrobo de Android basado en el bloqueo del *bootloader* y el almacenaje de las cuentas de Google con las que se ha autenticado el usuario en el dispositivo de forma que en caso de robo incluso tras un restablecimiento de fábrica el dispositivo imposibilita su uso hasta autenticarse con una de las cuentas almacenadas, modo ahorro de energía, mejor aplicación y configuración del sistema *MAC SELinux*, y otras mejoras de rendimiento, estabilidad y seguridad (Xataka, 2018).

En referencia a la arquitectura de la plataforma Android y del sistema operativo Android, estas se muestran respectivamente en las figuras Figura 9 y Figura 10 (AOSP, 2018).

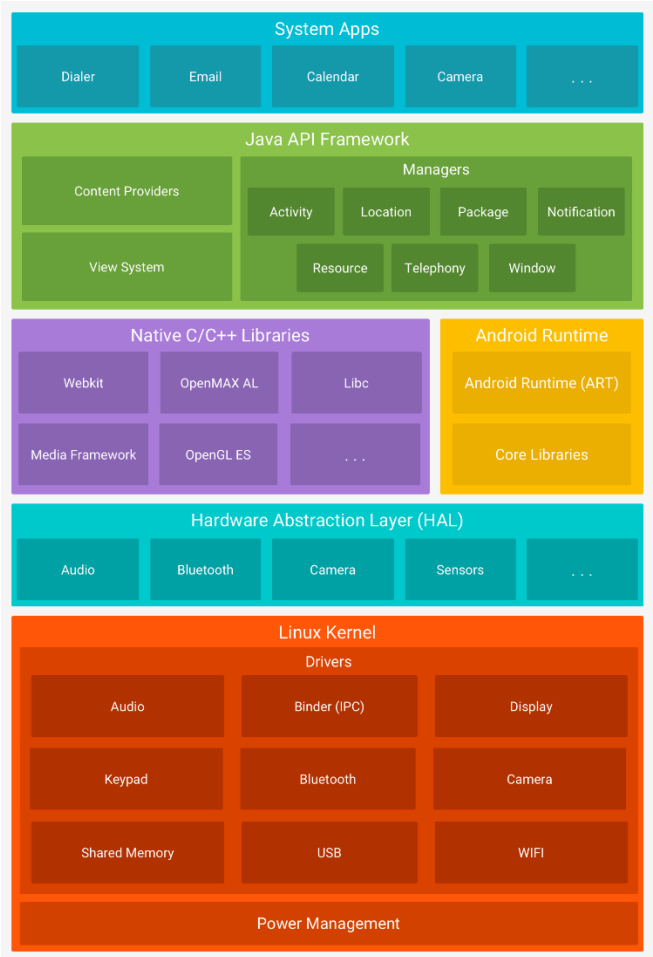

Figura 9. Arquitectura de la plataforma Android.

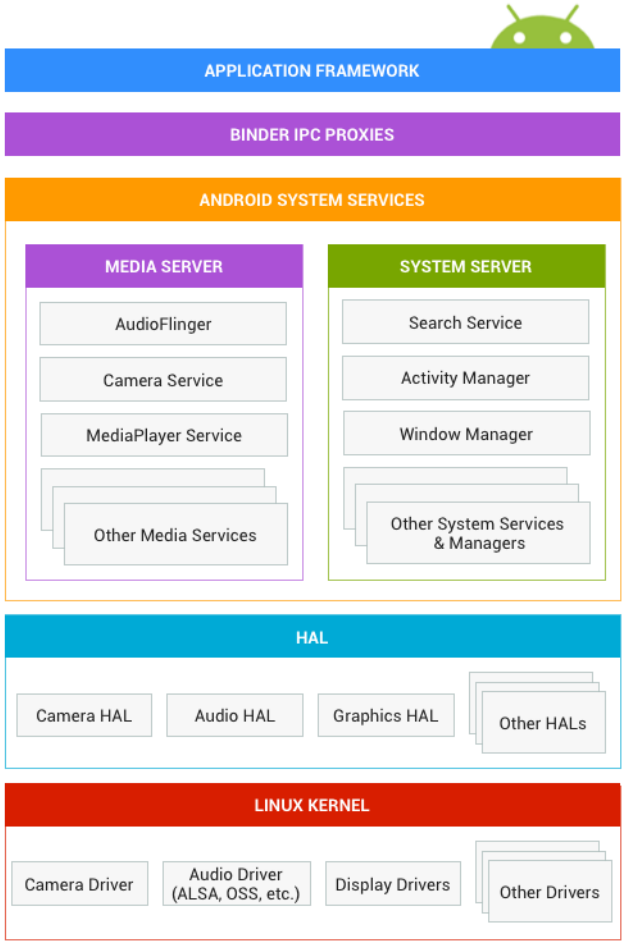

Figura 10. Arquitectura del sistema operativo Android.

Respecto a la arquitectura de la plataforma Android se realiza a continuación una explicación breve de algunas de sus partes:

- **Kernel de Linux**: Es la base de la plataforma y proporciona funcionalidades subyacentes como la generación de subprocesos (*thread*) y la administración de memoria a bajo nivel. Permite aprovechar funciones de seguridad claves y

también el desarrollo de drivers por los fabricantes al tratarse de un *kernel* conocido.

- **Capa de abstracción de hardware (HAL)**: Proporciona interfaces estándar que exponen las capacidades del hardware a las capas de más arriba. Se compone de varios módulos y cada uno cuenta con una interfaz para un tipo específico de componente hardware. Esta capa es la que permite independizar al hardware subyacente del resto de capas de niveles superiores.
- **Android Runtime**: Cada aplicación ejecuta sus subprocesos en una nueva instancia de la máquina virtual *Java ART*. Permite la compilación *AOT* en uso por defecto y *JIT* utilizada en versiones anteriores por defecto. También cuenta con el *Garbage Collector* que permite recuperar memoria eliminando elementos no utilizados (no referenciados) por las aplicaciones en ejecución.
- **Bibliotecas C/C++ nativas**: Son las bibliotecas escritas en *C/C++* que se denominan nativas porque se ejecutan directamente en el procesador en lugar de mediante una máquina virtual *Java*. Muchos de los componentes y servicios centrales del sistema Android se encuentran implementadas en código nativo (*C/C++*) que requiere del uso y apoyo de estas bibliotecas. Incluso el *framework Java* ofrece con su API acceso a las aplicaciones a estas bibliotecas nativas como es el caso de *Open GL ES*.
- **Framework Java**: Todo el conjunto de funciones del sistema operativo Android está disponible para las aplicaciones a través de las API que proporciona este *framework Java*. Permite la reutilización de componentes del sistema y el uso de los servicios centrales y modulares como son el sistema de vistas (para la creación de las interfaces gráficas), el administrador de recursos, el administrador de notificaciones, el administrador de actividad (por ejemplo, el ciclo de vida de las actividades) y proveedores de contenido entre otros.
- **Aplicaciones del sistema**: Son aplicaciones para los usuarios y como tales no tienen un significado especial frente a las otras aplicaciones que el usuario decida instalar. Es por ello que el usuario podría utilizar otras aplicaciones que sustituyan a estas que proporciona por defecto el sistema operativo. En cualquier caso, estas suelen ser ampliamente utilizadas por los usuarios y el que vengan incorporadas con el sistema proporciona funcionalidades claves a las que los desarrolladores pueden acceder desde sus propias aplicaciones. Algunos ejemplos son aplicaciones para el correo electrónico, mensajería SMS, calendarios, agenda de contactos, cliente Web (navegador), teclado, teclado de marcado para llamadas (*dialer*), etc.

Respecto a la arquitectura del sistema operativo Android se realiza a continuación una explicación breve de algunas de sus partes:

- **Framework de aplicaciones**: Utilizado por los desarrolladores de aplicaciones y debe ser tenido en cuenta por los desarrolladores de hardware porque en varios casos las API para los desarrolladores se mapean con las API de la capa *HAL* y podría revelar detalles sobre los *driver*.
- **Binder IPC**: Permite a las API de los *framework* de alto nivel interactuar con los servicios del sistema que ofrece Android ocultando la complejidad de las comunicaciones entre estos componentes y proporcionando la sensación de que simplemente funciona.

- **Servicios del sistema**: Se trata de componentes modulares como el *Window Manager*, el servicio de búsqueda o el gestor de notificaciones. Las funcionalidades expuestas a las aplicaciones con la API del *framework* de aplicaciones permite la comunicación con estos servicios del sistema para acceder a través de ellos al hardware subyacente. Los servicios del sistema están repartidos en dos grupos: sistema y *media*.
- **Capa de abstracción de Hardware (HAL)**: Proporciona una interfaz estándar que los fabricantes de hardware deben implementar. Esto hace a Android agnóstico de las implementaciones de los controladores de bajo nivel. Como ya se ha dicho antes, a efectos prácticos permite independizar a las capas superiores del hardware sobre el que se ejecutan sus componentes. Está compuesta de módulos que se van cargando según son necesitados.
- **Kernel Linux**: Desarrollar los controladores para un dispositivo es similar desarrollar un controlador de dispositivo Linux típico. Android utiliza una versión del *kernel* Linux con algunas modificaciones como un sistema de administración de memoria más agresivo, *wake locks* (servicio de gestión de energía), o el controlador o *driver Binder IPC* entre otros. Estas modificaciones no afectan al desarrollo de *driver* para otros componentes hardware.

Por otro lado, para el desarrollo de las aplicaciones se cuenta con componentes o bloques de creación esenciales. Cada uno de estos componentes son un punto mediante el cual se puede hacer uso de la aplicación (desde fuera). Estos componentes ayudan a definir el comportamiento general de la aplicación. Cada tipo tiene un fin específico y un ciclo de vida diferente que define cómo se crea y se destruye el componente. A continuación, se listan y explican brevemente estos componentes:

- **Actividades**: Representa una pantalla con interfaz gráfica. Aunque trabajan juntas para proporcionan una experiencia de usuario consistente cada actividad es independiente. Cualquier otra aplicación puede iniciar alguna de estas actividades si la aplicación que las implementa lo permite.
- **Servicios**: Es un componente que se ejecuta en segundo plano y está pensado para llevar a cabo tareas largas o tareas para procesos remotos. No proporciona una interfaz gráfica para el usuario.
- **Proveedores de contenido**: Es un componente que administra un conjunto de datos compartidos de la aplicación. De esta forma, otras aplicaciones pueden acceder (consultar e incluso modificar) datos que la aplicación ofrece a través de este componente siempre que lo permita. Se puede ver como un mecanismo que permite controlar a la aplicación cómo ofrece parte de su información a otras aplicaciones diferentes. También son útiles y empleados para leer y escribir datos privados por la propia aplicación y que no se comparten a otras aplicaciones.
- **Receptor de mensajes**: Es el componente que recibe todos los anuncios de mensajes de todo el sistema (mensajes de difusión generados por el propio sistema o generados por otras aplicaciones). Es habitual que estos mensajes se correspondan con avisos de eventos detectados y este componente pretende ser una “puerta de enlace” para otros componentes. Se puede decir que se encargan de poner en conocimiento a otros componentes de los mensajes que reciben. Por ello, comúnmente realizan cantidades mínimas de trabajo. No cuentan con una interfaz gráfica de usuario.

Tratando ahora el tema de la justificación de la plataforma Android por la que se ha optado para el desarrollo de HepApp se deben destacar algunos motivos de elevada importancia. A parte de todo lo explicado sobre Android y sus posibilidades en los párrafos anteriores de este subapartado y en el subapartado *Android*, por un lado, se ha decidido utilizarla porque como se puede comprobar en la figura Figura 11 se trata del sistema móvil claramente predominante en el mercado con una presencia del 74.36% (StatCounter, 2018).

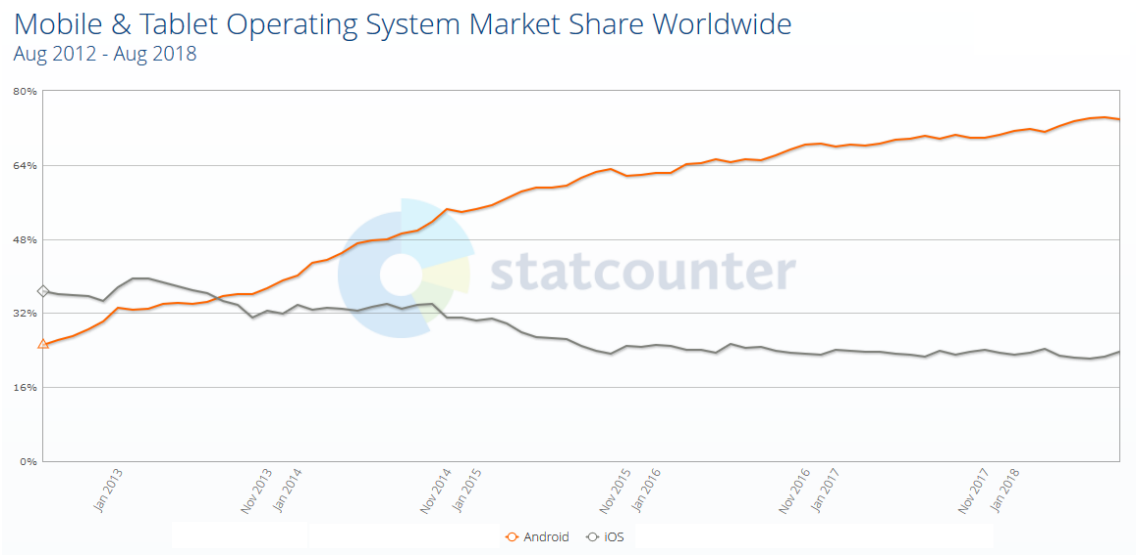

Figura 11. Distribución del mercado móvil entre las plataformas con una cuota relevante.

Por otro lado, otros motivos son el *IDE* que ofrece esta plataforma que puede ejecutarse en cualquier equipo sin obligar a contar con un equipo de Apple, también hace uso de *Gradle* y cuenta con la herramienta *Android Profiler* que permite obtener estadísticas de la aplicación en tiempo real de su ejecución para ayudar a detectar los cuellos de botella que se producen en esta y de esta forma permitir al desarrollador optimizar la aplicación (Android Developers, 2018). Además, este *IDE* se ha considerado de sencillo uso, completo y muy funcional. Hace uso de *ProGuard* para optimizar y reducir el código desarrollado, cuenta con un editor de código inteligente, un buen depurador, y editores gráficos y previsualizadores (vista previa) para la creación cómoda de las interfaces gráficas entre otros. La combinación con el *Emulador de Android* y la compatibilidad con el sistema *Git* para el control de versiones en un repositorio (en este caso se ha integrado para hacer uso de un repositorio privado en *GitHub*) son ya la puntilla para contar con un *IDE* y todas las herramientas necesarias que simplifican y hacen eficiente la tarea del desarrollo.

Además, se ha considerado el coste de la licencia para desarrolladores de *Google Play*.

A parte, de nuevo, se trata de una motivación de gusto personal y conocimiento previo, soltura y experiencia en el desarrollo con el lenguaje *Java* por parte del desarrollador.

También, se considera que las posibilidades de implementación que ofrece esta plataforma son las suficientes para el proyecto que se desea desarrollar. Si bien el soporte para visualizar ficheros PDF es un poco escaso y sólo disponible en versiones más altas de Android, al tratarse de software de código abierto fácilmente se da con varias bibliotecas que con suma facilidad se pueden integrar en la aplicación para contar con más que soporte para visualizar este tipo de ficheros. En este caso para visualizar los ficheros PDF se ha hecho uso de la biblioteca *Android PdfViewer* de *barteksc* en su última versión estable disponible. Para el caso de la

comunicación entre la aplicación Web y la aplicación móvil, las posibilidades ofrecidas por la biblioteca oficial *Google Volley* simplifican al máximo las tareas de uso de la infraestructura de red a través de Internet para lograr consumir el servicio Web desde la aplicación a través de la *API REST* que este ofrece.

La documentación disponible tan completa, de calidad, actualizada y con tantos ejemplos y guías son otro factor decisivo para optar por esta plataforma.

Respecto a la versión de Android elegida (*5.1 Lollipop*), dados los datos de la tabla Tabla 2 y de la figura Tabla 2, se considera que optando por esta versión se está abarcando el 82.8% del total de los dispositivos móviles que funcionan con Android al mismo tiempo que no se renuncia a poder hacer uso de las funcionalidades y características presentes a partir de esta versión concreta (Android Developers, 2018).

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.2%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.2%
4.2.x		17	1.9%
4.3		18	0.5%
4.4	KitKat	19	9.1%
5.0	Lollipop	21	4.2%
5.1		22	16.2%
6.0	Marshmallow	23	23.5%
7.0	Nougat	24	21.2%
7.1		25	9.6%
8.0	Oreo	26	10.1%
8.1		27	2.0%

Tabla 2. Distribución de las versiones de Android.

Capítulo 4: Algoritmos de hepatología

4.1 Introducción

El carcinoma hepatocelular (*HCC*) es un tumor agresivo que a menudo se produce en el contexto de la enfermedad hepática crónica y cirrosis. Por lo general, el diagnóstico es tardío y la supervivencia media tras el diagnóstico es aproximadamente de seis a veinte meses. Esto pone de relieve la necesidad de un diagnóstico precoz y un tratamiento agresivo con el fin de mejorar el resultado de esto (Pin Vieito et al, 2014).

A la hora de evaluar el pronóstico hay que tener en cuenta que, dado que el *HCC* aparece en la mayor parte de los casos sobre una cirrosis hepática, el pronóstico depende en gran medida de la función hepática existente. Actualmente se consideran 4 factores decisivos que afectan al pronóstico:

1. El estado, agresividad y patrón de crecimiento del tumor.
2. El estado general del paciente, definido habitualmente mediante el *performance status* (PST) de la OMS (Sorensen et al., 1993).
3. La función hepática, medida mediante la escala de Child-Pugh (Child & Turcotte, 1964).
4. El tratamiento que se aplique.

No es fácil una clasificación pronóstica que tenga en cuenta todas las variables citadas. Sin embargo, las cuatro clasificaciones más empleadas en la actualidad son la de *Okuda*, la del grupo italiano, CLIP (*Cancer of the Liver Italian Program Investigators*), la TNM y la de Barcelona Clinic. La más empleada es la clasificación de Barcelona, que a su vez tiene en cuenta la de *Okuda*, la funcionalidad de la OMS y la función hepática. Por una parte, permite definir aceptablemente bien el pronóstico, pero su virtud fundamental es permitir definir de una forma objetiva el tratamiento a aplicar (Linares et al., 2004).

En este capítulo se verán detenidamente todos los algoritmos de estadificación (Kinoshita et al., 2015) que están incluidos en la aplicación: *Okuda, CLIP, GETCH, TNM, Cupi y BCLC.* Además de las funciones hepáticas (*APRI, Child Pugh Score* y *MELD*) y los criterios que determinan si un paciente es candidato a trasplante o no, como son el *Criterio de Milán, TTV+AFP, UCSF* y *UpToSeven*.

4.2 Funciones hepáticas

El TH debe ser considerado en cualquier paciente con enfermedad hepática en estado avanzado en que el TH pueda significar un aumento en la esperanza de vida más allá de lo esperado conforme a la enfermedad subyacente, o un aumento en la calidad de vida (CdV) de dicho paciente. Se deben seleccionar, como candidatos a TH aquellos pacientes cuya calidad de vida sea mala o inaceptable, o cuya expectativa de vida sea menor de un año. Asimismo, se debe realizar una evaluación médica detallada con el fin asegurar la viabilidad del TH (EASL, 2016).

La elección del momento adecuado para la realización de TH es crucial, ya que los candidatos a TH por patología hepática en estado avanzado deben ser intervenidos antes de que ocurran complicaciones potencialmente mortales. Sin embargo, tampoco deberían ser trasplantados con excesiva precocidad, en cuyo caso los riesgos derivados de la propia intervención y de la inmunosupresión de por vida podrían superar a los beneficios del TH (EASL, 2016). En el pasado, la prioridad en la lista de espera venía determinada por el tiempo de espera y la gravedad de la patología hepática. Actualmente, para determinar la prioridad de los pacientes incluidos en la lista de TH, se utilizan la clasificación Child-Pugh-Turcotte y, desde 2002, también la escala MELD (*Model of End-stage Liver Disease*) (basada en variables objetivas como la creatinina, la bilirrubina, y el INR). La puntuación de Child-Pugh ha demostrado ampliamente su valor predictivo sobre la supervivencia de estos enfermos. Sin embargo, introduce variables con un componente subjetivo, como la intensidad de la ascitis o de la encefalopatía. El baremo MELD es el resultado de un modelo con 3 variables objetivas de fácil obtención (creatinina, bilirrubina, INR) (De la Mata & Barrera, 2003).

4.2.1 APRI

El índice de relación entre AST y plaquetas (APRI) determina la probabilidad de la fibrosis hepática y la cirrosis en pacientes con Hepatitis C.

Esta función hepática requiere de tres variables: AST, límite superior de AST y plaquetas.

El resultado se calcula con la siguiente expresión: ***APRI = ((AST / ASTLimit) / Plaquetas) x 100***

4.2.2 Child Pugh Score

El índice de *Child Pugh* se diseñó en la década de los setenta. Hasta ahora ha sido el índice pronóstico más utilizado y consta de cinco variables de las que tres reflejan la función hepática (albúmina, bilirrubina e IRN) y dos se refieren a complicaciones de la enfermedad (ascitis y encefalopatía). Como se puede ver en la figura Figura 12, cada variable se puntúa entre uno y tres puntos según el grado de afectación, por lo que la puntuación mínima es de cinco puntos y la máxima de quince (Child & Turcotte, 1964).

Clásicamente se establecen tres grandes grupos: *Child-Pugh A* si la puntuación es cinco o seis, lo que indica buena función hepática; los pacientes con este grado de afectación hepática tienen buena supervivencia a medio plazo (aproximadamente 80% a los cinco años) y no precisan ser tratados con un trasplante hepático. Cuando la puntuación es de siete a nueve puntos se considera *Child-Pugh B*, lo que significa función hepática intermedia. Estos pacientes tienen indicación de trasplante si han presentado alguna descompensación (ascitis o encefalopatía hepática). Los pacientes con diez a quince puntos se consideran *Child- Pugh C*, lo que significa mala función hepática, supervivencia muy comprometida a corto plazo e indicación de trasplante hepático.

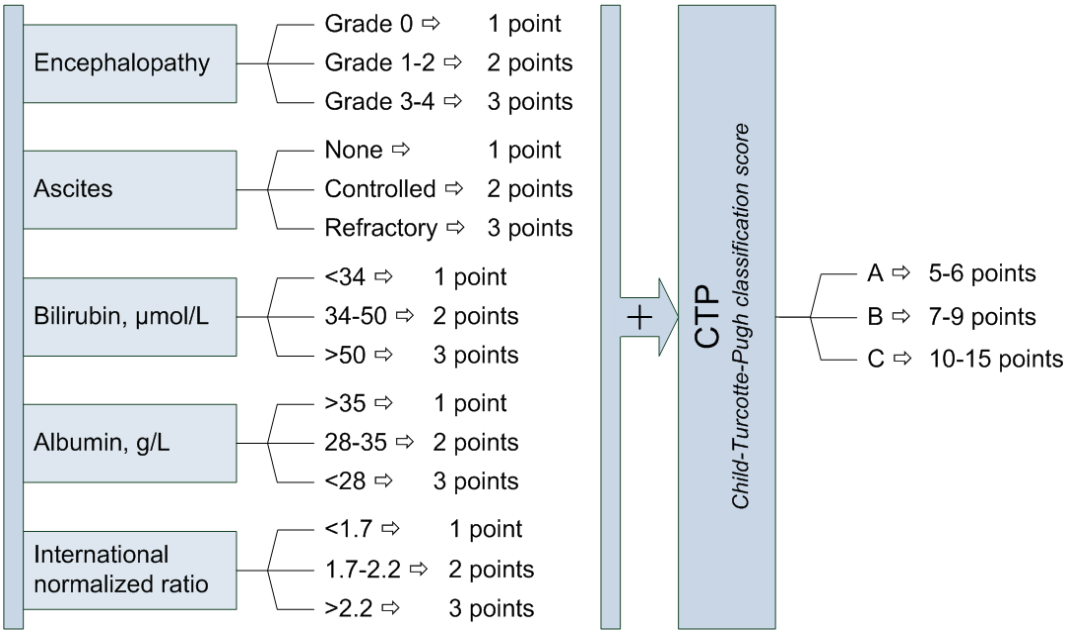

Figura 12. Esquema resumen de Chlid-Pugh Score.

El índice de *Child Pugh* es muy fácil de determinar, puede calcularse a la cabecera del paciente sin necesidad de calculadora y obtiene una buena predicción de la mortalidad a corto-medio plazo. Sus principales inconvenientes son que no se ha obtenido en análisis multivariante; la valoración del grado de ascitis y encefalopatía es subjetiva; no discrimina cuando la enfermedad está muy avanzada ya que tiene límites fijos para cada variable, por ejemplo, si el paciente tiene una bilirrubina de 3,5mgr/dl se aplican tres puntos, pero si es la bilirrubina es de 20 mgr/dl también se aplican tres puntos; y no tiene en cuenta parámetros de función renal que tienen valor pronóstico en la cirrosis hepática. A pesar de sus inconvenientes se sigue utilizando para valorar la indicación de trasplante hepático, que se considera indicado en los pacientes *Child* B y C (González-Pinto et al., 2007; Wiesner et al., 2003).

4.2.3 MELD

El *MELD* es un índice pronóstico utilizado para valorar la gravedad de la cirrosis hepática, que fue obtenido en un análisis multivariante y validado en más de 2.000 pacientes. A través de una fórmula que contiene tres variables objetivas, bilirrubina, INR y creatinina se obtiene una puntuación entre cinco y cuarenta puntos que se correlaciona muy bien con la mortalidad a tres meses. Desde el año 2002 se utiliza para priorizar los pacientes en lista de espera de trasplante hepático ya que su determinación no viene influida por valoraciones subjetivas y las variables que contiene son asequibles y reproducibles (Kamath et al., 2001).

El modelo *MELD* (Model for End-stage Liver Disease) se publicó por primera vez en el año 2000 y tiene un origen similar al índice de *Child-Pugh*. Aunque inicialmente incluía cuatro variables, ya que en el primer estudio incluía la etiología de la cirrosis (los pacientes con enfermedad colostática y de origen alcohólico tienen mejor pronóstico que los de otras etiologías), esta variable se consideró que no era objetiva y se suprimió, sin que ello alterase su capacidad pronóstica. Consta de tres variables que son objetivas, reproducibles y muy

asequibles: bilirrubina, INR y creatinina. Ha sido validado prospectivamente en cuatro cohortes que incluían más de 2.000 pacientes afectos de cirrosis hepática de Europa y USA, tanto hospitalizados como ambulatorios. Su mejor virtud es que es muy útil para determinar la mortalidad a tres meses con una C estadística de 0,80 (o sea que acierta en el 80% de los casos) (Kamath et al., 2001; Matthews et al., 2010). El cálculo del *MELD* es complicado ya que incluye logaritmos neperianos, por lo que se necesita calculadora como la que incluye la aplicación asociada a este informe.

Las principales ventajas del *MELD* sobre el *Child-Pugh* es que se ha obtenido de un análisis multivariante por lo que el peso de cada variable es distinto según su peso predictivo real (en el *Child-Pugh* todas las variables tiene el mismo valor), las variables son objetivas, evitándose valoraciones subjetivas de la ascitis o del grado de encefalopatía, y además valora de forma continua las variables sin un techo máximo como *Child-Pugh*. En la valoración de mortalidad a corto plazo (tres meses) es mejor que el *Child- Pugh*, ya que en la mayoría de estudios en los que se comparan ambos índices, se obtiene una mejor C estadística para el *MELD* que para el *Child-Pugh,* aunque en alguno de ellos la diferencia no alcanza significación estadística. Ello confirma que, a pesar de todos sus inconvenientes, el *Child-Pugh* también aporta una buena capacidad pronóstica. Además de predecir la mortalidad a tres meses en la cirrosis hepática, el *MELD* también es un buen índice pronóstico cuando se aplica a pacientes con otras enfermedades hepáticas graves tales como hepatitis alcohólica aguda o insuficiencia hepática aguda (Wiesner et al., 2003; Myers et al, 2013)

Se dispone de tres versiones de MELD (Myers et al, 2013):

- MELD: Es la función hepática original y se calcula mediante la siguiente fórmula matemática: ***MELD = 11.2 x log(IRN) + 3.78*log(bilirrubina) + 9.57 x log(creatinina) + 6.43***
- MELDNa: se ha sugerido que la adición de sodio al MELD mejora su capacidad de predicción de la supervivencia (Sethepatico, 2014) y se calcula mediante la siguiente expresión: ***MELDNa = MELD - sodio - (0.025 x MELD x (140 - sodio)) + 140***
- MELDV5: se añade además del sodio la albúmina como variable de cálculo y se calcula mediante la siguiente función: ***MELD5v = MELDNa + (5.275 x (4 - albumin)) - (0.163 x MELD x (4 - albumin))***

4.2 Algoritmos de estadificación

Los sistemas de estadificación en el cáncer se utilizan generalmente para el pronóstico, pero también pueden ser útiles para la elección de la terapia adecuada. HCC presenta una situación donde se requiere el sistema de estadificación para evaluar el riesgo a partir de dos condiciones que amenazan la vida del paciente: el cáncer y la cirrosis de hígado. Por lo tanto, es comprensible que cualquier sistema de clasificación usado para HCC debe incorporar indicadores de pronóstico no sólo para el estado de tumor, sino también para la función hepática y el estado de salud general (Pons et al., 2005).

El conocimiento actual de la enfermedad, sin embargo, impide la recomendación de un sistema de clasificación que se puede utilizar en todo el mundo. Los sistemas convencionales de estadificación para el carcinoma hepatocelular (HCC), tales como la etapa de Okuda o la etapa TNM han demostrado limitaciones importantes en la clasificación de los pacientes. Varios de los nuevos sistemas se han propuesto recientemente, y sólo tres de ellos han sido validados en este punto. La clasificación en estadios BCLC une el estadio de la enfermedad a una estrategia de tratamiento específico. La puntuación JIS se ha propuesto y utilizado en Japón, aunque necesita la validación occidental. La puntuación CLIP se utiliza en pacientes con tumores avanzados. Hay varias razones que explican la dificultad de identificar un sistema en todo el mundo. En primer lugar, HCC es una neoplasia compleja insertada en un hígado cirrótico pre-neoplásicas, y por lo tanto las variables de ambas enfermedades que conducen a la muerte se debe tener en cuenta. En segundo lugar, la enfermedad es muy heterogénea en todo el mundo, y esto refleja diferentes antecedentes epidemiológicos y factores de riesgo subyacentes. En tercer lugar, HCC es el único cáncer tratado por el trasplante en una pequeña proporción de los pacientes. En cuarto lugar, sólo alrededor del 20% de los casos se tratan actualmente mediante cirugía, impidiendo así el amplio uso de sistemas basados en la patología, tales como TNM. Por último, la relevancia potencial de una firma molecular identificado en términos de predicción de resultados es desconocido, y se necesita más investigación para obtener esta información biológica valiosa que puede ayudar en la clasificación de los pacientes (Leung et al., 2002; Cheng et al., 2011; Martinez-Miera et al., 2014).

4.2.1 Okuda

El sistema de estadificación de Okuda fue un avance en las clasificaciones anteriores de estadificación hepatocelular (HCC), en el sentido de que incorporó tanto las variables relacionadas con el cáncer como las variables relacionadas con la función hepática para determinar el pronóstico.

La clasificación de Okuda se ha aplicado ampliamente en pacientes con HCC en la última década. Incluye parámetros relacionados con el estado funcional del hígado (albúmina, ascitis, bilirrubina) y la etapa de tumor; más o menos del 50% de la superficie del hígado en cuestión. Esta clasificación de los pacientes se llevó a cabo correctamente cuando la mayoría de ellos fueron diagnosticados en una etapa avanzada. Es útil para identificar pacientes en etapa terminal (Okuda etapa III). Hoy en día, el conocimiento del diagnóstico ha avanzado y, por tanto, esta clasificación no es adecuada para clasificar a los pacientes antes de terapias radicales o paliativos, incluso cuando se dividen los pacientes en etapa Okuda I en dos subgrupos de acuerdo con el tamaño del tumor. Cuando se compara con sistemas de estadificación modernas, se ha demostrado que esta tiene menor capacidad de predicción.

El cálculo del algoritmo de Okuda se basa en un sistema de puntaje que depende del valor de las diferentes entradas de datos. Se obtendrá un punto si el tamaño del tumor es superior al 50% de la superficie del hígado y cero puntos en caso contrario. En el caso de que el paciente sí que tenga ascitis (acumulación de líquido en el área que rodea los órganos del abdomen) se sumará un punto y en caso contrario cero puntos. Si los niveles de albúmina están por encima de 3 mg/dL se aumentará en uno la puntuación de Okuda al igual que si los niveles de bilirrubina son mayores o iguales que 3 mg/dL. En casos contrarios no se sumará ningún punto (Okuda et al., 1985). Se puede ver este esquema en la Figura 13.

Una vez sumados los puntos se realiza la siguiente clasificación:

- Etapa I: si se han obtenido cero puntos.
- Etapa II: si se han obtenido de uno a dos puntos.
- Etapa III: si se han obtenido de tres a cuatro puntos.

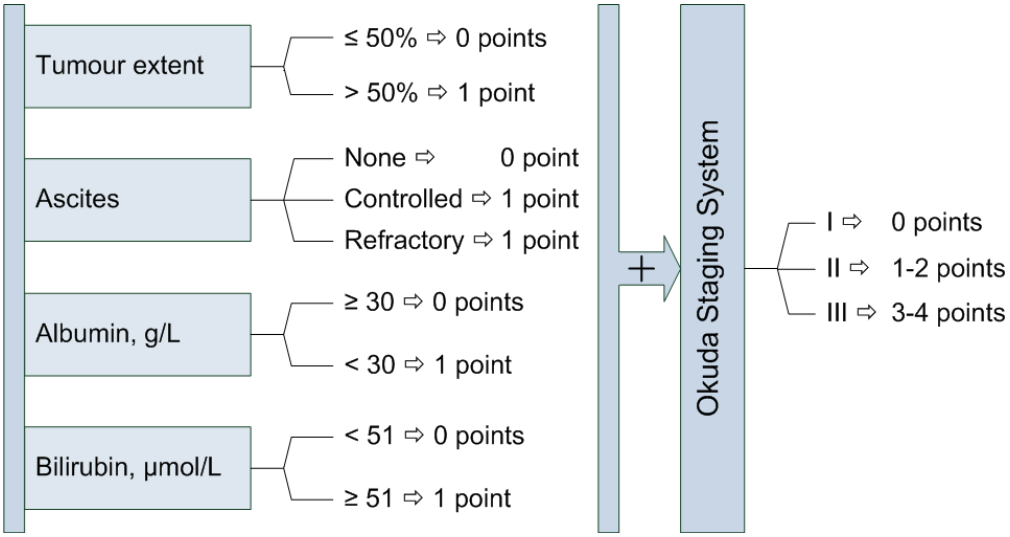

Figura 13. Esquema resumen de Okuda.

4.2.2 Clip

El programa de puntuación italiana del cáncer del hígado (CLIP) se construyó en un estudio retrospectivo y fue validado por los autores y otros grupos. Esta puntuación combina cuatro variables que proporcionan un sistema de clasificación de siete etapas. Ha sido comparado con Okuda y TNM, obteniendo éste un mejor poder discriminatorio. Grupos asiáticos han reportado tasas de supervivencia claramente diferentes a los autores originales, comprometiendo así su validación externa. También está limitado por el hecho de que no sirve para seleccionar el tratamiento adecuado para cada paciente (Leung et al., 2002; The CLIP Investigators, 1998; The CLIP Investigators, 2000).

Al igual que en el caso anterior este algoritmo se basa en un sistema de puntos en el que interviene el resultado de una función hepática, Child Pugh. En el caso de que sea *A* se sumarán cero puntos, si es *B* se sumará un punto y si es *C* se sumarán dos puntos. Si el tamaño del tumor es superior al 50% de la superficie del hígado y el número de tumores es mayor que uno se sumarán dos puntos, si la extensión del tumor es menor o igual al 50% y el número de tumores es mayor que uno se sumará un punto y en el caso de que la extensión del tumor sea menor o igual al 50% y el número de tumores sea igual a uno se sumarán cero puntos. Se sumará un punto en el caso de que sí tenga PVI y cero en caso contrario. Y, por último, se sumará un punto en el caso de que la cantidad de AFP sea superior o igual a 400 ug/L y cero en caso contrario. En la Figura 14 se puede ver el resumen de este algoritmo (The CLIP Investigators, 1998).

Una vez sumados los puntos individuales y obteniendo con ello la suma total se haría la siguiente clasificación:

- Etapa Temprana o *Early Stage*: si se han obtenido cero puntos.
- Etapa Intermedia o *Intermediate Stage*: si se han obtenido de uno a tres puntos.
- Etapa Avanzada o *Advanced Stage*: si se han obtenido de cuatro a seis puntos.

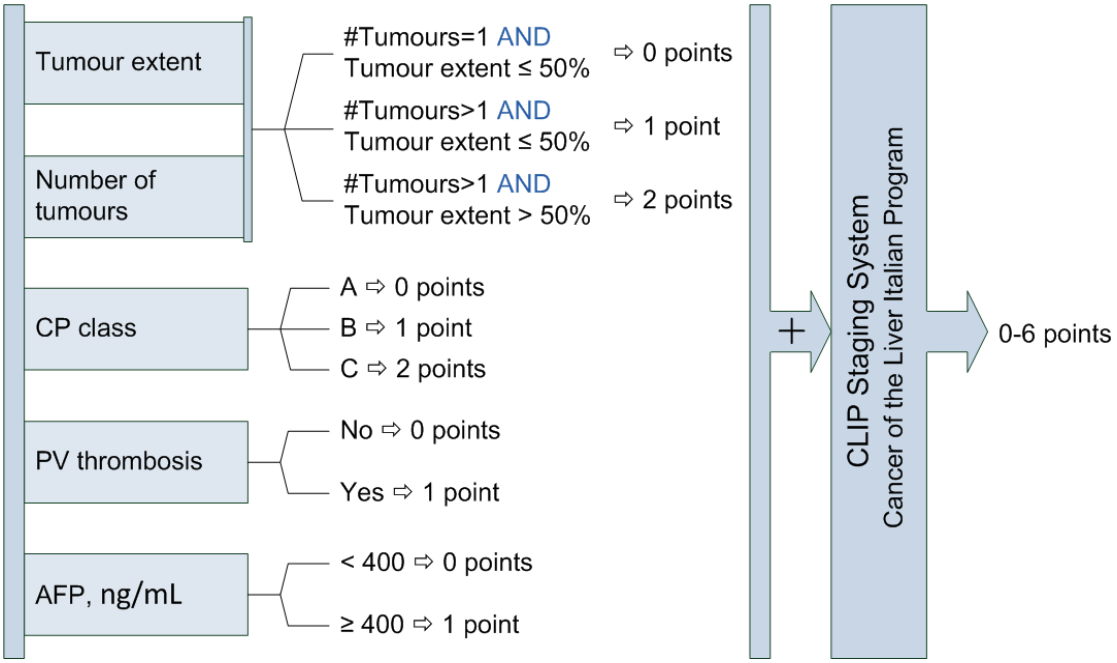

Figura 14. Esquema resumen de CLIP.

4.2.3 GETCH

Se trata de la clasificación francesa de Chevret et al. (1999) para evaluar el pronóstico de un paciente con carcinoma hepatocelular.

Como en los casos anteriores es un algoritmo basado en puntuación. Si *ECOG* es igual a cero o uno, se sumarán cero puntos y si es mayor o igual que dos se sumarán tres. Si la bilirrubina presenta cantidades superiores o iguales a 50 umol/L se sumará un punto, en caso contrario cero. Se hace una división entre el ALP y el ALPLimit; si este resultado nos da mayor o igual que dos se suman dos puntos y sino cero. También se sumarán dos puntos en el caso en el que las cantidades de AFP sean superiores o iguales a 35 ug/L. Por último, se sumará un punto más si el paciente tiene PVT (Chevret et al., 1999).

4.2.4 TNM

El sistema TNM convencional, que sólo contiene variables relacionadas con la etapa del tumor, ha sido probado sobre todo en el entorno quirúrgico, y se vio la pobre predicción de pronóstico en pacientes con HCC con intención de someterse a trasplante o resección. Con base en los resultados de una serie de 557 pacientes que se sometieron a resección, se ha propuesto una modificación reciente, incluyendo el estadio del tumor y la presencia de fibrosis. El nuevo sistema de cuatro etapas puede mejorar la estratificación de los tumores resecados, aunque es controvertido si van a aplicar a los pacientes no quirúrgicos. Ha sido aprobado por el Comité Conjunto sobre el Cáncer (AJCC) (Lei et al., 2006; Leung et al., 2002)

La clasificación de las diferentes etapas es la siguiente (Lei et al., 2006):

- Etapa I: si el número de tumores es uno y el paciente no tiene ni PVI ni Nodos ni Metástasis.
- Etapa II: si el paciente solo tiene un tumor y PVI o, si tiene más de un tumor, pero no tiene PVI y el tamaño de los tumores es menor de cinco cm. Para estar en esta etapa además debe cumplirse que no tenga ni Nodos ni Metástasis.
- Etapa IIIA: si el número de tumores es mayor que uno, aunque sean de mayor de cinco cm de diámetro, pero no tiene que tener ni PVI ni Nodos ni Metástasis.
- Etapa IIIB: si el número de tumores es mayor o igual que uno y tiene PVI, pero no tiene ni Nodos ni Metástasis.
- Etapa IVA: si el número de tumores es mayor o igual que uno y el paciente tiene Nodos, pero no tiene Metástasis.
- Etapa IVB: el número de tumores es mayor o igual que uno y tiene Metástasis.

4.2.5 Cupi

Los investigadores en Hong Kong describen un sistema de estadificación de análisis de su experiencia en 926 pacientes, la mayoría de ellos con cirrosis relacionada con el VHB. El Índice de Pronóstico de la Universidad de China (CUPI) considera seis variables predictivas, y divide a los pacientes en tres etapas. Los autores estiman que esta clasificación tiene una mejor estimación de la supervivencia de la puntuación CLIP y la etapa Okuda, aunque su poder discriminatorio en las primeras etapas es cuestionable, ya que la mejor supervivencia a 1 año fue de alrededor de 50%.

Este sistema de estadificación se basa en clasificar a los pacientes en función de una determinada puntuación, como en algunos de los algoritmos vistos con anterioridad. Para los valores de bilirrubina superiores o iguales que 52 se obtendrá cuatro puntos, si está entre 34 y 52 tres puntos y si es menor que 34, cero puntos. En el caso de que no tenga nada de ascitis se sumarán tres puntos y otros tres puntos en el caso de que los valores de ALP sean superiores o iguales que doscientos. También se sumarán dos puntos en el caso de que la cantidad de AFP sea superior o igual a quinientos. Para este sistema se utilizará también el resultado del algoritmo anterior, TNM. En el caso de que se esté en una etapa I o II se sumarán tres puntos, si se está en una etapa IIIA o IIIB se restará un punto y en el caso de que se esté en una etapa IVA

o IVB se sumarán cero puntos. Por último, se restarán cuatro puntos en el caso de que el valor de *ECOG* sea cero (Leung et al., 2002).

La clasificación para este algoritmo quedaría de la siguiente manera:

- *Low*: en el caso de que se obtenga una puntuación menor o igual que uno.
- *Intermediate*: en el caso de que se obtenga una puntuación que esté entre dos y siete incluidos.
- *High*: en el caso en el que se obtenga una puntuación mayor o igual que ocho.

4.2.6 BCLC

El sistema de estadificación clínica de Barcelona del cáncer de hígado (BCLC) se construyó sobre la base de los resultados obtenidos de varios. Esta propuesta no es un sistema de puntuación, ya que se deriva de la identificación de factores pronósticos independientes en el marco de varios estudios, conformando una clasificación por estados (Llovet et al., 1999). Esta clasificación utiliza variables relacionadas con la etapa del tumor, el estado funcional del hígado, estado físico, y los síntomas relacionados con el cáncer, y une las cuatro etapas descritas con un algoritmo de tratamiento. En resumen, los pacientes en estado cero con HCC muy temprano, son candidatos óptimos para la resección. Los pacientes en la fase A con HCC precoz son candidatos para terapias radicales (resección, trasplante de hígado o tratamientos percutáneos). Los pacientes en la etapa B con HCC intermedia se pueden beneficiar de la quimioembolización. Los pacientes en la etapa C con HCC avanzado pueden recibir nuevos agentes en el marco de ECA, y los pacientes en la etapa D con enfermedad en estadio final recibirán tratamiento sintomático.

Se ha sugerido que esta clasificación es la más adecuada para la orientación del tratamiento, y en particular para seleccionar a los pacientes que se encuentran en una etapa temprana que podrían beneficiarse de terapias curativas. En ese sentido, recientemente se ha validado como el mejor sistema de estadificación (Pons et al., 2005).

Como se mencionó anteriormente este algoritmo no se basa en un sistema de puntos. La clasificación de los pacientes en las diferentes etapas se realiza de la siguiente manera:

- *Very Early Stage:* si el resultado de Child Pugh Score es de clase A, *ECOG* es igual a cero y el paciente solo tiene un tumor de tamaño menor que 2 centímetros de diámetro.
- *Early Stage:* si el resultado de CPS es A o B y el valor de *ECOG* es igual a cero, así como que, o bien el paciente tenga un único tumor, o bien que tenga hasta tres tumores, pero con un tamaño menor que tres centímetros de diámetro.
- *Intermediate Stage:* que el resultado de CPS sea A o B y además que *ECOG* sea cero y el número de tumores sea menor que tres.
- *Advanced Stage:* que el resultado de CPS sea A o B y que ECOG sea igual a uno o a dos. Además, el paciente debe tener PVI, Nodos o Metástasis y el número de tumores tiene que ser mayor o igual que uno.

- *End Stage:* que el resultado del CPS sea C o que el valor de *ECOG* sea mayor que dos y que el número de tumores sea mayor o igual que uno.

4.2.7 Alberta

El objetivo último de esta calculadora podría reducirse a determinar un diagnóstico y un pronóstico en base a los resultados de los algoritmos estadísticos y funciones hepáticas que se obtienen a partir de los datos introducidos de un determinado paciente.

El Algoritmo HCC de Alberta se va a explicar claramente a partir del esquema que se muestra en la Figura 15-2 y que está implementado a partir del diagrama de bloques de la **¡Error! No se encuentra el origen de la referencia.**. Este esquema, en la aplicación desarrollada en Android, va a ser pintado, en un camino que va desde arriba hacia abajo, en función de los valores de los resultados y datos introducidos. Al final de ese camino se llegará al tratamiento más adecuado para el paciente en cuestión y se mostrará un pronóstico de la esperanza de vida. Además, en este esquema aparecen las diferentes etapas de clasificación definidas por al algoritmo *BCLC*, marcando con un óvalo la etapa en la que se encuentra el paciente (Alberta Health Services, 2017).

El camino o desarrollo que sigue este algoritmo se puede ver como si estuviera dividido en secciones. En una primera instancia o sección se encuentra lo referente a los tumores. En este caso el paciente presenta un único tumor con un tamaño menor o igual a dos centímetros de diámetro, lo que hace que se pinte este recuadro de azul. La siguiente sección depende únicamente del resultado obtenido en *Child Pugh Score* que se pintará de amarillo y que en este caso el resultado obtenido es A. En la siguiente sección se trata de las Cuestiones Clínicas y se identifica con el color rojo.

4.3 Criterios candidato a transplante

La mayoría de los pacientes con hepatocarcinoma en nuestro país tienen una cirrosis hepática subyacente. Esto hace que tanto el pronóstico del paciente como las posibles alternativas terapéuticas dependan no sólo de la estadificación tumoral, sino también del grado de insuficiencia hepática o de la existencia de hipertensión portal. En este contexto, el trasplante hepático permite el tratamiento tanto de la enfermedad tumoral como de la cirrosis hepática subyacente. Hasta los años noventa, el único límite para el trasplante hepático en pacientes con hepatocarcinoma era la presencia de metástasis a distancia. Por ello, los resultados de la mayoría de las series de trasplante hepático en pacientes con hepatocarcinoma eran muy malos, con una elevada incidencia de recidiva tumoral y una baja supervivencia a largo plazo. Sin embargo, también se comprobó que los pacientes con hepatocarcinomas incidentales (no detectados antes del trasplante por medio de los estudios radiológicos) tenían una supervivencia a largo plazo comparable con la de los pacientes trasplantados por afección no tumoral. Esto hizo pensar que el trasplante hepático podría ser una opción válida para pacientes con un escaso número de lesiones tumorales de pequeño tamaño (Herrero Santos, 2011).

4.3.1 Criterio de Milán

En 1996, el grupo del Dr. Mazzaferro publicó los resultados de una serie de 48 pacientes con hepatocarcinoma a los que se trasplantó, de acuerdo con unos criterios previamente establecidos: ausencia de invasión vascular o enfermedad tumoral a distancia y una única lesión tumoral que no excediera los cinco cm de diámetro o dos a tres lesiones tumorales menores de tres cm (Mazzaferro et al., 1996; Mazzaferro et al., 2009). La supervivencia actuarial a los cuatro años de estos pacientes fue del 75% y el riesgo de recidiva tumoral fue del 8%. Además, se comprobó que los pacientes que superaban los criterios anteriormente mencionados en el análisis anatomopatológico de la pieza de hepatectomía tenían un mayor riesgo de recidiva tumoral y una supervivencia inferior. Tras la publicación del artículo original de Mazzaferro, diversos grupos confirmaron los buenos resultados del grupo de Milán. Por ello, los criterios de Milán son los límites más aceptados para el trasplante hepático, y diversas sociedades, como la Sociedad Española de Trasplante Hepático y la Asociación Española para el Estudio del Hígado, se adhieren a estos criterios (Herrero Santos, 2011).

Para que un paciente sea candidato a trasplante a través del Criterio de Milán se tiene que cumplir alguna de las dos siguientes premisas (Mazzaferro et al., 1996):

- El paciente solo tiene un tumor de un tamaño menor o igual a cinco centímetros de diámetro.
- El paciente puede tener hasta tres tumores, pero ninguno de ellos puede tener un tamaño mayor que tres centímetros de diámetro.

Existen evidencias de que los límites actuales para el trasplante hepático (criterios de Milán) pueden sobrepasarse, ya sea trasplantando a pacientes que superan ligeramente estos límites o trasplantando a pacientes que sobrepasaban esos límites. Probablemente, la inclusión de parámetros biológicos que permitan estimar la agresividad del tumor servirá en el futuro para hacer una mejor selección de los pacientes candidatos a trasplante. La expansión de los criterios de trasplante tiene como contrapartida un aumento de los potenciales candidatos a trasplante, lo que podría ocasionar un aumento de la mortalidad y/o de la progresión tumoral en lista de espera. De esta forma, quedan en contraposición el beneficio personal y el beneficio general, ya que un aumento de la mortalidad en lista de espera causaría una menor supervivencia por intención de tratamiento. El trasplante de donante vivo podría paliar este problema, en parte (Mazzaferro et al., 2009).

4.3.2 UCSF

En 2001, Yao et al. comunicaron los resultados de la Universidad de California, San Francisco (UCSF), en la que los límites para el acceso al trasplante fueron un nódulo tumoral de hasta 6,5 cm, o dos a tres nódulos menores de 4,5 cm si la suma de sus diámetros era menor de 8 cm. Poco después comprobaron que la supervivencia de los pacientes que superaban los criterios de Milán, pero se mantenían dentro de los criterios UCSF, era igual que la de los pacientes que cumplían los criterios de Milán (Yao et al., 2002; Yao et al., 2007).

4.3.3 UpToSeven

Los criterios *Up-To-Seven* se introdujeron hace varios años, pero todavía no existe un consenso acerca de su efectividad. Doscientos veinte pacientes se dividieron en tres grupos de acuerdo a las características de sus tumores: la supervivencia global de uno, tres, y cinco años y una tasa de supervivencia libre de tumor para el grupo de criterios de Milán fueron mayores que los de los criterios *Up-To-Seven*. Sin embargo, las tasas generales y libres de tumor de carcinoma hepatocelular de supervivencia de los pacientes avanzados eran mucho más bajos. Así que teniendo en cuenta que los pacientes en el grupo de los criterios *Up-To-Seven* mostraron una tasa de supervivencia considerable, pero menor en comparación con el grupo de criterios de Milán, por lo que los criterios *Up-To-Seven* deben usarse con cuidado y de forma selectiva (Mazzaferro et al., 2009).

Para que un paciente sea candidato a trasplante a través del Criterio de Milán se tiene que cumplir alguna de las siguientes premisas:

- El paciente tiene un tumor con un tamaño menor a seis centímetros de diámetro.
- El paciente tiene dos tumores, uno con un tamaño menor que seis centímetros de diámetro y el otro menor que cinco.
- El paciente tiene tres tumores, uno con un tamaño menor que seis centímetros de diámetro, otro menor que cinco y el otro menor que cuatro.
- El paciente tiene cuatro tumores, uno con un tamaño menor que seis centímetros de diámetro, otro menor que cinco, otro menor que cuatro y el otro menor que tres.
- El paciente tiene cinco tumores, uno con un tamaño menor que seis centímetros de diámetro, otro menor que cinco, otro menor que cuatro, otro menor que tres y el otro menor que dos.
- El paciente tiene seis tumores, uno con un tamaño menor que seis centímetros de diámetro, otro menor que cinco, otro menor que cuatro, otro menor que tres, otro menor que dos y el otro menor que uno.

4.3.4 TTV + AFP

Para determinar si un paciente es candidato a trasplante o no en base a este criterio únicamente se necesita el volumen total de tumores y el AFP. En el caso de que el paciente tenga un volumen menor o igual que 115 y que las cantidades de AFP estén por debajo de 400 será un candidato apto, en caso contrario no lo será (Toso et al., 2015).

Capítulo 5: Manual de usuario del panel Web HepApp

En el presente capítulo se va a explicar desde un punto de vista de manual de uso la interfaz gráfica y las posibilidades de interacción que esta ofrece al usuario. Se utilizan figuras con capturas de pantalla para acompañar a las explicaciones textuales. El navegador Web utilizado es *Mozilla Firefox* en su versión 61.0.2. El acceso a este panel Web se encuentra restringido a las personas que, desde él, gestionan el contenido que después se sincroniza y se muestra en la aplicación Android HepApp.

5.1 Sobre la organización del contenido

El contenido de HepApp se reparte entre ocho secciones compuestas a su vez por la combinación de recursos, módulos, recursos compuestos y categorías de imágenes. El contenido como tal se almacena en forma de recursos, y el resto de elementos son para proporcionar organización y jerarquía a los recursos dentro de una sección. Se pueden ver también a las secciones como los tipos de recursos que existen.

La sección Capítulos contendrá recursos del tipo PDF que se conocerán como capítulos. La sección Podcasts contendrá recursos que son vídeos de *YouTube*. La sección Figuras, que a su vez se compone de otras cinco subcategorías o subsecciones, contendrá recursos que son imágenes. La sección Recursos contendrá recursos del tipo marcadores a sitios Web externos. La sección *Cards*, *PubMed* e Información están formadas por un único recurso del tipo sitio Web externo. La sección Calculadoras es una sección fija o no configurable desde el panel Web, por lo que no contiene recursos. Su contenido está formado por varias calculadoras para un uso académico y de apoyo médico en escenarios reales.

Un módulo entonces es una agrupación de recursos. Como si de una asignatura se tratara, se utilizan para marcar bloques o partes distintas dentro de una sección o categoría. El uso de módulos aporta un nuevo nivel jerárquico en la organización de los recursos. Los módulos sólo son compatibles con las secciones Capítulos, Podcasts y la subsección Figuras de capítulos de la sección Figuras. El uso para el que están pensados es el de poder agrupar todos los recursos de las secciones con las que son compatibles que se refieren o forman una parte suficientemente diferenciada del contenido total. Así, por ejemplo, si se considera crear un módulo para agrupar los capítulos del uno al diez, este módulo debería tener también asociados todos los recursos podcast con las clases que se refieren a los capítulos mencionados y todos los recursos imagen de las imágenes que aparecen en dichos capítulos. Aunque un módulo agrupe recursos de distintos tipos, desde cada sección, al acceder al módulo, se mostrarán todos los recursos que tiene dicho módulo, pero sólo del tipo definido por esa sección. Por ello, al acceder a estas secciones mencionadas lo que se muestra directamente no es un listado de recursos sino un listado de módulos. Se debe tener en cuenta que, si un módulo no cuenta con recursos de uno de los tipos definidos por una sección, este no será listado al acceder a dicha sección. Es por ello que, aunque no debería ser algo frecuente, un módulo podría aparecer sólo en algunas secciones.

Un recurso compuesto es una agrupación de recursos (o recursos simples). La diferencia con respecto a los módulos es que los recursos compuestos sólo agrupan recursos simples del

tipo definido por la sección Figuras. Además, sólo pueden agrupar hasta un máximo de cuatro de estos recursos (para los módulos no hay definido un máximo). El uso para el que están pensados es el de agrupar unos pocos recursos imagen que van a presentar un tipo de interacción *switch*. Esta interacción es simple y consiste en poder cambiar entre cualquiera de los recursos simples que se han agrupado para formar el recurso compuesto. Se puede explicar mejor el concepto con un ejemplo: Dadas dos imágenes donde la segunda es la primera a la que se han hecho algunas pequeñas modificaciones, como por ejemplo unas anotaciones, la interacción *switch* consiste en permitir cambiar directamente de una imagen a otra de forma que se consigue resaltar fácilmente las diferencias entre las dos imágenes. De forma visual, es lo que puede ocurrir con los recursos imagen mostrados en las figuras **¡Error! No se encuentra el origen de la referencia.** y **¡Error! No se encuentra el origen de la referencia.**. La primera muestra un escenario normal y la segunda el mismo escenario, pero con hipertensión (enseñando también sus efectos adversos). Así pues, en este caso los recursos compuestos no están destinados a proporcionar un nivel extra de jerarquía, sino que agrupan recursos compatibles en base de la interacción descrita. Los recursos compuestos sólo son compatibles con la subsección Figuras interactivas de la sección Figuras. Por ello, en este caso se pueden utilizar ambos términos como sinónimos.

Con lo explicado hasta ahora, se puede comprobar que las subsecciones de la sección Figuras no sólo pretenden clasificar las imágenes en base a un subtipo que definen, sino que también los clasifican en función de la interacción que estos presentan y su relación con otras secciones. Las subsecciones que clasifican las imágenes definiendo un subtipo de imagen son las subsecciones Tabla de contenidos, Esquemas y Figuras de capítulos. Las subsecciones que clasifican las imágenes en función del tipo de interacción son las subsecciones Figuras interactivas y Pintando.

Las subsecciones Tabla de contenidos y Esquemas están pensadas para mostrar recursos del subtipo que definen respectivamente sus nombres. La subsección Figuras de capítulos primero lista los módulos y después aquellas imágenes que presentan relación con otras secciones. La subsección Pintando lista las imágenes sobre las cuales se puede hacer uso de la interacción de igual nombre que la subsección.

Por último, se debe mencionar que, aunque a veces pueda carecer de interés y no ocurra, un recurso imagen puede asociarse (y por lo tanto aparecer) a varias subsecciones de la sección Figuras.

Dado que el contenido de la aplicación Android se sincroniza con el contenido dispuesto en el panel Web, en la aplicación seguirá la misma organización y jerarquía aquí descritas.

5.2 El panel Web

En la pantalla inicial del panel Web sólo aparece el formulario de inicio de sesión de la figura Figura 15.

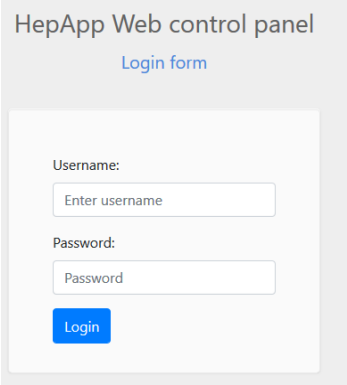

Figura 15. Formulario para iniciar sesión en el panel Web.

Tras introducir las credenciales de usuario correctas, se carga una pantalla que muestra todas las tablas con todo el contenido que se haya introducido hasta el momento en el sistema con usos anteriores de este panel Web. En la figura Figura 16 se puede ver el aspecto de esta pantalla.

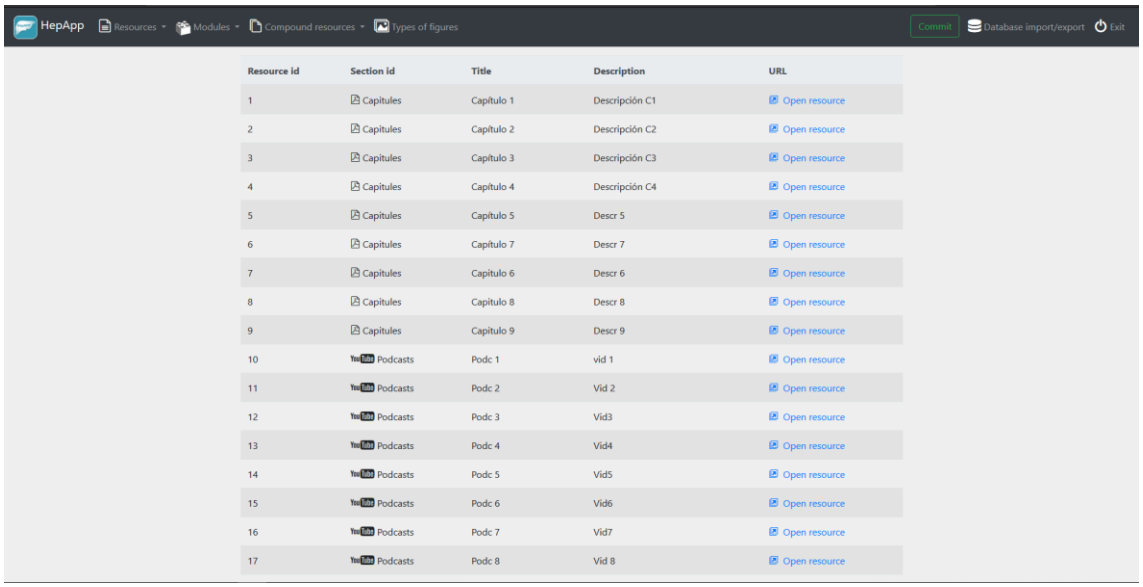

Figura 16. Pantalla de visión general y confirmación de la operación solicitada.

El panel Web presenta en la parte superior una barra de navegación que estará disponible desde todas las diferentes pantallas. En la figura Figura 17 se puede ver esta barra de navegación.

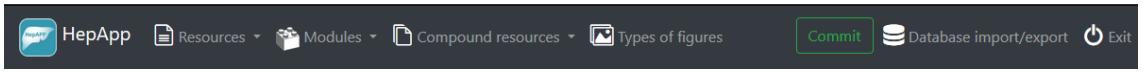

Figura 17. Barra superior de navegación del panel Web.

De izquierda a derecha se explican todos los controles disponibles en la barra:

1) El logotipo del proyecto que permite regresar a la pantalla de visión general de la figura Figura 16.
2) Gestión de los recursos, muestra un submenú desplegable de las operaciones disponibles relacionadas con los recursos. En la figura Figura 18 se puede ver el submenú.

3) Gestión de los módulos, muestra un submenú desplegable de las operaciones disponibles relacionadas con los módulos. En la figura Figura 18 se puede ver el submenú.
4) Gestión de los recursos compuestos, muestra un submenú desplegable de las operaciones disponibles relacionadas con los recursos compuestos. En la figura Figura 18 se puede ver el submenú.
5) Gestión de las subsecciones de la sección Figuras. Carga la pantalla que permite realizar operaciones de organización de los recursos imagen.
6) Control que permite aplicar al sistema todos los cambios y gestiones realizadas hasta el momento. Se explica con más detalle más adelante.
7) Carga la pantalla que permite realizar operaciones de copiar y restaurar de forma completa la base de datos en la que se almacena todo el contenido del proyecto.
8) Cierra la sesión y sale del panel Web regresando a la pantalla inicial con el formulario para iniciar sesión.

En la figura Figura 18 se muestran respectivamente los submenús de los controles 2, 3 y 4 de la barra superior de navegación.

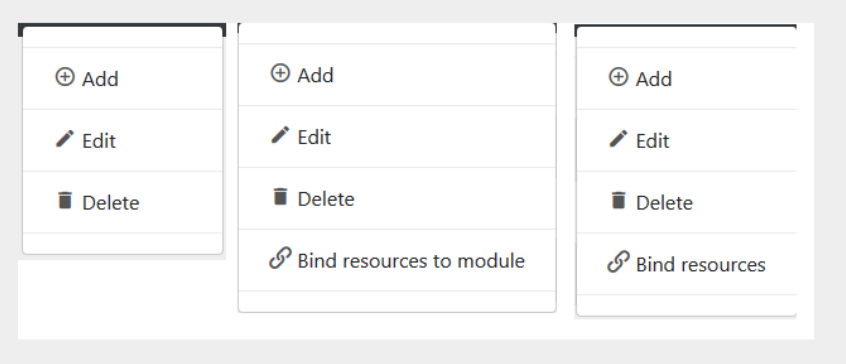

Figura 18. Submenús de operaciones disponibles.

Con la figura Figura 19 se ilustra la pantalla desde la que se puede llevar a cabo la acción de agregar un nuevo recurso al sistema.

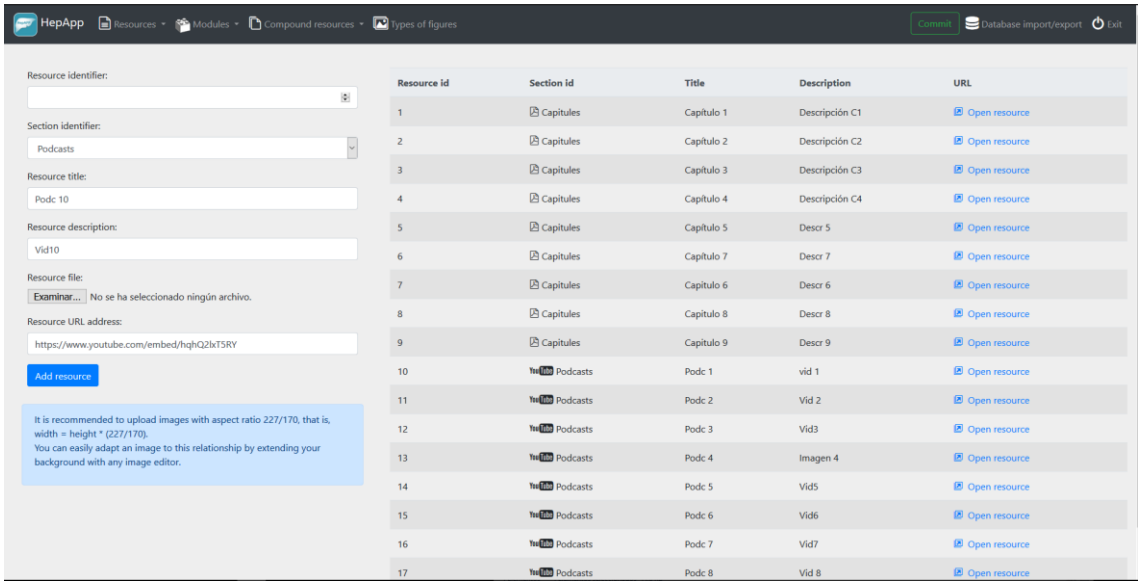

Figura 19. Pantalla para agregar un recurso.

En esta pantalla se encuentra en la parte superior izquierda el formulario para agregar un recurso. El campo para introducir el identificador del recurso habitualmente se dejará vacío para indicar al sistema que le asigne automáticamente un identificador. El motivo de este campo es que, si se desea, se pueda especificar el índice de un recurso que ya tuvo asignado un recurso y después fue eliminado. Según el tipo de recurso que se quiera agregar se elige una de las secciones de la lista desplegable. Especificar un título y una descripción es algo opcional, pero al menos, en el caso del título, siempre es recomendable. Al agregar un recurso PDF o un recurso imagen se debe utilizar el botón examinar para seleccionar un fichero local que se va a subir a la máquina servidora. Para el resto de secciones se utiliza en su lugar el último campo del formulario para colocar una dirección URL.

Debajo del formulario se indica una recomendación sobre la relación alto-ancho que deberían tener las imágenes que se suban para lograr el funcionamiento óptimo del sistema. Si no se utiliza la relación alto-ancho recomendada las imágenes podrían mostrarse mal en la aplicación Android tanto en los listados de imágenes como en el visualizador.

A la derecha del formulario se muestran en una tabla todos los recursos ya existentes en el sistema. En una columna de la tabla se encuentra un enlace que permite abrir cada recurso. A modo de ayuda se han utilizado diferentes iconos para reconocer de forma más cómoda y visual las secciones a las que pertenece cada recurso. Esta regla se puede tomar como general pues se han utilizado en la medida posible iconos en el diseño gráfico del panel para facilitar el reconocimiento rápido y visual de todos los elementos que se muestran.

En la figura Figura 20 se muestra la pantalla desde la que se puede llevar a cabo la acción de editar un recurso ya existente en el sistema.

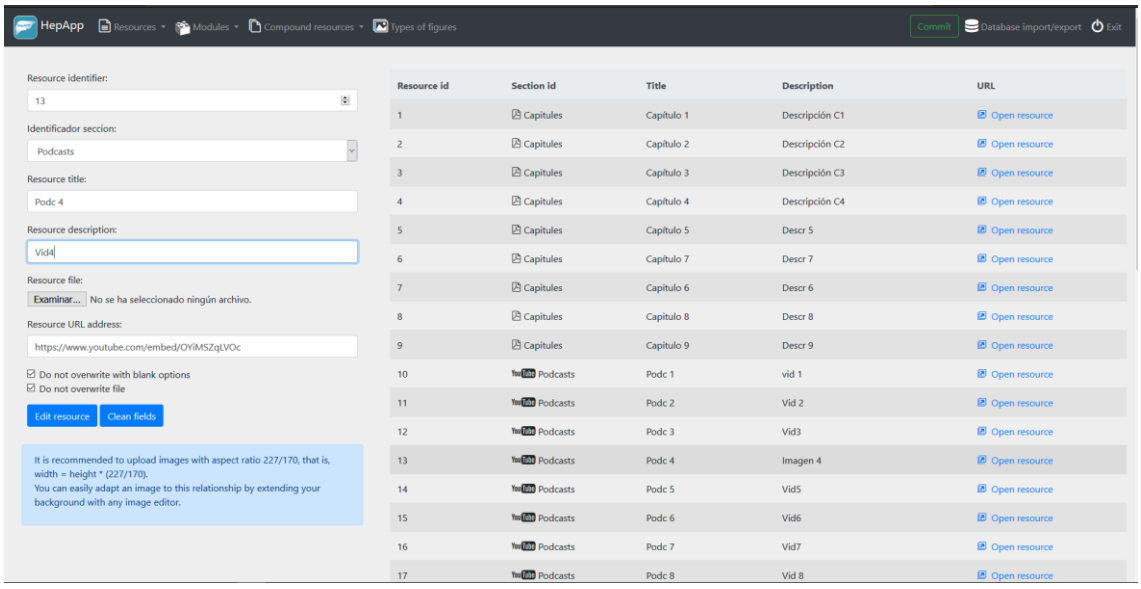

Figura 20. Pantalla para editar un recurso.

A simple vista parece que lo único que ha cambiado es sólo y de forma leve el formulario. Como novedad, ahora se puede pulsar sobre los elementos de la tabla de la derecha y el efecto que tendrá es el de rellenar los campos del formulario con los valores del recurso que se ha pulsado. De esta forma será más fácil editar un recurso.

Respecto al formulario, en esta ocasión sí es necesario utilizar el campo del identificador del formulario para especificar qué recurso se desea editar. El control para subir un nuevo fichero sigue siendo restringido sólo a las secciones Capítulos y Figuras, pero ahora también se puede utilizar el campo para direcciones URL con las secciones Capítulos y Figuras siempre que para estas se especifiquen direcciones URL que apunten a ficheros de recursos alojados en la máquina servidora que también aloja este panel Web. De forma adicional se cuenta con un botón para borrar los campos del formulario y dos casillas por defecto marcadas para controlar si se quiere que al editar un recurso no especificar un valor en un campo del formulario no implique borrar dicho campo del recurso que ahora cuenta con un valor asignado y para controlar si no se quiere sobrescribir con el fichero que se desea subir al ya existente en caso de que ambos cuenten con el mismo nombre. Para evitar borrar lo que no se quiere por un despiste, estas dos casillas se encuentran marcadas por defecto.

Para borrar un recurso del sistema, de nuevo se carga una pantalla similar donde ahora el formulario es el que se indica en la figura Figura 22 y en la tabla de la derecha se siguen pudiendo pulsar los elementos para rellenar los campos de los que consta el formulario. En el caso de esta operación, dado que implica el borrado de información, se muestra un diálogo para volver a confirmar que se desea hacer la eliminación.

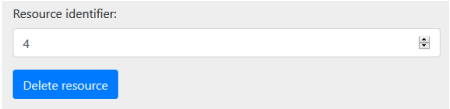

Figura 21. Formulario para eliminar un recurso.

Cada vez que se realiza una gestión, salvo que se produzca un error de validación, se regresa a la pantalla de la visión general del sistema y se muestra justo debajo de la barra superior de navegación un mensaje que confirma el resultado de la realización de la gestión hecha. Se indicará con color azul que la operación se ha realizado correctamente, se indicará con amarillo que la operación se ha realizado correctamente pero que no se han seguido todas las recomendaciones para el funcionamiento óptimo y se indicará con rojo que se ha producido un error al realizar la operación.

Las advertencias por incumplimiento de las recomendaciones que se pueden mostrar son la de la relación alto-ancho de las imágenes que se suban y la de que la dirección URL de que el vídeo de YouTube proporcionado no cuenta con la configuración de vídeo embebido (provoca que la visualización en la aplicación Android no sea óptima). Para obtener la dirección URL de un vídeo de YouTube con la configuración de vídeo embebido, en YouTube, con el vídeo abierto, se debe dar en el botón compartir que hay debajo del vídeo, después al botón insertar de la ventana *pop-up* que se abre y finalmente copiar la dirección URL del vídeo que aparece entrecomillado en el código de la parte derecha con un formato similar a *https://www.youtube.com/embed/xxxxxx* donde las *x* se corresponderán con el identificador del vídeo.

En el caso de los módulos y los recursos compuestos, las operaciones de agregar, editar y eliminar cuentan con pantallas análogas a las vistas para los recursos simples solo que con formularios simplificados puesto que en estos elementos sólo se puede especificar un título y una descripción (a parte del identificador). En las pantallas de estas operaciones también se muestra en la parte derecha una tabla que lista los módulos y los recursos compuestos que

existen actualmente en el sistema. En las figuras Figura 22 y Figura 23 se pueden ver estas tablas (de las tres presentes sólo sería la de arriba a la derecha).

Los módulos y los recursos compuestos cuentan con una cuarta pantalla de operación. En la figura Figura 22 se puede ver la pantalla para asociar los recursos simples con los módulos.

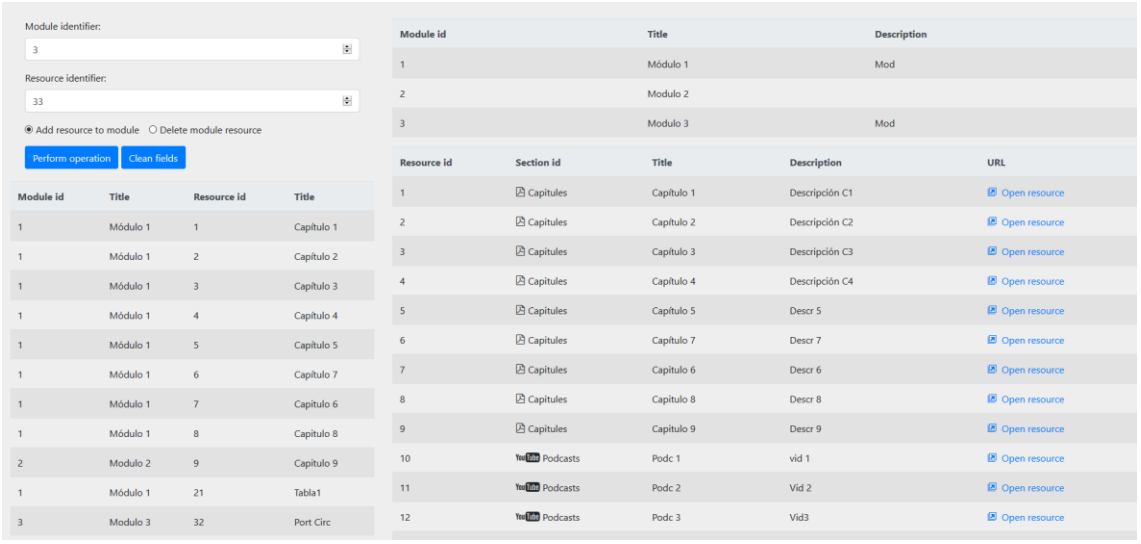

Figura 22. Pantalla de gestión de las relaciones entre recursos y módulos.

Arriba a la izquierda se encuentra el formulario con el que se pueden añadir y eliminar relaciones entre recursos simples y módulos según la operación que se seleccione (ambas son excluyentes y no se pueden seleccionar las dos simultáneamente). En la tabla de la parte superior derecha se listan los módulos existentes en el sistema. Al pulsar los elementos de esta tabla se rellenará el campo del identificador de módulo del formulario con el del módulo seleccionado. Debajo de esta tabla se encuentra la tabla que lista los recursos existentes en el sistema. Al pulsar los elementos de esta tabla se rellenará el campo identificador de recurso del formulario con el del recurso seleccionado. Debajo del formulario se encuentra la tabla que lista las relaciones entre recursos y módulos que existen registradas en el sistema. Al pulsar los elementos de esta tabla se rellenarán ambos campos de identificadores del formulario.

Para gestionar las relaciones entre recursos compuestos y recursos simples, como se observa en la pantalla de la figura Figura 23, el funcionamiento es análogo al descrito en el párrafo anterior. Se debe recordar que en este caso sólo se pueden asociar recursos de tipo imagen al recurso compuesto.

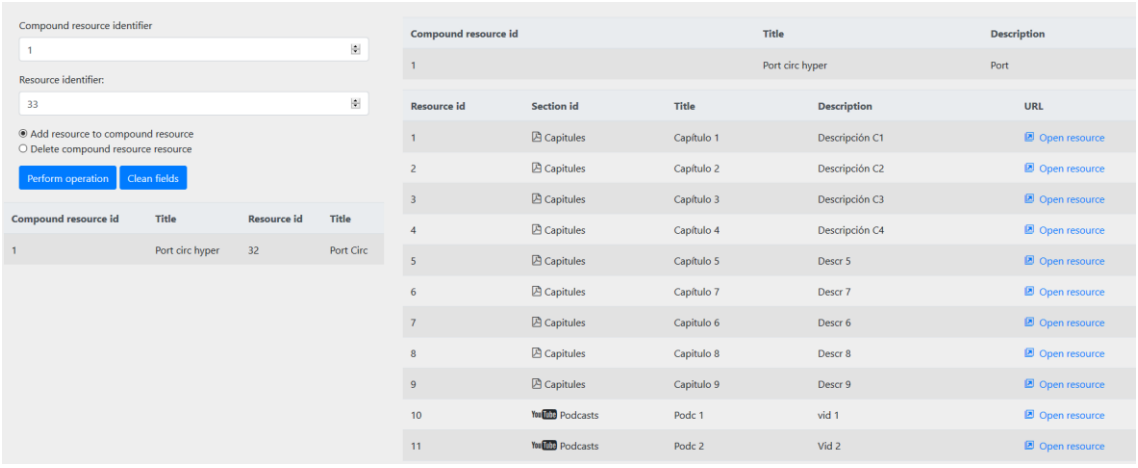

Figura 23. Pantalla de gestión de las relaciones entre recursos y recursos compuestos.

En referencia a las subsecciones de la sección Figuras, como se muestra en la figura Figura 24, se pueden realizar las operaciones de gestión desde esta pantalla.

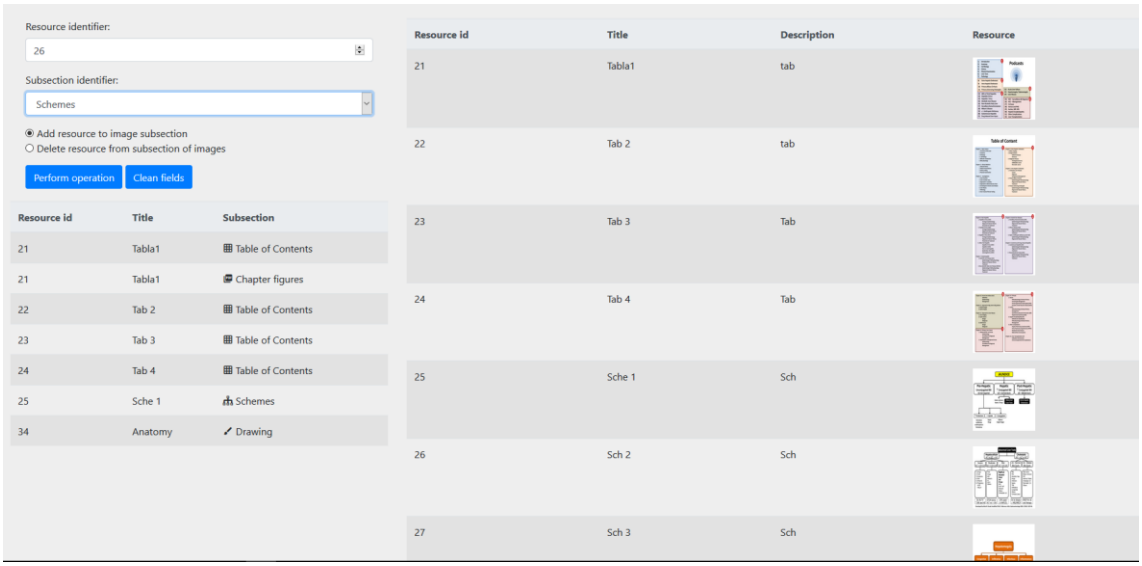

Figura 24. Pantalla de gestión de las asociaciones entre recursos y subsecciones de imágenes.

En este caso sólo se muestran dos tablas, una a cada lado, y como algo particular, dado que sólo se pueden asociar los recursos simples de tipo imagen, en la tabla de recursos de la derecha ahora se muestran miniaturas de las imágenes en lugar de enlaces para poder abrirlos en una nueva pestaña del navegador. Las diferentes subsecciones, a excepción de la subsección Figuras interactivas que se gestiona con los recursos compuestos, se encuentran en la lista desplegable del formulario. También, en esta pantalla, los elementos de las tablas se pueden pulsar para rellenar automáticamente los campos del formulario. Se debe recordar que nada impide asociar un mismo recurso imagen con varias subsecciones diferentes.

Con el control de la barra superior de navegación con el icono de una base de datos se carga la pantalla de la figura Figura 25.

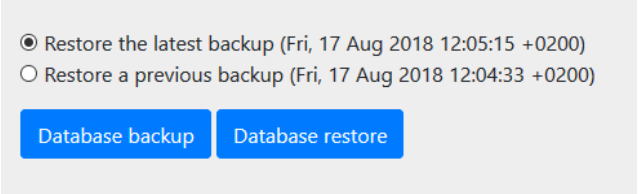

Figura 25. pantalla de gestión de las copias de seguridad de la base de datos del sistema.

Desde esta pantalla se pueden llevar a cabo las operaciones de crear una copia de seguridad de la base de datos y restaurar la base de datos de forma completa. El sistema no permite almacenar más de dos copias de seguridad completas y por ello con cualquiera de las dos operaciones se muestra un diálogo de confirmación extra ya que en ambas se podría perder de forma accidental información, pues crear una nueva copia de seguridad siempre implica eliminar la más antigua de las dos que pueda tener almacenadas el sistema y restaurar una copia implicaría perder todos los cambios hechos después de la fecha de la copia de seguridad que se va a restaurar. En los controles mutuamente excluyentes que permiten seleccionar la copia de seguridad a restaurar se muestra de forma detallada la fecha de cada una de las dos copias de seguridad que tiene almacenadas actualmente el sistema. La operación de creación de una nueva copia de seguridad omite la selección de estos controles mutuamente excluyentes.

Respecto al uso del panel Web, finalmente se deben comentar otras consideraciones importantes que se deben tener en cuenta:

- Control *Commit* de la barra superior de navegación: Todas las gestiones que se realizan desde las otras pantallas del panel no se aplican y pasan a estar disponibles para la aplicación Android hasta que no se pulsa este control. El motivo para que exista este control es que se puedan realizar tranquilamente todas las gestiones del contenido que se deseen para aplicarlas después todas de golpe de forma segura y así evitar que un usuario de la aplicación Android pueda ejecutar la sincronización de la aplicación a mitad de las gestiones de forma que obtenga como resultado una base de datos incoherente que pueda causar el malfuncionamiento de la aplicación. Por ello, se debe tener en cuenta que se deben hacer todas las gestiones deseadas primero y después aplicarlas a través de este control.
- Las gestiones hechas no se pueden revertir se hayan aplicado o no sobre el sistema.
- El panel Web expira la sesión de forma automática tras varios minutos de inactividad para evitar ataques de robo de sesión. Esto no supone ningún problema porque cerrar sesión no implica perder las gestiones realizadas, aunque no se hayan aplicado. Todas las gestiones que se realicen quedan permanentemente guardadas en el sistema, aunque sólo se apliquen al pulsar el botón *Commit*.
- Todos los recursos de las secciones Capítulos y Podcasts deben estar obligatoriamente asociados al menos a un módulo o esto podría provocar en la aplicación Android el malfuncionamiento de estas dos secciones y de la barra lateral de navegación que, como se encuentra en todas las pantallas, además, podría provocar que se muestren constantemente molestas notificaciones de error.

- Del mismo modo, todos los recursos imagen de la subsección Figuras de capítulos deben estar obligatoriamente asociados al menos a un módulo y a dicha subsección.
- El proceso de la realización de una copia de seguridad completa de la base de datos supone la pérdida de todas las gestiones hechas desde el panel Web que no se hayan aún aplicado de forma permanente haciendo uso del control *commit* de la barra superior de navegación. El proceso de restauración de una copia de seguridad completa de la base de datos también supone la pérdida de todas las gestiones hechas desde el panel Web que no se hayan aún aplicado de forma permanente haciendo uso del control *commit* de la barra superior de navegación.

Capítulo 6: Descripción técnica y sobre el desarrollo del panel Web HepApp

El panel Web HepApp es la aplicación Web que permite gestionar los contenidos que se van a mostrar en la aplicación Android. Para ello, se ofrece a la aplicación a modo de servicio Web a través de una *API REST* la posibilidad de obtener todo este contenido desde una máquina servidora para cargarlo y mostrarlo al usuario en la aplicación Android. La implementación se ha desarrollado utilizando *HTML 5*, *CSS 3*, *JavaScript*, el *framework Bootsrap 4* y la biblioteca *JQuery 3.3.1* para la parte *frontend* y para la parte *backend* la distribución de *Apache XAMPP* en su versión 7.2.6 que incluye *PHP 7.2.6*, *MariaDB 10.1.33*, *phpMyAdmin 4.8.1* y el servidor Web *Apache 2.4.33*.

PHP permite la implementación tanto en un estilo orientado a procedimientos como en un estilo orientado a objetos. En este caso, dado que no se trata de un proyecto con una excesiva complejidad ni longitud en lo que a implementación se refiere, se ha optado por el estilo orientado a procedimientos.

Respecto a la parte *frontend*, esta se explicará de pasada y se la hará alusión cuando sea relevante para explicar alguna parte del *backend*.

El sistema se puede decir que se compone de cuatro partes diferenciadas no independientes que son el *frontend*, el *backend*, la *API REST* y la base de datos. Sin embargo, en este caso el sistema se va a describir según las distintas partes funcionales que se presentan a continuación.

6.1 Configuraciones base y estructura del sistema

El panel Web cuenta con un fichero de nombre *core.php* que reúne en un único lugar algunas configuraciones básicas globales de todo el sistema. Utilizando la función de PHP *include_once()*, este fichero es el único que se va a incluir sólo en todos los demás ficheros PHP que no vayan a ser a su vez incluidos en otros como, por ejemplo, los que contienen un documento HTML. Sólo en este caso, se deberá especificar como argumento la ruta relativa al fichero. Algunas de las configuraciones básicas son el nombre del directorio raíz que va a contener todo el sitio Web y la ruta absoluta del sistema hasta este directorio. Aunque se ve más adelante, se especifica también el valor de un temporizador de expiración de sesión y se configura qué errores debe informar PHP.

La ruta absoluta se calcula, por lo que no se necesita configurar manualmente. En el resto del código se va a utilizar esta ruta absoluta cuando se vaya a especificar un fichero como argumento de la función *include_once()*. Esto permite solucionar un problema con las rutas relativas que podría hacer no funcionar al sistema. Si se cuenta con un fichero *a.php*, un fichero *b.php*, un directorio *c*, un directorio *d*, un fichero *e.php* en el directorio *c* y un fichero *f.php* en el directorio *d*, se puede dar una situación en la que si *b.php* incluye a *f.php* usando rutas relativas *d/f.php*, si después *a.php* incluye a *b.php* usando rutas relativas y *e.php* incluye a *b.php* usando rutas relativas, no se producirá ningún problema en *a.php* pero se producirá un error en *e.php*

porque desde su ubicación no podrá resolver la ruta relativa *d/f.php*. Por ello, si se usan rutas absolutas para todos los argumentos que se utilicen para llamar a *include_once()*, este problema se resuelve.

Lo anterior habla sobre las rutas y los argumentos que se utilizan con la función *include_once()*, pero el correcto uso de esta función junto a una separación adecuada de las partes del sistema en distintos ficheros PHP es clave para logar un sistema organizado y ahorrar repetir código (sobre todo de cara a evitar tener que editar el mismo código en múltiples ficheros con el riesgo de no hacer en todos la misma modificación). Lo que se busca y en parte se logra es que los grupos de funciones o partes funcionales del sistema al quedar cada una en un fichero se puedan incluir siempre que se necesiten como si se trataran de módulos que se habilitan o deshabilitan.

6.2 Definición de usuarios y conexión con la base de datos

Para el uso de la base de datos por el sistema se han definido tres cuentas de usuario en el sistema gestor de base de datos. Estos usuarios se diferencian por los permisos que tienen sobre la base de datos y por las partes del sistema desde las que se va a utilizar cada uno. Respecto a la conexión con la base de datos, se define en un fichero el código necesario para comprobar que una conexión con la base de datos se ha establecido correctamente y establecer algunos ajustes iniciales como el uso del conjunto de caracteres *UTF-8*. Después se definen tres ficheros, uno por cada usuario, que abren la conexión con la base de datos usando las credenciales propias de cada usuario y después hacen uso del otro fichero que comprueba errores y establece ajustes iniciales. De esta forma, en el resto del sistema, cuando se necesite hacer uso de la base de datos se incluirá alguno de estos tres ficheros según qué usuario se necesite en cada momento. Para el uso de la base de datos, tanto en estos ficheros como en otros que se explican en subapartados posteriores se ha utilizado la extensión *mysqli* de PHP.

Se deben mencionar dos detalles importantes. El primero, que también es general para todo el sistema, es que los ficheros propios de cada usuario incluyen al fichero de comprobación de errores y establecimiento de ajustes iniciales utilizando la variable *$directorioRaiz* definida en *core.php* aunque no incluyan a este fichero. Esto se debe a que el uso pensado para estos ficheros de cada usuario es el de ser siempre incluidos en otros ficheros que ya tendrán definida dicha variable porque ellos sí incluyen a *core.php*. Si los ficheros propios de cada usuario tuvieran que incluir a *core.php* para usar la variable *$directorioRaiz* para incluir a su vez al fichero de comprobación de errores y establecimiento de ajustes iniciales se seguiría sin estar resolviendo el problema planteado antes en el subapartado anterior. Lo segundo, es que en cada caso se utilizará el fichero del usuario que con las mayores restricciones posibles permita realizar las operaciones que se necesitan. Lo usuarios definidos son *admin*, *webpannel* y *androidclient*. Aunque se explica más adelante el primer usuario se utilizará sólo al copiar y restaurar la base de datos, el segundo se utilizará en el panel Web y el tercero se utilizará para el servicio Web ofrecido a la aplicación Android a través de la *API REST*.

6.3 Gestión de las sesiones

El uso de todo el panel Web está restringido a disponer una cuenta de usuario válida con la que poder iniciar sesión en el sistema. A parte de esta restricción, el registro de nuevas cuentas de usuario sólo es posible con una petición al administrador del sistema que deberá hacer el registro de forma manual. En un principio, sólo se espera que exista una única cuenta de usuario para utilizar el panel Web.

Para el uso de las sesiones, se debe llamar al principio de cada fichero PHP a la función *session_start()*. Para la gestión de las sesiones se definen los ficheros *login.php*, *loginRequired.php* y *logout.php*.

El fichero *login.php* es el que se encarga de procesar el formulario de la pantalla inicial o de inicio de sesión. Tras incluir al fichero *core.php*, después al fichero del usuario *webpannel*, seleccionar la base de datos con la función *mysqli_select_db()* e incluir al módulo de validación y saneamiento de datos de entrada que se explica más adelante, obtiene correctamente los datos recibidos del formulario, cambia todas las mayúsculas a minúsculas del nombre de usuario con *strtolower()* y prepara, ejecuta y obtiene el resultado de una consulta a la base de datos solicitando los datos de la cuenta de usuario con el nombre de usuario especificado.

Con los datos recibidos, primero comprueba que la cuenta existe y en caso de existir ejecuta la función hash *SHA512* sobre la clave de usuario recibida del formulario. El resultado lo compara con lo recibido de los datos de la cuenta del usuario de la base de datos. Si el hash calculado coincide con el almacenado en la base de datos se considera la sesión iniciada y establece los datos de la sesión en el array asociativo *$_SESSION*. Uno de esos datos de sesión es la hora actual del sistema que se obtiene con la función *time()*. Esto permitirá implementar un mecanismo básico de seguridad basado en un temporizador que se ve más adelante. Finalmente se redirige a la pantalla de la visión general del panel Web haciendo uso de la función *header()*.

Todos los ficheros que constituyen pantallas del panel Web, es decir, que producen una salida por pantalla (no la produce el fichero incluido aunque llame a *echo()*, sino aquel en el que este está incluido), incluirán, justo después de llamar a *session_start()* e incluir a *core.php*, al fichero *loginRequired.php*. Este fichero primero incluye al fichero *sessionExpirationHandler.php* que se ve más adelante y después simplemente comprueba que si no está establecido con un valor adecuado un *flag* en el array *$_SESSION* o la sesión ha expirado de forma automática tras un determinado tiempo de inactividad, se llama a la función *exit()* de PHP tras mostrar un mensaje de inicio de sesión requerido con un enlace a la pantalla inicial con el formulario de inicio de sesión. El *flag* del array *$_SESSION* se ha establecido en el fichero *login.php*. De esta forma, aunque se conozca la dirección URL de alguna de las pantallas privadas del panel Web, al cargarla directamente desde el navegador se hará efectiva la restricción de necesitar iniciar sesión.

Finalmente, el fichero *logout.php* es aquel que se carga tras pulsar el control de cerrar sesión ubicado a la derecha del todo en la barra superior de navegación del panel Web. Este fichero simplemente llama a las funciones *session_unset()* y *session_destroy()* para realizar después una redirección con *header()* a la pantalla inicial con el formulario de inicio de sesión.

6.4 Estructura de la base de datos

Antes de continuar explicando el funcionamiento del sistema, para facilitar la comprensión se va a explicar la estructura de las bases de datos. Para el diseño de las tablas de las bases de datos se ha empleado la normalización de las bases de datos aplicando hasta la tercera formal con el objetivo de reducir posibles inconsistencias que pudieran darse y facilitar el uso de la base de datos a través de las consultas que se realicen. A continuación, se listan y explican las tablas de las que consta cada una de las dos bases de datos utilizadas por el sistema. Junto a cada tabla se mostrará la consulta con la que esta se crea y que define su estructura.

La base de datos de nombre *hepapp* cuenta con las siguientes tablas:

- La tabla *recurso* que almacenará los recursos del sistema.
- La tabla *modulo* que almacenará los módulos del sistema.
- La tabla *recursoModulos* que almacenará las relaciones o asociaciones entre los recursos y los módulos del sistema.
- La tabla *recursosSubsecciones* que almacenará las relaciones o asociaciones entre los recursos imagen y las subsecciones de la sección Figuras.
- La tabla *recursoCompuesto* que almacenará los recursos compuestos del sistema.
- La tabla recursoRecursosCompuestos que almacenará las relaciones o asociaciones entre los recursos simples y los recursos compuestos del sistema.
- La tabla *usuarios* que almacenará las cuentas de usuario del panel Web.
- La tabla *controlversiones* que almacenará la versión resultante de la última actualización de las tablas de la base de datos y de la propia base de datos.
- La tabla *versionFicherosRecursos* que almacenará la versión de los ficheros asociados a los recursos.

La base de datos *hepapptemp* cuenta con sólo algunas de las tablas de la base de datos *hepapp* con la misma estructura. Las tablas que forman esta base de datos son *recurso*, *modulo*, *recursoModulos*, *recursosSubsecciones*, *recursoCompuesto*, *recursoRecursosCompuestos* y *versionFicherosRecursos*.

Las tablas *controlversiones* y *usuarios* necesitan contenido inicial antes de la puesta en marcha del sistema. Así pues, para la tabla *usuarios* se debe incluir un mínimo de una cuenta de usuario. En el caso de la tabla *controlversiones* se deben crear varias entradas todas con una

misma versión inicial. Se puede introducir como versión inicial “a” para los identificadores del 0 al 6 ambos incluidos. El identificador 0 se reserva para definir la versión de la propia base de datos y los identificadores del 1 al 6 para las tablas *recurso*, *modulo*, *recursoModulos*, *recursosSubsecciones*, *recursoCompuesto* y *recursoRecursosCompuestos*. Las correspondencias entre los identificadores y las tablas se explican más adelante.

Respecto a los nombres de las tablas, por mejorar la legibilidad se han escrito utilizando una mayúscula en el inicio de cada palabra que se usa para formar el nombre de algunas tablas. Sin embargo, en el sistema realmente se utilizan todos los nombres en minúsculas.

En todas las tablas de ambas bases de datos se ha utilizado la codificación de caracteres *UTF-8* especificando el cotejamiento *utf8_unicode_ci*.

6.5 Imprimir tablas de datos

El fichero *imprimirTablasDatos.php* es uno de los módulos que implementa funciones para imprimir las diferentes tablas que se muestran en las pantallas del panel Web. Este módulo se incluirá con *include_once()* cuando se necesite mostrar algunas tablas de datos en la pantalla.

Así pues, hay un par de funciones por tabla diferente de todas las que se muestran en las imágenes de las pantallas del panel Web en el capítulo *Manual de usuario del panel Web HepApp*. Respecto a la primera función, cada una ejecuta la consulta SQL apropiada (a veces es una consulta formada por varias anidadas) sobre la base de datos para obtener todos los elementos (recursos, módulos, relaciones, etc…) con los que conformar la tabla y después haciendo uso de la función *echo()* y *stripslashes()* imprimen la tabla en el lugar desde el que se llaman a estas funciones. Con las consultas SQL, de los elementos a mostrar sólo se obtienen los campos que se desean visualizar en vez de obtenerse todos. Estas funciones reciben en forma de *flag* como argumento si deben utilizar o no la segunda función asociada a la hora de imprimir las filas de las tablas. En caso de indicar que se usen, el efecto que tienen estas segundas funciones asociadas es el de establecer para cada fila el atributo *onclick* de *HTML* con el valor para que se llame a una función *JavaScript*, pero en cada fila con los argumentos correctos para proporcionar la funcionalidad de que al pulsar cada fila se rellenen los campos del formulario (sólo algunos o todos) de la operación correspondiente en cada pantalla. Para el aspecto de las tablas y el efecto cebra se utilizan las clases *CSS table*, *table-striped*, *table-hover* y *thead-light* predefinidas en el *framework Bootstrap*.

Por otro lado, se implementan un par de funciones, que utilizan las funciones que imprimen las tablas, para sustituir los identificadores de las secciones y de las subsecciones de imágenes por un icono y nombre propios para cada una.

En el caso de la tabla que lista todos los recursos, en lugar de mostrar las direcciones URL directamente, estas se utilizan para formar un elemento ancla (etiqueta "*a*") *HTML* o enlace. En la tabla que sólo lista los recursos imagen, la dirección URL se utiliza para formar un elemento *img* de *HTML* y así mostrar una miniatura de las imágenes.

6.6 Protección frente al cambio de los identificadores de los campos de formulario

El fichero *nombresCamposFormularios.php* sólo define un array asociativo que cuenta con una entrada o elemento por cada formulario distinto que existe en el panel Web. Cada una de estas entradas se asocia con un nuevo array asociativo. Cada uno de estos nuevos array asociativos vinculados a un formulario distinto cuentan con una entrada o elemento por cada campo con el que cuenta el formulario. Se puede comprobar que la finalidad del array definido en este fichero es el de definir un nuevo identificador para cada uno de los identificadores *frontend* de cada campo de cada formulario distinto, así como almacenar las asociaciones entre todos estos identificadores. El objetivo que se pretende con esto es triple. Por un lado, se separa y protege frente al cambio al *backend* de los cambios en los identificadores del *frontend* ya que el *backend* va a utilizar sólo los nuevos identificadores (las claves de los array asociativos) que nunca van a cambiar. Al centralizar todo en un único fichero, si en el futuro se cambian los identificadores del *frontend* sólo habrá que cambiar de forma coherente estos identificadores también en el fichero *nombresCamposFormularios.php* sin tener que ir haciendo cambios en varias partes distintas de otros ficheros del *backend*. Por otro lado, precisamente esto permite que se pueda declarar un único bucle con pocas líneas de código que permita obtener al procesar un formulario sólo los datos de los campos de ese formulario. Lo importante es que ese único bucle es genérico y sirve para todos los distintos formularios, lo cual evita tener que realizar una implementación propia para cada formulario distinto. Finalmente, se logra aumentar la seguridad del sistema ya que se ha logrado limitar los posibles identificadores que pude reconocer el sistema a sólo los necesarios de forma que cualquier modificación malintencionada y premeditada de los identificadores *frontend* no supondrá ningún problema porque equivaldrá a recibir un valor vacío que fácilmente será procesado por la parte encargada de la validación generando el correspondiente error esperado y controlado.

Como efecto, se obtiene una separación parcial de dependencias entre *frontend* y *backend* en algo necesario, aunque inevitablemente se encuentren bastante ligados para otras cosas.

Aunque esta idea se ha expresado aplicada de forma concreta a los formularios del panel Web, también se ha desarrollado y aplicado de forma análoga en otros puntos del sistema en los que se ha considerado necesario.

6.7 Validación y saneamiento de los datos de entrada

El fichero *validarLimpiarDatos.php* es uno de los módulos que implementa funciones para obtener, validar y sanear los datos introducidos en alguno de los formularios y que se reciben para poder procesarlos y llevar a cabo la operación solicitada. Este módulo se incluirá con *include_once()* cuando se necesite validar y obtener los datos introducidos en un formulario.

Como se ha indicado en el subapartado anterior, la función *obtenerDatosFormulario()* recibe como argumento el identificador del formulario del que se van a obtener los datos introducidos haciendo uso del array asociativo del fichero *nombresCamposFormularios.php*. Además, esta función comprueba que los datos recibidos se han obtenido a través del método *HTTP POST* de un formulario procedente del propio sitio Web del panel Web.

En el fichero también se definen varias funciones generales de validación y saneamiento de los datos recibidos que se basan en las funciones *trim()*, *explode(';', $campo)*, *strlen()*, *addslashes()*, *htmlspecialchars()*, *preg_match($patron, $campo)*, *is_numeric()*, *stripslashes()* y *isset()* de PHP. Estas funciones permiten, comprobar si un campo requerido se encuentra vacío, proteger de un intento de inyección SQL basado en varias sentencias, comprobar que un campo no excede cierta longitud máxima, limpiar espacios en blanco, quitar caracteres escapados, sustituir caracteres especiales por entidades *HTML* pues pueden tener significado especial en *HTML*, comprobar si una cadena se encuentra vacía, validar un nombre de usuario con expresiones regulares, validar un identificador entero y escapar caracteres especiales como las barras invertidas entre otros.

Sobre todas estas funciones se construyen otras ya más avanzadas y particularizadas a los formularios que se pueden utilizar desde las diferentes pantallas del panel Web como son las funciones *validarInicioSesion()*, *validarIdentificadorSeccion()*, *validarRecurso()*, *validarModulo()*, *validarRecursoCompuesto()*, *validarRecursoSubseccion()* y *validarIdentificadorSubseccion()*. Como sus propios nombres indican permiten comprobar y validar los diferentes elementos manejados por el sistema, así como comprobar correctamente que los identificadores de las listas de secciones y subsecciones son correctos. Estas funciones reciben argumentos y con ellos controlan las posibles variaciones de validación como por ejemplo el que para añadir un recurso el identificador no sea un campo obligatorio, pero sí lo sea al editar o eliminar un recurso. Se debe destacar la gran efectividad de las expresiones regulares con los patrones que se utilizan bien formados.

6.8 Gestión y realización de las operaciones

El fichero *realizarOperacion.php* es el que se encarga de procesar los diferentes formularios de las pantallas privadas del panel Web. Tras incluir al fichero *core.php*, después al fichero *loginRequired.php*, incluir el módulo de validación y saneamiento de datos de entrada del subapartado anterior, incluir el fichero del usuario *webpannel*, seleccionar la base de datos con la función *mysqli_select_db()* e incluir al módulo de aplicación de gestiones que se explica más adelante, obtiene correctamente de un campo oculto de los formularios la operación que se solicita realizar y deduce a partir de esta con una estructura *switch* las operaciones que debe

realizar para llevarla a cabo. De esta forma, cualquier operación no esperada por el sistema no contará con una entrada *case* en la estructura *switch* y se tomara la acción por defecto que es redireccionar a la pantalla de visión general del panel Web.

Una vez seleccionadas las operaciones a realizar según el tipo de operación o gestión solicitada, se obtienen los datos introducidos en el formulario con *obtenerDatosFormulario()* pasando el identificador correspondiente, después llama a las funciones de validación y saneamiento del módulo visto en el subapartado anterior necesarias para esa operación y finalmente llama a la función que efectúa la operación o gestión solicitada.

Por ello, este fichero define una función para cada operación o gestión posible además de otras varias funciones más generales sobre las que se apoyan estas funciones particularizadas a cada operación. Se debe tener en cuenta que el módulo de validación y saneamiento de datos de entrada realiza lo que se puede entender como un primer nivel de validación que se refiere a los campos y tipos de datos introducidos en cada uno, pero no comprueba que los valores introducidos a nivel lógico tengan sentido. Este segundo nivel de validación se implementa en las funciones implementadas en este fichero.

En referente a la gestión de los recursos, las validaciones lógicas varían según la sección seleccionada del recurso. Para las secciones *Cards*, *PubMed* e Información se comprueba que sólo puede existir un único recurso asociado a dichas secciones. En el caso de la sección Podcast se comprueba que la dirección URL especificada se trata de un vídeo de *YouTube* y si cumple la recomendación de contar con la configuración de vídeo embebido. En el caso de las secciones Capítulos y Figuras se comprueba que el fichero a subir cuenta con el formato correcto (respectivamente PDF para la primera y los formatos PNG, JPEG y JPG para la segunda) teniendo en cuenta si ya existe en el sistema un fichero con el mismo nombre y si se permite su sobrescritura (algo que es configurable desde el formulario), comprueba que el tamaño del fichero no sea superior al máximo permitido y comprueba la recomendación del ratio alto-ancho para los ficheros imagen. Además, según se trate de un fichero PDF o de una imagen el fichero se guarda en uno u otro directorio y se calcula el nombre del fichero como un hash del nombre original del fichero. Esto evita problemas de espacios y caracteres especiales presentes en el nombre del fichero de forma que se asegura que no vaya a causar problemas a la aplicación Android. También se comprueba al agregar un recurso si ya existe uno con el mismo identificador y en el resto de gestiones se comprueba si no existe un recurso con el identificador especificado. Al subir un fichero agregando un nuevo recurso, se registra en la base de datos la versión inicial del recurso, de forma que cuando se edita el fichero asociado a dicho recurso sea actualizando el mismo fichero o sea cambiando su dirección por la de otro fichero distinto también presente en el sistema se incrementa la versión registrada del recurso. Consecuentemente debe validar que la nueva dirección URL se refiera a un fichero local del sistema. A parte de todas estas validaciones lógicas, después se conforma la sentencia SQL de la operación solicitada y se intenta su ejecución.

Es importante decir que las consultas SQL se intentan ejecutar sobre la base de datos *hepapptemp* y que sólo en caso de efectuarse con éxito también se registran después otras cosas como la versión del recurso como se ha dicho, las tablas afectadas por la consulta y la consulta ejecutada con éxito. Aunque se explica mejor más adelante, esto se guarda para luego poderse

aplicar las mismas gestiones sobre la base de datos *hepapp* porque esta es la que usa la aplicación Android para la sincronización de sus contenidos.

Cuando las operaciones se refieren a establecer o eliminar relaciones entre otros elementos del sistema se debe comprobar que todos los elementos implicados existen en el sistema, que en el caso de establecer la relación que esta no exista ya y en el caso de eliminar la relación comprobar que esta exista en el sistema. Por otro lado, al establecer relaciones se debe comprobar que estas son entre elementos compatibles. Sería por ejemplo el caso de los recursos compuestos que sólo pueden tener asociados recursos simples de imagen.

Para el resto de elementos, de forma análoga, también se realizan validaciones lógicas, se ejecutan consultas y se registran datos para aplicar después sobre la otra base de datos. En el caso concreto de los recursos compuestos se debe comprobar también al agregarle un recurso simple que no se exceda el máximo impuesto de hasta cuatro recursos simples por recurso compuesto.

Cuando todos los valores introducidos son válidos y coherentes con la operación o gestión solicitada, si no se produce ningún problema inesperado al llevar a cabo la gestión y todo resulta exitoso se produce una redirección a la pantalla de visión general del panel Web mostrando una notificación del estado correcto de la operación y mostrando si fuera necesario las advertencias por haber incumplido alguna de las recomendaciones proporcionadas.

6.9 Aplicación de cambios y transacciones sobre la base de datos

Siguiendo la idea del subapartado *Protección frente al cambio de los identificadores de los campos de formulario*, se ha creado un array asociativo con los identificadores de cada tabla de forma que se utilizarán en adelante las claves o identificadores textuales del array asociativo.

Los identificadores de las tablas afectadas en cada gestión realizada se guardan en el fichero *controlVersiones*. Cuando una operación o gestión implica la edición de una tabla se guarda en este fichero su identificador, pero si luego con otra operación o gestión se vuelve a modificar la misma tabla no es necesario volver a guardar en este fichero su identificador. Lo que se pretende es guardar los identificadores de las tablas sobre las que al menos se ha ejecutado una consulta SQL que ha supuesto la modificación de la tabla. Por otro lado, cada consulta ejecutada sobre la base de datos *hepapptemp* se guarda en un fichero *querysTransaccion.sql* al final del mismo. De esta forma se almacenan todas las consultas y en el orden en el que se han efectuado.

Para la gestión de estos ficheros que van a guardar las consultas realizadas y tablas afectadas, en los ficheros *IOficheros.php* y *registrarCambio.php* se implementan la funciones necesarias para poder comprobar que dichos ficheros existen en el sistema o crearlos en caso

contrario, leer las tablas ya registradas como modificadas, establecer el valor inicial de estos ficheros si fuera necesario, obtener el índice de una tabla a partir de su identificador, registrar una consulta SQL ejecutada y registrar una nueva tabla afectada o modificada.

El motivo de funcionar con dos bases de datos es que todas las operaciones y gestiones realizadas desde el panel de control se aplican sobre una base de datos de carácter más temporal (*hepapptemp*) y no se aplican sobre la base de datos principal (*hepapp*) que es de la que realmente la aplicación Android obtiene su contenido. Los cambios se aplican sobre la base de datos principal al realizar un *commit* que consiste en hacer que un conjunto de cambios provisionales se confirme de forma permanente. Estos cambios, combinados con el uso de las transacciones sobre de las bases de datos permitirán que el conjunto de cambios que las forman se apliquen de forma atómica, es decir, que o se realicen todos o ninguno y que, mientras se están realizando, no se pueden ejecutar otras operaciones sobre la base de datos. Las transacciones son bloqueantes mientras se llevan a cabo. Además, si algo sale mal o cualquiera de las operaciones que conforman la transacción falla se revierte todo al estado en el que se estaba antes de comenzar la transacción incluso aunque algunas de sus operaciones ya se hubieran llevado a cabo con éxito. Lo que se busca con todo ello es evitar situaciones como por ejemplo cuando se agrega un recurso capítulo al sistema y un usuario sincroniza el contenido de la aplicación Android justo antes de que el recurso se asocie a un módulo. Esto provocaría que el recurso sí se sincronizara en la aplicación, pero no se mostrara al no estar vinculado a un módulo. Dicho de otro modo, permite con calma que se realicen desde el panel Web todas las gestiones deseadas de forma que estas sean consistentes y se puedan aplicar de golpe evitando situaciones en las que un cliente (aplicación Android) se conecte y realice la sincronización en mitad de todas las gestiones que se están realizando desde el panel Web.

El fichero *ejecutarCommit.php* es aquel que se carga tras pulsar el control *commit* ubicado en la barra superior de navegación del panel Web. Este implementa el código y las funciones necesarias para poder ejecutar el *commit* y algunas tareas posteriores. Para realizar el *commit* se debe tener en cuenta que se debe seleccionar la base de datos *hepapp* con la función *mysqli_select_db()*. Si los ficheros *controlVersiones* y *querysTransaccion.sql* no existen o se encuentran en su estado inicial (vacíos) se considera que no hay ningún *commit* que hacer pues no hay cambios pendientes de aplicar de forma permanente. En caso de que se pueda efectuar el *commit*, se genera un identificador para este. El identificador se genera con los treinta y dos primeros caracteres del hash *SHA512* de la hora actual del sistema, es decir, *substr(hash('sha512', time()), 0, 32);*. Una vez generado el identificador se cargan todas las consultas registradas en *querysTransaccion.sql* en un array y después se genera y añade una consulta destinada a la tabla *controlversiones* por cada una de tablas modificadas que se indican en el fichero *controlVersiones* estableciendo como nueva versión el identificador del *commit*. También se establece este como versión de la base de datos (identificador 0).

Teniendo en un array todas las consultas SQL que van a formar la transacción, esta se inicia llamando a la función *mysqli_begin_transaction()*. A continuación, se ejecutan con un bucle *foreach* todas las transacciones del array y después se lanza la transacción llamando a la función *mysqli_commit()*. Si esta última retorna un valor que indica que se ha producido un error se deberá llamar a la función *mysqli_rollback()*. En caso contrario, se dará la transacción por finalizada y ejecutada con éxito, y se podrá hacer volver a los ficheros *controlVersiones* y

querysTransaccion.sql a su estado inicial. Finalmente se realiza una redirección a la pantalla de visión general del panel Web.

6.10 Servicio Web con API REST

Siguiendo la idea del subapartado *Protección frente al cambio de los identificadores de los campos de formulario*, se ha creado un array asociativo con los nombres de las tablas que se pueden obtener a través de la *API REST* donde para cada nombre de tabla se especifica otro array no asociativo que lista los campos de cada tabla. De esta forma se facilita notablemente la implementación del código que va a generar unos array que posteriormente se van a codificar en el formato *JSON*.

El fichero *validarPeticionAPI.php* implementa un módulo para la validación de las peticiones que se realizan sobre la *API REST*. La función *obtenerDatosPeticionAPI()* permite obtener los parámetros incluidos en la petición realizada a la API. Esta función comprobará que la petición se ha realizado haciendo uso del método *POST* de *HTTP*. Después obtendrá los parámetros obligatorios tipo de operación de lectura y el parámetro identificador del cliente, y el parámetro identificador de recurso que sólo será obligatorio en algún tipo de operación de lectura concreta. La función *validarIDcliente()* comprobará si el valor del identificador del cliente que accede a la API es *HepApp* para evitar dar servicio a otro cliente que no especifique un identificador o que el que especifique no se encuentre registrado en el servicio Web. En cualquier caso, esto no evitaría que un cliente pudiera hacerse pasar por el único cliente legítimo registrado y admitido que es la aplicación Android HepApp, pero, para ello, antes se debe descubrir este requerimiento de especificar un identificador, qué identificador especificar y cuál es el nombre del parámetro con el que se especifica. La función *obtenerTablaDatosLeer()* recibe como argumento el valor del parámetro que especifica el tipo de operación de lectura que se debe realizar. Así pues, cuenta con una estructura *switch* que, a partir del tipo de operación de lectura recibido, selecciona el nombre de la tabla que se va a leer y cuyo contenido se va a proporcionar a través de la API. El objetivo de esto es el de ocultar por seguridad los nombres de las tablas de la base de datos haciendo que el cliente utilice a cambio nombres de operaciones de lectura equivalentes. Por otro lado, también se permite acotar o limitar el número de operaciones que se pueden solicitar estableciendo un *case* por cada operación posible y estableciendo que el comportamiento por defecto para cualquier otro tipo de operación que se solicite sea proporcionar un mensaje de error y parar la ejecución. Por ejemplo, no existe una entrada *case* para reconocer un nombre o identificador de operación que se corresponda con leer los registros de la tabla *usuarios* de la base de datos *hepapp* dado que los datos de esta tabla no pueden estar disponibles desde fuera del sistema.

Por seguridad, a todos los mensajes de error que pueda mostrar la API, antes de mostrarlos, se les aplica una función hash segura de forma que se oculten los detalles a cualquier hipotético cliente malintencionado (se muestra el hash del mensaje de error para todos los clientes, incluidos los legítimos). Al igual que si se tratara de identificadores de errores, el desarrollador dispone de las correspondencias entre mensajes de error y sus hash (más que correspondencias tiene los mensajes de error y puede calcular su hash para compararlos) de forma que sólo este puede “revertir” la función hash.

El fichero *read.php* como tal es el que ofrece el servicio Web y proporciona la *API REST*. Primeramente incluye el fichero *core.php* y establece con la función *header()* algunas cabeceras HTTP como *Access-Control-Allow-Origin: ** y *Content-Type: application/json; charset=UTF-8*. Después incluye los ficheros *validarPeticionAPI.php* y el fichero de la conexión a la base de datos que usa al usuario *androidclient*.

A continuación llama a las funciones *obtenerDatosPeticionAPI()*, *validarIDcliente()* y *obtenerTablaDatosLeer()* para obtener correctamente todos los datos de la petición recibida en la API ya validados como el tipo de operación de lectura y el identificador del recurso si existiera.

En este fichero se implementan las funciones *leerTabla()* y *leerRegistro()* que generan y ejecutan la consulta sobre la base de datos que permite obtener respectivamente una tabla completa o un registro concreto de una tabla. Además, asociadas a estas, se implementan otras dos funciones de nombre *obtenerListaRegistrosTabla()* y *obtenerRegistroTabla()* que, con los resultados obtenidos de las funciones anteriores y haciendo uso del array asociativo definido al principio de este subapartado, permiten procesar la información obtenida de la base de datos para darla una estructura adecuada para la posterior codificación en formato *JSON*, y por lo tanto, poder proporcionar al cliente los datos solicitados a través de Internet. La primera función genera un array no asociativo de array asociativos donde cada uno de estos array es un registro de la tabla correspondiente. En el caso de la segunda función sólo se genera un array asociativo correspondiente a sólo un registro de la tabla que es el que específicamente se ha solicitado. Respecto a los array asociativos que se utilizan para almacenar cada uno de los registros de la tabla, se han utilizado por claves, para proporcionar la estructura, las definidas en el array asociativo del principio del subapartado.

Explicadas estas funciones, en *read.php* tras llamar a las funciones de *validarPeticionAPI.php*, en función del tipo de operación solicitado (una tabla completa o sólo un único registro) se llama a las funciones *leerTabla()* y *obtenerListaRegistrosTabla()* o a las funciones *leerRegistro()* y *obtenerRegistroTabla()*. Finalmente, con el array que se obtiene como resultado tras llamar a alguno de estos pares de funciones se aplica la codificación JSON y se imprime el resultado con *echo()*, es decir, *echo json_encode($informacionSolicitada);*.

A excepción de la tabla *usuarios* que no es accesible a través de la API, a la tabla *versionficherosrecursos* sólo se puede acceder para solicitar un único registro y al resto de tablas se puede acceder tanto solicitando un único registro como solicitando la tabla completa, siendo este último el caso más habitual.

6.11 Consideraciones sobre seguridad

En el desarrollo tanto del panel Web como del servicio Web se han aplicado algunas consideraciones básicas de seguridad vistas en los anteriores subapartados como la validación y saneamiento de los datos de entrada (escapar y codificar caracteres especiales, proteger de algún tipo de inyección SQL, validación de ficheros, etc…), validaciones lógicas, la ocultación de información sobre la estructura interna del sistema y la limitación de operaciones con un comportamiento por defecto de denegación (errores ocultos con hash y uso de array asociativos

junto con estructuras *foreach* y *switch*, etc…), autenticación simple de la API, definición de cuentas de usuario (se almacena el hash de las claves) y usuarios de la base de datos, la expiración y restricción de sesión, el uso del método *POST* de *HTTP*, y el uso de transacciones y *commit* sobre la base de datos (seguridad en el manejo de la información).

En este subapartado se profundiza algo más y se proporcionan detalles sobre algunos de estos mecanismos y se describen otros no mencionados hasta el momento.

Como se ha dicho en el subapartado *Gestión de las sesiones*, en el fichero *loginRequired.php* se incluye al fichero *sessionExpirationHandler.php* para hacer uso de la única función que este implementa y que se llama *comprobarExpiracionSession()*.

Esta función primero comprueba el estado de inactividad para saber si debe cerrar automáticamente la sesión (expiración automática). Para ello, cuando se llama a esta función por primera vez se establece en el array asociativo *$_SESSION* la hora actual del sistema con la función *time()*. En las sucesivas veces que se llama a la función, se toma de nuevo con *time()* la hora actual del sistema y se compara con el sello temporal almacenado en *$_SESSION* para comprobar si ha vencido el temporizador de inactividad si la comparación es mayor que el valor fijado para el temporizador en el fichero *core.php* que toma un valor de treinta minutos expresado en segundos. Si el temporizador ha vencido la sesión se destruye y si el temporizador no ha vencido se actualiza el sello temporal almacenado en *$_SESSION* a la nueva hora actual del sistema. Es importante destacar que la inactividad se comprueba cada vez que es llamada la función *comprobarExpiracionSession()*, lo cual sólo ocurre al cambiar de pantalla del panel Web. Esto se ha implementado así porque lo que se quiere detectar es la inactividad en el uso o ejecución de una consulta SQL sobre la base de datos, ya que realizar cualquier gestión, cambiar de pantalla del panel Web o actualizar la pantalla actual implicará la ejecución de alguna consulta sobre la base de datos. Esta medida pretende reducir el riesgo de que se produzca un robo de sesión basado en el robo de *cookies* de sesión.

Por otro lado, esta función también implementa otro mecanismo basado en temporizador para proporcionar protección frente al ataque *session fixation*. Para ello, se hace uso del sello temporal que se establece en *login.php* al iniciar sesión también en el array asociativo *$_SESSION*. Entonces, se realiza la misma comparación que en el caso anterior. Compara el sello temporal almacenado con el resultado de llamar a *time()* y se comprueba que si se supera el valor fijado del temporizador que es también de treinta minutos. La diferencia es que en este caso este sello temporal no se actualiza hasta que se produce la expiración del temporizador por lo que siempre se espera el temporizador completo hasta que este vence. Cuando vence, la acción que se lleva a cabo es la de llamar a la función *session_regenerate_id()* para forzar a que se utilice un nuevo identificador de sesión invalidando el anterior.

No se debe confundir este con el primer temporizador, porque se trata de dos temporizadores distintos a pesar de tener la misma duración (se están utilizando y almacenando en *$_SESSION* dos sellos temporales distintos). Así pues, se puede dar la situación de no producirse inactividad durante más de una hora por un uso constante del panel Web, lo cual no produciría una expiración automática de la sesión, pero si producirá al menos en dos ocasiones que el identificador de la sesión se regenere. De los dos mecanismos explicados, sólo el segundo es totalmente transparente para el usuario.

Por otro lado, como ya se ha mencionado, se utiliza la función *error_reporting(0);* en *core.php* para evitar que en caso de producirse algún error o que se pueda mostrar alguna advertencia al usuario tanto en el panel Web como en el servicio Web con la *API REST*, estos mensajes no se lleguen a mostrar para evitar revelar información sobre el funcionamiento interno del sistema. El desarrollador siempre puede en su entorno de pruebas deshabilitar esta restricción y tratar de reproducir el error para estudiarlo. En combinación con esta medida se utiliza el símbolo @ precediendo algunas líneas de código como las que abren la conexión con la base de datos y algunas llamadas a *include_once()*.

Además, se configura el servidor Web Apache con los permisos adecuados sobre el sistema de ficheros para que no permita mostrar y listar el contenido de los diferentes directorios del sitio Web. Por ejemplo, en un sistema *UNIX*, usando las listas *ACL* y el sistema de permisos tradicional *rwx*, se podría especificar para el usuario *apache* (que ejecuta el servidor Web) que sólo se permite el permiso *x* (acceso) para los directorios y sólo se permite el permiso *r* (lectura) para los ficheros. Este es el escenario más restrictivo y en este caso afectaría al funcionamiento normal del sistema pues algunos ficheros requieren llevar a cabo escrituras de ficheros en otros directorios. Se podrían utilizar entonces como complemento los sistemas *MAC* como *SELinux*. Para este proyecto, simplemente se ha retirado el permiso de lectura de los directorios, lo cual ya permite evitar que su contenido pueda ser listado sin impedir que pueda ser accedido en caso de conocer de antemano las rutas y su contenido. La forma más cómoda y que al mismo tiempo funcionaría en cualquiera que sea el sistema operativo (con un sistema de ficheros concreto asociado) que esté ejecutando el servidor Web *Apache*, sería utilizar el fichero de configuración del servidor Web *Apache* empleando la directiva *Options -Indexes*. Otra opción sería, dado que habitualmente la directiva *AllowOverride* lo permite, el uso de un fichero *.htaccess* con el contenido *Options -Indexes* en cada directorio en el que se quiera aplicar esta restricción.

Como ya se ha introducido, el uso de la función hash para que en lugar de guardar la clave en texto claro se almacene en la base de datos su hash permite que se realice correctamente la autenticación del usuario a la vez que, en caso de filtración del contenido de la tabla que almacena las cuentas de usuario, no se revelen las claves de usuario ya que no se puede revertir la función hash para recuperar la calve si esta función es segura. Si, por ejemplo, un usuario utilizara la misma clave en dos servicios Web, en caso de que uno de los servicios Web sucumba a un ataque teóricamente no proporcionaría al atacante la calve de usuario necesaria para acceder al otro servicio Web suplantando al usuario cuya clave se pretendía robar. A pesar de ello, siempre se proporciona al usuario la recomendación de no reutilizar claves en sus cuentas de usuario. En cualquier caso, puesto que no supone ninguna desventaja relevante siempre será mejor almacenar el hash de la clave en lugar de almacenar esta directamente en texto claro.

Con intención de proteger el acceso a la base de datos se han creado varios usuarios para intentar limitar los permisos sobre la base de datos al máximo en cada caso permitiendo realizar sólo las operaciones que de verdad se necesitan utilizar en cada situación. Para ello, se define el usuario *webpannel* para ser empleado sólo por el panel Web con permisos que restringen que sólo se puedan ejecutar las consultas SQL *SELECT*, *INSERT*, *UPDATE* y *DELETE*

sobre ambas bases de datos. En el caso del usuario *androidclient*, que será el usuario utilizado por el servicio Web con la *API REST*, sólo podrá ejecutar la consulta SQL *SELECT* y sólo sobre la base de datos *hepapp*. Finalmente, el usuario *admin* cuenta con mayores privilegios para permitir a las utilidades como *mysql* y *mysqldump* ejecutar las operaciones necesarias para poder realizar y restaurar copias de seguridad completas de las bases de datos.

Respecto a las copias de seguridad, este se puede considerar como un mecanismo más de seguridad al aportar la posibilidad de obtener redundancia de los datos almacenados en la base de datos, así como de exportar estas a un fichero. Es más, el uso de la base de datos *hepapptemp* como efecto colateral estaría proporcionando redundancia de la mayoría de las tablas (y las más importantes) de la base de datos *hepapp*.

El fichero *copiaSeguridadBaseDatos.php* es el encargado de procesar el formulario de la pantalla de la gestión de las copias de seguridad de la base de datos. Tras incluir al fichero *core.php* y otros varios que necesita que ya se han visto en subapartados anteriores, llama a la función *obtenerDatosFormulario()* para obtener los datos recibidos del formulario y después llama a sus propias funciones de validación implementadas apoyándose en las funciones del módulo *validarLimpiarDatos.php*. Seguido, llama a una función propia que permite obtener los nombres de los ficheros de las copias de seguridad completas de la base de datos que se hayan podido realizar anteriormente. Se debe recordar la restricción de que sólo se pueden almacenar como máximo hasta dos copias de seguridad completas de las bases de datos. Posteriormente, con una estructura *switch* deduce si se ha pedido la operación de realizar una copia de seguridad o la operación de restaurar una copia de seguridad. En este segundo caso, además, con otra estructura *switch* anidada permite deducir qué copia concreta se desea restaurar. El fichero implementa una función para cada uno de los tres casos de uso que pueden ejecutarse desde las estructuras *switch* definidas. Como en otros casos, después de la estructura *switch* se realiza una redirección a la pantalla de visión general del panel Web.

En el caso de crear la copia de seguridad, la función a la que se llama primero comprueba si ya existen dos ficheros de copias de seguridad para en caso afirmativo borrar el más antiguo haciendo uso de la función *unlink()* de PHP. Después, se hace uso de la función *exec()* de PHP para mediante la utilidad *mysqldump* realizar la copia de seguridad de sólo la base de datos *hepapp*. A la utilidad *mysqldump* se le especifica como argumentos o parámetros la cuenta de usuario que debe utilizar (*admin*), sus credenciales, y la opción *--single-transaction*. El nombre del fichero de la copia de seguridad con extensión *.sql* tendrá por nombre el sello temporal de la hora actual del sistema que se obtiene ejecutando la función *time()* de PHP. Esto permitirá mostrar en la pantalla del panel Web la fecha de cada copia de seguridad a partir de su nombre. Además, quedarían correctamente ordenados los ficheros de copias de seguridad mostrándose tanto en orden alfabético como en función de la fecha de última edición. Es importante destacar que sólo se realiza una copia de seguridad de la base de datos *hepapp*, lo cual implica que en la operación de restauración se restaura la base de datos *hepapptemp* a partir de la base de datos *hepapp*. Esto implica que al hacer la copia de seguridad se podrían perder todas las gestiones hechas sobre *hepapptemp* aún no aplicadas con un *commit* sobre la base de datos *hepapp*.

Para restaurar una copia de seguridad, teniendo ya seleccionado el fichero de la copia de seguridad que se va a restaurar, haciendo de nuevo uso de la función *exec()* y con la utilidad *mysql* se restauran las bases de datos *hepapptemp* y *hepapp*. Como la base de datos *hepapptem* ahora en este momento tiene las mismas tablas y contenido que la base de datos *hepapp* se

ejecutan algunas consultas SQL *DROP* sobre *hepapptemp* para eliminar algunas tablas como *controlversiones* y *usuarios*. Finalmente, se restaura el contenido inicial de los ficheros *controlVersiones* y *querysTransaccion.sql* para eliminar las gestiones registradas en estos ficheros de un posible *commit* anterior que no llegara a haberse ejecutado (y que se perdería). Aunque no es el uso previsto para esta funcionalidad de copias de seguridad, se podrían utilizar la opción de copia de seguridad seguida de la de restauración para revertir los cambios y gestiones realizadas desde el panel Web de un *commit* que aún no se haya ejecutado o aplicado permanentemente sobre la base de datos.

Como se ha podido comprobar, al ofrecer un servicio Web disponible públicamente se deben aumentar las consideraciones en materia de seguridad. Además, en este caso, si se quiere atacar al sistema desarrollado es de esperar que el atacante pretenda manipular o acceder al servicio Web en lugar de a la aplicación Android pues este centraliza y controla todo el contenido y funcionamiento de la aplicación Android.

6.12 Uso de fuentes en lugar de imágenes.

Como ya se ha comentado, se han utilizado varios iconos por todo el panel Web para facilitar el reconocimiento de forma fácil y visual de los distintos elementos y la información que en este se presenta. Para optimizar el rendimiento y reducir la cantidad de información que debe transmitir el servidor Web en lugar de imágenes para los iconos se han generado y utilizado fuentes propias de iconos que, junto a algunas clases *CSS*, permite que estos se utilicen como si se tratara de cualquier otro carácter. Algunas de las ventajas que esto presenta es la reducción del número de peticiones HTTP porque sólo se necesita solicitar un único fichero de muy poco tamaño para obtener y mostrar todos los iconos, la compatibilidad con todos los navegadores, al tratarse de vectores se adaptan a las diferentes densidades y resoluciones de pantalla, como cualquier texto formado por caracteres se puede cambiar su tamaño sin necesidad de pasar por un editor de imágenes, y también soportan cualquiera de los efectos y animaciones de *CSS3*. En definitiva, es una recomendable y muy buena práctica dada su alta eficiencia especialmente desde el punto de vista del rendimiento.

6.13 Relleno automático de formularios.

Como se ha explicado en el subapartado *Imprimir tablas de datos*, utilizando la base de datos y PHP al imprimir las tablas se puede habilitar que se proporcione a cada fila de las tablas que se muestran qué función concreta y con qué valores de sus argumentos se debe llamar para auto rellenar los campos de los formularios con los valores de cada fila. Las funciones *JavaScript* cuyos puntos de llamada se establecen con PHP al imprimir las tablas se implementan en el fichero *gestiones.js*. En este fichero se implementan funciones generales que se apoyan y usan la biblioteca *JQuery* para vaciar y rellenar los campos de los formularios con los valores que reciben como argumentos. Sobre estas funciones o apoyándose en estas funciones más generales, se crean otras más específicas particularizadas a los distintos formularios del panel Web. Se implementan también en este fichero las funciones que muestran los diálogos extra

que aparecen para volver a confirmar que se desea realizar una operación o gestión cuando esta es de carácter un poco más crítico ya que puede conllevar a una pérdida accidental de información. Estas son llamadas con el atributo *onsubmit* de la etiqueta *form* de *HTML* en los formularios correspondientes del panel Web.

6.14 Otros.

Como ya se ha mencionado no se ha implementado una validación *frontend* de los formularios con el atributo *onsubmit* de la etiqueta *form* de *HTML* y funciones *JavaScript*. La única validación *frontend* presente es la propia de *HTML 5* estableciendo correctamente el valor del atributo *type* de las etiquetas *input* de *HTML*. Es el caso de los campos para introducir un identificador y una dirección URL. Como ya se ha explicado detalladamente ya se cuenta con una exhausta validación en la parte *backend* donde siempre es obligatoria realizarla dado que el *frontend* no se puede controlar y la validación *frontend* no es fiable ya que una vez enviado al cliente este puede modificarlo para desactivar los mecanismos de validación. La validación *frontend* suele ser sin embargo recomendable debido a que puede suponer una gran ayuda a los usuarios que hagan un uso legítimo del sistema pues esta validación se realizaría antes de mandar y procesar los datos del formulario por el *backend* lo cual, de ocurrir, podría mermar el rendimiento y percepción de uso del sistema por parte del usuario. La validación proporcionada por *HTML 5* se considera suficiente como validación *frontend* sin que se necesite establecer otra a mayores.

Debido al empleo de los *framework*, bibliotecas y versiones concretas de las tecnologías Web que se han utilizado para el panel Web, el *frontend* es *responsive* y por lo tanto su visualización es correcta en diferentes anchos de pantalla. Además, como ya se ha dicho, para la facilitar el reconocimiento de las filas de cada tabla se ha establecido el estilo cebra en estas.

Respecto a los documentos *HTML*, estos también se encuentran ordenados y estructurados al haber separado las partes comunes en ficheros distintos y al usar después las funciones *include()* y *include_once()* de PHP para incluirlos donde se necesitara. Por ejemplo, la barra de navegación presente en todas las pantallas del panel Web se encuentra en un fichero que es incluido en un documento *HTML* cada vez que se desea mostrarla. Los diferentes formularios de las pantallas del panel Web también se encuentran en varios ficheros que son posteriormente incluidos en el documento *HTML* de la correspondiente pantalla. Se logra de esta forma una cierta modularidad. Esta práctica permite ahorrar código *HTML* al evitar tener que replicarlo en varias ocasiones y también que cuando se debe cambiar algo sólo sea necesario hacer el cambio una única vez en un sólo punto.

Capítulo 7: Manual de usuario de HepApp

En el presente capítulo se va a explicar desde un punto de vista de manual de uso la interfaz gráfica y las posibilidades de interacción que esta ofrece al usuario. Se utilizan figuras con capturas de pantalla, tomadas tanto de dispositivos emulados como de dispositivos físicos, para acompañar a las explicaciones textuales. Sobre el contenido y su organización, en HepApp, esta es la misma que la utilizada por el panel Web que se describe en el subapartado *Sobre la organización del contenido*. El dispositivo físico utilizado es un Smartphone *BQ Aquaris U* con Android *Nougat* versión 7.1.1 y el dispositivo emulado es una Tablet *Nexus 7* con Android *Oreo* versión 8.1. Todo el contenido de la aplicación está disponible para cualquier usuario a excepción de algunos sitios Web de la Universidad de Calgary que se restringen a sus alumnos.

7.1 Pantalla inicial

En la figura **¡Error! No se encuentra el origen de la referencia.** se muestra la pantalla inicial que se carga al abrir la aplicación. En la cabecera de la pantalla, debajo del título se muestra el logotipo de la aplicación que también aparece en el cajón de aplicaciones del *launcher* del dispositivo Android. A través de los botones que presenta se pueden acceder a las diferentes secciones de la aplicación.

En la figura **¡Error! No se encuentra el origen de la referencia.** se muestra la animación de sincronización del contenido de la aplicación con la maquina servidora a través de Internet. La tarea de sincronización se puede cancelar pulsando el botón atrás que ofrece el dispositivo.

7.2 Sección Capítulos

Con la figura **¡Error! No se encuentra el origen de la referencia.** se muestra el listado de módulos que cuentan con recursos de tipo capítulo registrados. Para acceder a los recursos de un módulo vale con pinchar el módulo deseado.

Las funciones de los elementos numerados en la figura se explican a continuación:

1) Permite mostrar la barra lateral de navegación que se describe más adelante.
2) Permite retroceder a la pantalla anterior.
3) Su utilidad es la de volver a la pantalla inicial de la aplicación.
4) Es el título propio de cada pantalla. También permite conocer la posición de lo que se muestra dentro de la organización de contenidos que sigue HepApp.

Se debe destacar que no se listan aquellos módulos sin recursos de tipo capítulos registrados. En la figura **¡Error! No se encuentra el origen de la referencia.** se muestra el listado de recursos de tipo capítulos del módulo 1. Se ha rodeado el control que permite avanzar a la siguiente página del listado.

La figura **¡Error! No se encuentra el origen de la referencia.** muestra el visualizador de PDF integrado en HepApp con el recurso *Capítulo 3* cargado.

De nuevo, se marcan y numeran partes cuyas funciones son:

1) La notificación que el *Gestor de Descargas* de Android ofrece para mostrar que la descarga del recurso se ha completado.
2) Es el control que permite eliminar el recurso descargado del sistema.
3) Son los controles que permiten cambiar el recurso a visualizar a otro contiguo de los de la lista desde la que se ha abierto el visualizador. Estos controles evitan tener que retroceder al listado para cambiar de recurso.
4) Permite acceder al listado de recursos del mismo módulo, pero de la sección Podcasts.
5) En la esquina derecha superior, en la barra superior, se aprecia con un icono de una cámara el control que permite tomar capturas de pantalla de la aplicación.

Inicialmente, el recurso no existe en el sistema y no se puede cargar hasta que se descarga. Por ello, aparecerá el visualizador vacío y en la barra inferior, a la izquierda, junto al icono de la papelera, aparece el control de descarga que tiene el aspecto de la figura **¡Error! No se encuentra el origen de la referencia.**. Si el recurso ya se descargó, pero tiene disponible una actualización, en la misma ubicación que el control de descarga, aparecerá el control de actualización con la apariencia de la figura **¡Error! No se encuentra el origen de la referencia.**. Cuando el recurso se encuentra descargado en el sistema y no dispone de ninguna actualización no se mostrarán ninguno de los dos controles.

Es importante mencionar que el título y descripción de un recurso se actualiza de forma automática cuando se realiza la sincronización del contenido con la máquina servidora al iniciar la aplicación. Lo que realmente permiten gestionar estos controles para descargar, actualizar y borrar es el fichero asociado al recurso. A este grupo de controles se les llamará en adelante *controles de gestión del recurso*.

7.3 Sección Podcasts

Como se muestra en las figuras **¡Error! No se encuentra el origen de la referencia.** a **¡Error! No se encuentra el origen de la referencia.**, el funcionamiento de esta sección es análogo al descrito para la sección anterior. La única particularidad es que el visualizador de recursos en esta ocasión no muestra nunca los controles de gestión del recurso porque el tipo de recurso de esta sección no se puede descargar.

Se debe observar que, de nuevo, sólo se muestra el listado de los módulos que cuentan con recursos de tipo podcast registrados.

7.4 Sección Cards

En esta sección, como se muestra en la figura **¡Error! No se encuentra el origen de la referencia.**, simplemente se muestra un sitio Web de la Universidad de Calgary con acceso restringido a sus alumnos.

7.5 Sección Figuras

En esta sección se muestran las imágenes con las que cuenta la aplicación. Estas se organizan en las subcategorías o subsecciones que aparecen listadas en la figura **¡Error! No se encuentra el origen de la referencia.**. En esta sección, como se ve más adelante, al listar las imágenes de una subsección sólo se muestra una miniatura de la imagen cuando esta ya existe en el sistema porque se ha descargado previamente. Si el recurso imagen no existe aún en el sistema, en su lugar, se mostrará por defecto el icono de la sección.

7.5.1 Subsecciones Tabla de contenidos y Esquemas

En estas dos categorías, como se puede ver en las figuras **¡Error! No se encuentra el origen de la referencia.** y **¡Error! No se encuentra el origen de la referencia.**, se muestran respectivamente imágenes de tablas de contenidos y esquemas.

Como se puede observar en la figura **¡Error! No se encuentra el origen de la referencia.**, para cargar los recursos imagen de estas subsecciones se utiliza un visualizador específico con los tipos de interacciones disponibles deshabilitados.

En la sección Figuras, el visualizador de recursos específicos que se usa se encarga de realizar la descarga del recurso que se desea visualizar de forma automática si este no se encuentra ya descargado en el sistema. El control para eliminar un recurso imagen no se encuentra disponible. Cuando el recurso dispone de una actualización, esta no se realiza de forma automática y aparece en la barra inferior el control de actualización antes visto.

7.5.2 Subsección Figuras interactivas

En esta subsección se presentan los recursos compuestos que están formados por un conjunto de hasta un máximo de cuatro recursos simples. Se llama recurso compuesto a este conjunto de recursos simples que cuenta con su propio título y descripción. En el listado de los recursos compuestos de la figura **¡Error! No se encuentra el origen de la referencia.**, la imagen del recurso simple que se muestra es la del recurso simple representativo del recurso compuesto. Este será también el que se cargue por defecto al abrir el recurso compuesto en el visualizador.

Como se puede comprobar en la figura **¡Error! No se encuentra el origen de la referencia.**, el visualizador encargado de mostrar los recursos compuestos activa una

interacción que permite cambiar (*switch*) entre los recursos simples asociados al recurso compuesto que se está visualizando. En los controles excluyentes ubicados en la parte central inferior, se muestran los títulos de cada recurso simple. Se muestran tantos controles excluyentes como recursos simples tiene asociados el recurso compuesto.

7.5.3 Subsección Figuras de capítulos

La única particularidad de esta subsección respecto a las subsecciones Tabla de contenidos y Esquemas es que presenta antes el listado de módulos con recursos de tipo imagen registrados. Así pues, al igual que ocurre con las secciones Capítulos y Podcasts, al acceder a un módulo se van a listar los recursos imagen, y desde este se podrán cargar los recursos en el visualizador con las interacciones deshabilitadas. El objetivo de esta subsección es el de presentar los recursos imagen organizados de forma coherente a como aparecen en los recursos capítulos de la sección Capítulos. En las figuras **¡Error! No se encuentra el origen de la referencia.** y **¡Error! No se encuentra el origen de la referencia.** se puede ver en la primera el listado de módulos y en la segunda el listado de recursos imagen asociados con el módulo que se haya seleccionado.

Como ya se ha explicado antes, en el listado de módulos sólo se mostrarán aquellos que tienen recursos imagen registrados.

7.5.4 Subsección Pintando

Como en otras, al abrir esta subsección se muestra un listado de los recursos imagen disponibles. Al seleccionar uno, como se muestra en la figura **¡Error! No se encuentra el origen de la referencia.**, se abre el visualizador de imágenes con la interacción que permite pintar sobre la imagen habilitada.

En la parte superior derecha (rodeados) se muestran dos controles propios de este tipo de interacción que permiten seleccionar otro color o borrar todos los trazos hechos sobre la imagen. Al pulsar el control para cambiar de color se muestra el diálogo que se muestra en la figura **¡Error! No se encuentra el origen de la referencia.**. Se observa también, que puesto que no hay más recursos registrados en esta subsección el visualizador no muestra los controles de la barra inferior que permiten cambiar a un recurso contiguo.

Se pueden guardar los trazos hechos sobre la imagen realizando una captura de pantalla con el control de la cámara en la barra superior.

7.6 Sección Calculadoras

Se trata de la única sección cuyo contenido es fijo y no se sincroniza con la máquina remota. El contenido de esta sección, de necesitarse se actualizaría a través de actualizaciones

de la aplicación. En la figura **¡Error! No se encuentra el origen de la referencia.** se listan las diferentes calculadoras disponibles en la aplicación. La primera es la calculadora completa que permite obtener el resultado de todos los algoritmos explicados en el capítulo *Algoritmos de hepatología*. Las demás calculadoras son calculadoras parciales. En la figura **¡Error! No se encuentra el origen de la referencia.** se puede ver la barra lateral de navegación desplegada. Con esta barra se puede acceder en cualquier momento al listado de recursos de un módulo concreto para las secciones Capítulos y Podcasts o a cualquiera de las otras secciones sin tener que pasar por la pantalla inicial.

7.6.1 Calculadora completa

En referencia a la calculadora completa, como se puede observar en las figuras **¡Error! No se encuentra el origen de la referencia.** a **¡Error! No se encuentra el origen de la referencia.**, esta está formada por tres pantallas con formularios para introducir los valores médicos de todos los parámetros necesarios para realizar el máximo número de cálculos posibles.

A continuación, se explican los elementos marcados con números en la figura **¡Error! No se encuentra el origen de la referencia.**:

1) Es el control que permite borrar todos los valores introducidos en la pantalla.
2) Este control permite restablecer sólo en la pantalla que se visualiza los valores introducidos para un cálculo anterior.
3) Muestra la capa oculta con información detallada sobre la pantalla.
4) Permite acceder a la siguiente pantalla de la calculadora.
5) Muestra la capa oculta que permite establecer los ajustes de la calculadora.

En la figura **¡Error! No se encuentra el origen de la referencia.** se observa la capa oculta de los ajustes de la calculadora. Si se cierra esta capa sin dar al control que permite guardar los ajustes introducidos, estos se pierden. Comparando las figuras **¡Error! No se encuentra el origen de la referencia.** y **¡Error! No se encuentra el origen de la referencia.** se comprueba el efecto de activar y desactivar las unidades internacionales.

La figura **¡Error! No se encuentra el origen de la referencia.** muestra la cuarta pantalla de la calculadora completa en la que se calculan y presentan los resultados de todos los algoritmos médicos que se han podido calcular a partir de los valores introducidos en pantallas anteriores. El motivo por el que algunos algoritmos pueden no presentar un resultado (carácter "-") es que se necesita algún valor no introducido en las pantallas anteriores o que los cálculos se han parado al encontrar que un valor introducido no es válido. Además, si con los valores introducidos se ha podido calcular el algoritmo *Alberta* se mostrarán hasta dos sugerencias de posibles tratamientos si estuvieran disponibles.

De nuevo, se explican los elementos marcados con números:

1) Tablas con los resultados de los diferentes algoritmos.
2) Sugerencias de posibles tratamientos.
3) Permite retroceder a la primera pantalla de formulario de la calculadora completa. De cara al control de restaurar los valores anteriores, al mostrarse la pantalla de resultados, se considera que se han realizado los cálculos.
4) Permite retroceder a la segunda pantalla de formulario de la calculadora completa.
5) Permite retroceder a la tercera pantalla de formulario de la calculadora completa.
6) Permite mostrar la pantalla que recopila y presenta todos los valores introducidos a lo largo de todas las pantallas de formulario de la calculadora. En la figura **¡Error! No se encuentra el origen de la referencia.** se muestra la pantalla mencionada.
7) Sólo si se ha podido calcular el algoritmo *Alberta*, se mostrará la capa oculta de más información que se refiere a los resultados obtenidos. En la figura **¡Error! No se encuentra el origen de la referencia.** se muestra la capa oculta más información sobre la pantalla de resultados.
8) Sólo si se ha podido calcular el algoritmo *Alberta*, se mostrará la pantalla con el esquema correspondiente a los resultados obtenidos. En la figura Figura 69 se muestra la pantalla con el esquema.

Se debe destacar que desde las pantallas de resumen de valores y del esquema de Alberta, pulsando el botón atrás, se regresa a la pantalla de resultados, pero desde esta o desde cualquiera de las pantallas de formulario el comportamiento del botón atrás es el de salir de la calculadora completa. Esto es así porque desde la pantalla de resultados, si se retrocede a alguna de las pantallas de formulario se entiende que es porque se quiere cambiar alguno de los valores introducidos y actualizar después los resultados consecuentemente (implica un nuevo cálculo de todos los algoritmos) porque si lo que se quiere es consultar los valores introducidos, ya se pueden consultar todos en la pantalla de resumen de valores.

Otro detalle importante es que, a diferencia de las calculadoras parciales que cuentan con un botón exclusivamente para solicitar el cálculo de los resultados, la calculadora completa realiza los cálculos de forma automática al pasar de la última pantalla de formulario a la pantalla de resultados de forma que cuando esta última se carga ya muestra los resultados obtenidos.

7.6.2 Calculadora parcial CPS

Es una de las calculadoras parciales que sólo permite obtener el resultado del algoritmo *Child Pugh Score* (CPS). En la figura **¡Error! No se encuentra el origen de la referencia.** se muestra la calculadora parcial y en la figura **¡Error! No se encuentra el origen de la referencia.** la capa de más información para esta calculadora.

Como ya se había mencionado, las calculadoras parciales contarán con un botón exclusivo para solicitar el cálculo de los resultados a partir de los valores introducidos. Estas calculadoras sólo cuentan con una pantalla en la que se muestran tanto el formulario para introducir los valores como el resultado obtenido en un recuadro en la parte superior derecha. El funcionamiento de los controles de la barra inferior de la calculadora es el mismo que el descrito en la calculadora completa. En la figura **¡Error! No se encuentra**

el origen de la referencia. se pueden ver los ajustes que serán comunes en todas las calculadoras parciales.

7.6.3 Calculadora parcial MELD

Es una de las calculadoras parciales que permite calcular el resultado de los algoritmos *Model For End-Stage Liver Disease* (MELD), MELDNa y 5vMELD. En la figura **¡Error! No se encuentra el origen de la referencia.** se puede observar la pantalla de la calculadora.

7.6.4 Calculadora parcial Okuda

Es una de las calculadoras parciales que sólo permite calcular el resultado del algoritmo *Okuda*. En la figura **¡Error! No se encuentra el origen de la referencia.** se puede observar la pantalla de la calculadora.

7.6.5 Calculadora parcial CLIP

Es una de las calculadoras parciales que sólo permite calcular el resultado del algoritmo *Cancer of the Liver Italian Program* (CLIP). En la figura **¡Error! No se encuentra el origen de la referencia.** se puede observar la pantalla de la calculadora.

7.7 Sección Recursos

Como se observa en la figura **¡Error! No se encuentra el origen de la referencia.**, al entrar en esta sección se listan todos los recursos disponibles registrados en esta sección.

Al seleccionar cualquiera de ellos, tal y como se puede ver en la figura **¡Error! No se encuentra el origen de la referencia.**, se abre el visualizador Web para mostrar el sitio Web al que se refiere el recurso seleccionado.

7.8 Sección PubMed

En esta sección, como se muestra en la figura **¡Error! No se encuentra el origen de la referencia.**, simplemente se muestra el sitio Web *PubMed*.

7.9 Sección Información

En esta sección, como se muestra en la figura **¡Error! No se encuentra el origen de la referencia.**, simplemente se muestra más información acerca de este proyecto.

Capítulo 8: Descripción técnica y sobre el desarrollo de HepApp

La aplicación HepApp es una aplicación desarrollada de forma nativa para el sistema operativo Android. Se trata de una aplicación originalmente ideada por el Dr. Kelly Burak con la intención de proporcionar una nueva y eficaz herramienta tanto para la docencia sobre hepatología que imparte en la Universidad de Calgary en Canadá como para el apoyo al diagnóstico de pacientes en relación a la hepatología. La idea ha dado lugar a un proyecto colaborativo de investigación entre dicha universidad y la Universidad de Valladolid en España. El proyecto lleva abierto desde 2011. En el momento del inicio del desarrollo de este trabajo se contaba con una implementación para iOS creada con *PencilCase* y una implementación para Android basada en esta anterior. En el desarrollo de HepApp se han tenido en cuenta las implementaciones existentes de partida para mantener en la medida de lo posible el aspecto de la aplicación o, dicho de otra forma, la mayor coherencia posible con la interfaz gráfica de las otras implementaciones. En paralelo a este trabajo, se ha realizado el desarrollo de otra implementación de la aplicación para iOS, pero esta vez de forma nativa. Aunque teniendo como referencia los proyectos anteriores, debido a la naturaleza de lo que se ha desarrollado se decidió comenzar desde cero.

Sin entrar mucho en detalle, para el diseño y desarrollo de HepApp se ha intentado cumplir con lo dispuesto en el patrón estructural conocido como *Modelo-Vista-Controlador* (patrón *MVC*) (Limaye et al, 2018). Este patrón pretende dar la base para diseñar una arquitectura que separa la información, el tratamiento que se hace sobre esta y su presentación al cliente (usuario final u otro software) de forma que queda dividido en capas o subsistemas que pueden ser fácilmente extensibles. El patrón busca que al aplicarse el software resultante cuente con características como mantenibilidad, tolerancia a fallos, escalabilidad, portabilidad y entre otros, correcta presentación al cliente. Se trata de un patrón ampliamente conocido y básico en el sentido de que actualmente se da por hecho que su aplicación sobre todo desarrollo software es imprescindible.

Sin embargo, como ocurre con los patrones de diseño software, se trata de recomendaciones y buenas prácticas a tener altamente en cuenta y, sobre todo, dentro de que sean más o menos específicos son generalistas e interpretables, es decir, que el hecho de combinarse y dependiendo de las necesidades y características de cada proyecto hacen que más que aplicar los patrones correctos sea más importante saber en cada caso cuándo de forma justificada se puede saltar lo que estos indican para ajustarse mejor a lo que se pretende hacer. Entonces se puede dar una situación en la que implementar una parte del proyecto puede resultar excesivamente ardua y compleja siguiendo estrictamente el patrón *MVC* frente a otra alternativa que, si bien se desvía algo de la guía, resulta sumamente más simple de aplicar. Por ello, aunque se vea en detalle más adelante, al tratarse de un desarrollo relativamente complejo en ocasiones no se han seguido tan estrictamente las indicaciones de los patrones de diseño empleados y se han realizado variaciones o interpretaciones convenientes de estos.

El proyecto se ha estructurado en las siguientes partes, capas, subsistemas y/o paquetes software que se presentan a continuación.

8.1 Vista

La vista es la interfaz que se ofrece al cliente del sistema para que interactúe con él. Como ya se ha mencionado, el cliente podría ser otro software, pero en este caso el cliente es el usuario final por lo que se trata de una interfaz gráfica. Presenta el modelo al cliente. Tanto la vista como el controlador atienden peticiones que el cliente hace sobre el sistema. La diferencia está en que la vista atenderá sólo las peticiones que cambian la forma en la que la información del modelo es presentada mientras que el controlador atiende las peticiones que cambian o modifican el modelo utilizando la interfaz proporcionada por el modelo.

Como se ha indicado, ya se parte con el diseño visual de una interfaz gráfica hecho que simplemente se debe mantener todo lo más fielmente posible. Se debe tener en cuenta que se pueden realizar implementaciones que visualmente para el usuario final tengan exactamente la misma apariencia, pero no tengan nada que ver unas con otras. Lo anterior se refiere a que se ha optado por intentar realizar la implementación de la vista de forma lógica, estructurada y evitando la repetición del código gracias al uso de mecanismos del paradigma de orientación a objetos de *Java* y la gran flexibilidad que aporta el sistema de ficheros de recurso disponible en Android. A nivel efectivo, aunque se explica más delante de forma más detallada, esto ha permitido jerarquizar de alguna forma la implementación mediante el uso de herencia en *Java* y las posibilidades de *include*, *merge* e *inflate* con ficheros de recursos XML para los *layout*.

Entonces, la vista, a su vez, se ha organizado como sigue según las partes que deben ser estáticas o dinámicas.

8.1.1 Actividad común

Se trata de una clase abstracta con el principal objetivo de implementar todas las partes que van a ser comunes en cualquier otra pantalla de la interfaz gráfica. De esta forma, no se necesita duplicar todo el código que es común en cada nueva actividad que se desarrolla, lo cual sólo tiene beneficios. Es una clase que hereda de *AppCompatActivity* y tiene la intención de proporcionar flexibilidad a todas las demás actividades que vayan a heredar de esta, lo cual lo logra sobrescribiendo el método *setContentView()* con dos objetivos. El primero es que detecta si recibe o no el identificador de un recurso de disposición de vistas o *layout* para, en función de esto, permitir a cierta actividad utilizar una disposición particular completa o cargar los elementos comunes y luego los elementos propios de esa actividad. Lo segundo es que, dentro de la opción de cargar los elementos comunes, permite que algunos sean configurables o habilitables para cada actividad, es decir, que una actividad concreta podría necesitar que se visualicen la mayoría de los elementos comunes, pero no todos.

El primer objetivo se logra pasando el valor *-1* como argumento de *setContentView()* cuando se deseen reemplazar por completo los elementos comunes y el identificador de un recurso cuando se quiere cargar la disposición de dicho recurso para la parte propia de esa actividad, pero con los elementos comunes deseados. El segundo objetivo se logra estableciendo que la condición de las estructuras *if-else* que muestran u ocultan cada elemento común configurable sea el valor booleano que retornan métodos específicos para cada

elemento común. De esta forma, en cada actividad se puede configurar la visibilidad de cada elemento común sobrescribiendo cada uno de estos métodos de interés para que devuelvan el valor booleano *false* en lugar del valor booleano *true* que todos estos métodos retornan por defecto en su implementación en la clase abstracta.

De forma más concreta ahora, para comprender mejor lo anterior se explica la disposición de vistas de esta actividad común y que usarán todas las demás.

Se utiliza un *DrawerLayout* que cuenta con un *RelativeLayout* y un *NavigationView*. Este último será la barra lateral de navegación. Se trata de uno de los elementos comunes configurables y está formada por un *ListView* que, en caso de estar el elemento habilitado, carga su contenido de forma dinámica según la información disponible en el modelo tal y como se explica más tarde. La disposición relativa cuenta con un *Toolbar* que será la barra superior y un *FrameLayout* inicialmente vacío que supone el resto de la superficie de pantalla disponible o no ocupado por los elementos comunes para que cada actividad cargue en este sus disposiciones de vistas propias. La barra superior no es configurable, aunque sí lo son algunos de los controles que contiene como es el caso del botón de captura de pantalla y el botón por defecto oculto reservado para futuros usos. La barra superior cuenta con un control para mostrar y ocultar la barra de navegación lateral, un control para volver atrás en la pila de actividades, un control para volver a la pantalla inicial de la aplicación y también cuenta con el título inicialmente vacío que cada actividad concreta carga dinámicamente en tiempo de ejecución.

Si se desea editar u ocultar la barra superior, puesto que esta no es configurable, se tendría que reemplazar la disposición de vistas que por defecto se carga con los elementos comunes llamando al método *setContentView()* desde la subclase pasando como argumento el identificador de un fichero de recurso con la disposición de vistas completa en lugar del valor *-1*.

Respecto a cómo se carga de forma dinámica el contenido de la barra lateral de navegación, esta clase cuenta con la lógica de vista (aquella referente sólo a la visualización y presentación de la información obtenida del modelo a través del controlador) necesaria. De las ocho secciones con las que cuenta la aplicación, sólo lo que se muestra de estas en la barra lateral de dos de ellas va a ser variable según la información contenida en el modelo (aunque haya otras secciones cuyo contenido también sea variable). Es el caso de las secciones Capítulos y Podcasts, que se van a mostrar como subelementos para cada módulo registrado en el contenido de la aplicación. Así pues, primero se pide al controlador una lista de los módulos (la longitud de la lista es indefinida de antemano). Después, proporcionando la lista recibida, la lista de nombres asociados a cada sección y el array de las restantes seis secciones fijas, pide al controlador que conforme la lista final de elementos que deben mostrarse en la barra lateral. Para terminar, sólo carga esta lista final que recibe. Cabe destacar que el controlador no obtiene directamente la lista de nombres de las secciones porque estos los obtiene la lógica de vista a partir del fichero de recursos de cadenas de texto *strings.xml* según el idioma configurado en el terminal que ejecuta la aplicación, es decir, son variables según el idioma seleccionado por el usuario.

Como sólo mostrar la barra lateral no sirve de nada sin que sus elementos puedan ser pulsados por el usuario para generar una solicitud a las capas de la aplicación que desencadene

una acción, esta clase también cuenta con la implementación necesaria para a partir de la posición del elemento pulsado en la lista total de elementos de la barra lateral conociendo la longitud de esta (no se conoce antes de cargarla, pero cuando se pulsa uno de sus elementos ya está cargada y se conoce su longitud) y su estructura se puede deducir la sección y si compete el módulo concreto que ha seleccionado el usuario para generar el *intent* con los argumentos necesarios que va a producir que se llame y cargue la actividad correspondiente.

Como es de esperar, también implementa los métodos asociados al evento de pulsación sobre los controles (botones) antes descritos de la barra superior (*Toolbar*). Para el control de las capturas de pantalla puede mencionarse que sólo implementa la parte de la lógica de vista que se encarga de obtener la instantánea de la pantalla actual y generar a partir de esta un *Bitmap*. Después delega en el controlador (este a su vez lo hace en la capa correspondiente) el guardado de este *Bitmap* en el sistema de ficheros como un fichero de formato imagen en la ubicación que se haya decidido. Como es norma general, la vista cuando solicita operaciones que no son de su competencia al controlador sí que debe esperar al menos a recibir el estado de éxito o fracaso de la realización de dicha solicitud para mostrar retroalimentación al usuario de la acción que este ha realizado y que ha desencadenado que se haga la solicitud.

Se implementan unos métodos con la lógica de vista necesaria para que fácilmente pudiera mostrarse con el aspecto y la ubicación correcta desde una subclase el control extra tipo *Button* de la *Toolbar* o barra superior reservado para futuros usos. Uno de estos métodos llama a un método abstracto (sólo declarado y sin implementación) que la subclase deberá implementar. El motivo de este método abstracto es que se desconoce el fichero de recurso *drawable* que se utilizará en cada subclase que necesitaría de este control adicional. Por ello, la implementación del método abstracto vale con que simplemente sea una llamada al método *setBackgroundResource()* pasando el recurso *drawable* deseado que ejecutará sobre la referencia de tipo *Button* que recibe dicho método abstracto como parámetro.

Consecuentemente, también será tarea de la subclase implementar el método de acción asociado al control con el atributo *onClick* en xml. Este método de acción deberá tener la firma *public void accionBotonExtra(View vista)*.

8.1.2 Actividad principal

Es la actividad que origina a la pantalla principal que se muestra al abrir la aplicación. El contenido que debe mostrar esta actividad es fijo, es decir, es conocido de antemano y no depende de la información de la que dispone el modelo. Como se trata de una pantalla especial porque tiene una apariencia diferente a la común de la mayoría de las restantes actividades, es decir, aunque es similar en algunos aspectos, tiene su propia disposición de vistas completa, de primeras se piensa en que no sea una subclase de la actividad común. A pesar de ello, se ha implementado como una subclase de la actividad común. Esto se debe a que la actividad común es compatible con el reemplazo completo de la disposición de vistas y sobre todo porque ya cuenta con la lógica de vista necesaria para generar los *intent* correspondientes para las actividades correspondientes de cada sección. De otra forma, esta lógica se hubiera tenido que duplicar en esta actividad. Se pueden ver como decisiones de diseño que, primero, esta lógica sólo estuviera implementada en un único lugar y segundo que esta lógica debía estar en la

actividad común en lugar de en la actividad principal para lograr que cualquier otra actividad de la aplicación que la pudiera necesitar (aunque sólo fuera para la barra lateral) ya contara con ella.

Así pues, hereda de la actividad común y llama al método *setContentView()* de su superclase con un argumento de valor distinto a *-1*. Se debe mencionar también que es esta actividad la que de forma automática al iniciarse la aplicación genera y manda al controlador la solicitud de sincronización de la base de datos local con la remota de la máquina servidora. Aunque se explica en detalle más adelante, como esta sincronización se realiza como una única transacción sólo hay dos posibles resultados para la sincronización: el éxito y el error. Dicho de otro modo, no se puede realizar a medias la sincronización porque se ve esta como una única unidad indivisible. Esto es interesante desde el punto de vista de que es seguro abortar la sincronización en cualquier momento y no supone una operación crítica que por ello no se permita cancelar. Enlazando de nuevo esto con la parte de la vista, entre otros, se aprovecha el ciclo de vida de las actividades en Android para sobrescribir métodos como *onDestroy()* y *onPause()* para generar y mandar al controlador la petición de cancelar la sincronización cuando el usuario manualmente la cancela o cuando se cancela de forma automática por otras circunstancias. En el caso de además finalizarse la aplicación intenta esperar que se procese la petición de cancelar la sincronización antes de abortar la ejecución de la aplicación.

8.1.3 Visualizadores de recursos

Suponen una de las partes más importantes de la vista, porque mientras que otras partes y actividades se encargan de presentar de forma ordenada y adecuada los recursos y su jerarquía clasificados según secciones, subsecciones y módulos aquí es donde se puede decir que se usan los recursos para su uso final que es el de mostrárselos al usuario final y permitir que este pueda interactuar con ellos. Aunque se han creado tantos visualizadores específicos como tipos de recursos pueden existir en el sistema se ha creado una clase abstracta que va a actuar de visualizador genérico y que va a contener bastante lógica de vista común a todos los visualizadores específicos y que de alguna u otra forma en algún momento estos van a necesitar.

8.1.3.1 Visualizador genérico

El visualizador genérico por lo tanto es una clase abstracta y hereda de la actividad común. Este recibe del *intent* algunos de los argumentos que va a necesitar el visualizador como el identificador del recurso que se ha solicitado cargar en el visualizador desde la actividad de listado de recursos y una lista de los otros identificadores de recursos contiguos (con las mismas características) al seleccionado cuya carga podría solicitarse directamente desde el visualizador sin tener que retroceder a ninguna actividad de listado.

También, como cada vez que se carga un recurso distinto directamente desde el visualizador, solicita al controlador la dirección URL del recurso y el *flag* que indica si se trata de una dirección remota o local. Además, genera y manda al controlador la petición de obtención de la versión del recurso a la máquina servidora independientemente del *flag* anterior obtenido. Siempre se obtiene la versión del recurso remoto cuando se carga un recurso en el visualizador

incluso aunque se trate de uno descargado y registrado localmente en el sistema pues el fichero asociado a este podría disponer de una nueva versión, es decir, una actualización.

Por otro lado, se sobrescribe el método *setContentView()* para extender lo que ya de por sí hace el método de igual nombre de la superclase para que el *FrameLayout*, que en adelante se conocerá como *container*, se infle con un fichero de disposición de vistas escrito en xml que, al modo de lo que ocurre con la actividad común, lo que hace es añadir una barra inferior general de los visualizadores dejando en este caso un *RelativeLayout* vacío que ocupa el resto de la superficie de la pantalla para que sea este a su vez inflado por una subclase (o sea, un visualizador específico). Esta barra inferior tendrá los controles (botones) para descargar, actualizar y borrar un recurso, los controles para avanzar y retroceder entre recursos contiguos de la lista de identificadores recibida en el *intent* y finalmente uno para mejorar más si cabe la navegabilidad de la aplicación. En relación con estos controles, a parte de las acciones descritas más adelante que se realizan al producirse un evento de pulsación sobre los mismos, el visualizador general incluye la lógica que realiza las comprobaciones necesarias para mostrar u ocultar cada uno de estos en base al *flag* que indica si se trata de un recurso registrado localmente, al *flag* de disponibilidad de actualización que el controlador establece directamente en la vista y a la posición del recurso en la lista de identificadores de recurso recibida del *intent*. Los controles para descargar y borrar un recurso sólo pueden aparecer cuando se trata de un recurso registrado localmente, y el control de actualizar que es mutuamente excluyente con el de descarga sólo debe aparecer cuando el recurso está registrado localmente y además su fichero asociado dispone de una actualización. Los controles de avanzar y retroceder un recurso de la lista deben aparecer siempre que el recurso visualizado no se encuentre en alguno de los extremos de la lista o la lista sólo esté formada por dicho recurso.

Respecto a las acciones que se llevan a cabo al detectarse un evento de pulsación sobre los controles, en el caso del control de descarga y actualización (caso particular de descarga) simplemente manda una petición de descarga del fichero asociado al recurso seleccionado al controlador y después activa una animación para indicar al usuario que la descarga está en curso. Con la intención de evitar que reiteradas pulsaciones sobre el control, mientras se encuentra una descarga en curso, den lugar a peticiones repetidas de descarga se implementa un sencillo mecanismo basado en un *flag* que lo soluciona. Como el que se muestren u oculten estos controles depende también del *flag* de disponibilidad actualización mencionado antes, se debe obtener su valor para cada recurso que se carga en el visualizador de forma previa.

Como ya se ha mencionado y aunque se explique más adelante más en detalle, es el controlador el que establece directamente este *flag* en la vista según si ejecuta con éxito o no la petición de obtención de la versión remota del recurso que se desea cargar. Dado que realizar esa petición implica la conexión a través de Internet con la máquina servidora, puede darse la situación de que la comunicación falle (o ni llegue a establecerse). Por ello, es realmente el controlador el que llama a los métodos con lógica de vista de este visualizador que actualizan la visibilidad de algunos de los controles y realizan algunas de las funciones descritas antes. Con intención de aclararlo, esa parte de la lógica de vista, aunque la llame el controlador se encuentra implementada en el visualizador porque maneja elementos de la interfaz gráfica que compete saber cómo manejarlos sólo al visualizador, es decir, el controlador realiza una petición directamente sobre la vista, pero sigue siendo la vista la que sabe mejor cómo atender esa petición.

Considerando entonces “crítica” la operación de obtener la versión del recurso remoto de cara a actualizar la visibilidad de los controles, se implementa un contador de reintentos

(automáticos o no) adicionales al primer intento para reenviar la solicitud de realizar esta operación sólo si ya ha fallado antes. Cuando se agota el contador, se toma el comportamiento por defecto, que es el de mostrar la versión registrada localmente del recurso si existiera (comportamiento offline del sistema) o de no existir el recurso registrado localmente simplemente informar al usuario de la situación. En ambos casos, se ocultan todos los controles que pueden implicar una descarga.

En el caso del control de borrado se manda al controlador la petición de borrado del fichero asociado al recurso y la petición de eliminarlo del registro de recursos registrados como locales en el sistema. Aprovechando su condición de saberse borrado el recurso, puede actualizar los *flag* de los que depende la visibilidad de los controles como el de descarga, actualización y borrado.

En relación a la descarga de un recurso se debe diferenciar las dos partes de las que consta. Por un lado, está la descarga del fichero y por otra el registro del recurso como descargado. La segunda parte se realiza sólo tras completarse la primera con éxito. Dicho de otro modo, la descarga en conjunto se considera como una operación atómica. Esto podría dar la situación de que si justo se cancela la descarga en el cambio de parte, aunque quede correctamente descargado el fichero en el sistema de ficheros este no queda registrado y como tal no se considera descargado. Que ya exista el fichero de cara a una nueva solicitud de descarga del mismo fichero no supone nunca ningún problema porque las descargas siempre sobrescriben en caso de que ya exista el fichero.

La primera parte se delega en el *Gestor de Descargas* de Android que es externo a la aplicación. Aunque se ve mejor más adelante, cuando se realiza una solicitud de descarga al gestor este proporciona un identificador de descarga para que, debido al mecanismo de retrollamada que consiste en lanzar un *intent* (lleva adjunto el identificador de descarga de la petición a la que responde dicho *intent*) que se va a obtener en la aplicación que hizo la petición con un *BroadcastReceiver* con un filtro de *intent* correcto, la aplicación que originó la solicitud pueda comprobar si se trata del resultado de su petición de descarga o se ha capturado un *intent* también del gestor pero originado por la solicitud de descarga que hizo otra aplicación. Dado este mecanismo, que el *BroadcastReceiver* está gestionado por el controlador y que se decidió que si el visualizador de recursos se cerraba esto implicaba la cancelación (no el intento de cancelación) de todas las descargas de recursos en curso, puesto que la primera parte se delega y es externa a la aplicación, por necesidades de implementación se tuvo que ubicar en el visualizador otro mecanismo de retrollamada entre el controlador y la vista para que sea la vista la que en el momento correspondiente creara y pasara al controlador la solicitud de registro del recurso en el sistema como recurso descargado o recurso local. Esto último, en sí parece carecer de primeras de todo tipo de sentido porque la segunda parte de la descarga de un recurso se trata de algo que ya podría realizar directamente el controlador en el momento de activarse la retrollamada del *Gestor de Descargas*.

Sin embargo, como no podía ser de otra forma, detrás de la decisión de implementación del mecanismo de retrollamada entre el controlador y la vista existen algunos motivos que lo justifican. Por un lado, sería el controlador el que abusando directamente tuviera que llamar a parte de la lógica de la vista implementada en el visualizador y para ello necesitaría almacenar una referencia a la instancia de la actividad del visualizador que puede resultar en *leaks* (fugas de memoria). Por ejemplo, en caso de fracaso de la primera parte de la descarga se notifica con

un mensaje al usuario. Podría intentarse una implementación alternativa que evite al controlador acceder directamente a la vista con el uso de *threads*, pero aparte de ser una alternativa no exenta de nuevos problemas implica un gran aumento de la complejidad del desarrollo del sistema por no decir que pudiera no solucionar el problema realmente. En el caso de los mensajes de error se tendría que implementar algún mecanismo, que se ejecute en paralelo (*threads*), de detección de actualización del estado de la primera parte de la descarga para entre otros informar al usuario. Otros problemas que no tendrían fácil solución sin el mecanismo de retrollamada son la actualización de los controles de la barra inferior (implica actualizar los *flag* relacionados), ocultar la animación de descarga en curso y cumplir la restricción de que sólo se puede hacer la segunda parte de la descarga si existe la instancia del visualizador que ha generado dicha descarga, pues la mejor forma de asegurar esto es que sea el propio visualizador y por ello esa instancia la que inicie la solicitud de registro del recurso al controlador. De otra forma, podría cerrarse el visualizador justo al saltar la retrollamada del *Gestor de Descargas* y si el controlador no implementa algo para detectar la existencia del visualizador el recurso se registraría incluso con este cerrado. También, permite solucionar que iniciada la descarga de un recurso se pueda cambiar a otro recurso contiguo sin que esto impida que se complete la descarga del otro recurso, es decir, sin que implique su cancelación, o que incluso se pueda iniciar la descarga de este segundo teniendo a medias la anterior. La condición restringe a la existencia de la instancia del visualizador, pero no impide que se pueda realizar la segunda parte de la descarga de un recurso contiguo porque cargar otro recurso no implica cerrar el visualizador. Relacionado con esto, el mecanismo de retrollamada también permite que si se realiza el registro del mismo recurso que se encuentra cargado en el visualizador, es decir, que no se ha cambiado de recurso a uno contiguo, de forma automática se recarga el mismo recurso en el visualizador tras la descarga. Existen aún otros motivos, aunque no se consideran relevantes.

En este caso el mecanismo de retrollamada está posibilitando hacer todo lo mencionado antes y sobre todo en el momento adecuado sin obligar a bloquear mientras la aplicación y evitando tener que realizar otras implementaciones bastante más complejas. Además, no viola o no de forma relevante la independencia entre capas que presenta el patrón MVC. Tras hablar tanto de este mecanismo de retrollamada, no se ha llegado ni a mencionar en qué consiste. Simplemente junto a la petición de descarga se pasa el nombre de la clase del visualizador que origina dicha solicitud. De esta forma, tras la retrollamada del gestor de descargas de Android, el controlador gracias a contar con el nombre de la clase puede generar y lanzar un *intent* destinado a la actividad del visualizador. Utilizando los *flag* adecuados en el *intent* de acuerdo a la documentación oficial de Android, se puede muy fácilmente lograr que este *intent* en lugar de implicar la creación de una nueva instancia del visualizador (supondría eliminar la ya existente) traiga al frente la actividad a la que ya va destinada en caso de existir en la pila de actividades (ya se encuentra igualmente arriba de la pila) (Android Developers, 2018). Visto más desde la implementación, no implica llamar al método *onCreate()* del visualizador, sino que la recepción de este *intent* provoca la llamada del método *onNewIntent()* que se sobrescribe para que haga todas las tareas mencionadas de la segunda parte de la descarga. Además, cabe mencionar que gracias a *Java Reflection*, no se produce acoplamiento entre el visualizador y el controlador a pesar de que el primero pasa el nombre de la clase al segundo y más tarde este lanza un *intent* destinado al primero.

Falta mencionar que volviendo a aprovechar el ciclo de vida de las actividades en Android, se usan en este visualizador los métodos *onResume()* para solicitar al controlador que registre el *BroadcastReceiver* que se necesita para el mecanismo de retrollamada del *Gestor de Descargas* de Android, y el método *onDestroy()* para solicitar eliminar el registro del *BroadcastReceiver*, solicitar al *Gestor de Descargas* que cancele todas las descargas pendientes, solicitar cancelar la obtención de la versión del recurso remoto de la máquina servidora y destruir la instancia del visualizador que, como se ha visto, implica cancelar las descargas en la segunda parte de la descarga.

Por último, mencionar que, similar a como ocurre con la acción de borrar un recurso, descargar un recurso o actualizarlo implica conocer que se está ahora en ese estado y por ello se pueden actualizar directamente los *flag* de los que depende la visibilidad de los controles como el de descarga, actualización y borrado.

8.1.3.2 Visualizador local de PDF

Se trata de uno de los visualizadores específicos que en este caso permite cargar ficheros PDF. La parte importante es que permite cargar los ficheros desde la propia aplicación, es decir, no lo hace a través del navegador con el visualizador de PDF online de Google ni usa otra tercera aplicación externa a la que llama con un *intent*. Se ha utilizado la biblioteca *AndroidPdfViewer* de *barteksc* en su última versión estable disponible para proporcionar a HepApp la capacidad propia de cargar ficheros PDF (PdfViewer, 2018).

Este visualizador simplemente infla el *RelativeLayout* del visualizador general que se había quedado vacío en lo que antes se ha llamado *container* con la vista *PDFView* propia de la mencionada biblioteca. Para realizar esto, también sobrescribe el método *setContentView()*.

Cuando se inicia este visualizador, obtiene una referencia a esta vista y sólo si el recurso está registrado como local en el sistema, es decir, sólo si existe en el sistema de ficheros el fichero en formato PDF asociado al recurso que se desea visualizar es cuando procede a cargarlo con los métodos *fromFile()* y *load()* propios de la vista. Esto es lo que hace el método *cargarRecursoEnVisualizador()* del visualizador.

Para lograr que el recurso se cargue automáticamente en el visualizador tras completarse su descarga, se sobrescribe el método *onNewIntent()* para que llame al método *cargarRecursoEnVisualizador()* tras llamar al método de la superclase con el mismo nombre.

El comportamiento descrito por el método *cargarRecursoEnVisualizador()* también se lleva a cabo cada vez que se cambia de recurso a visualizar a uno contiguo. Esto lo logra sobrescribiendo algún método del visualizador general.

8.1.3.3 Visualizador Web

Se trata de otro de los visualizadores específicos que en este caso no se trata de nada más que del uso de *WebView* que es el cliente Web nativo de Android para que los

desarrolladores puedan utilizarlo en las actividades de sus aplicaciones sin tener que integrarlo o recurrir a otras bibliotecas, es decir, se puede utilizar como cualquier otra vista de la API Android. Este visualizador es el que se va a considerar como visualizador por defecto, por lo que es el que se utilizará para cargar un recurso que no es de un formato concreto que tenga otro visualizador específicamente asociado.

Así pues, de forma análoga al visualizador local de PDF, infla el *RelativeLayout* vacío del *container* con la vista *WebView* gracias a sobrescribir *setContentView()*.

Cuando se inicia este visualizador, obtiene una referencia a esta vista y procede a cargar el recurso a partir de su dirección URL con el método *loadUrl()*. Esto es lo que hace en este visualizador el método *cargarRecursoEnVisualizador()*.

Para lograr que el recurso se cargue automáticamente en el visualizador tras completarse su descarga, se sobrescribe el método *onNewIntent()* para que llame al método *cargarRecursoEnVisualizador()* tras llamar al método de la superclase con el mismo nombre.

El comportamiento descrito por el método *cargarRecursoEnVisualizador()* también se lleva a cabo cada vez que se cambia de recurso a visualizar a uno contiguo. Esto lo logra sobrescribiendo algún método del visualizador general.

Como el uso que se hace de este visualizador en HepApp es para cargar siempre recursos remotos a través de *HTTP* (a pesar de que pueda cargar direcciones URL locales), se fuerza a deshabilitar siempre los controles descargar, actualizar y borrar de la barra inferior del visualizador puesto que el recurso por tratarse siempre de un sitio Web externo se encuentra siempre actualizado, no se puede descargar y por lo tanto tampoco borrar.

Dado que los recursos que carga este visualizador son sitios Web externos que pueden tener enlaces a otras páginas del mismo sitio o de otros distintos, se sobrescribe el método *onKeyDown()* para que al pulsarse el botón atrás este actúe como el botón atrás del navegador (cliente Web) y no como el botón atrás en la pila de actividades de la aplicación. Se sigue disponiendo del control home en la barra superior y de la barra lateral de navegación de la actividad común.

En el método *onCreate()* se extrae del *intent* que ha llamado al visualizador el dato identificador de sección adjunto en este. Del mismo *intent*, como ya se ha visto antes, ya se encarga de extraer otros datos el método de la superclase de mismo nombre, es decir, el método *onCreate()* del visualizador general. Además, este método utiliza el identificador de sección para deshabilitar y ocultar completamente o no la barra inferior del visualizador según la sección. Esto se debe a que hay secciones conformadas por un único recurso como la sección *Cards*, *PubMed* y *Information*, es decir, que para estas secciones no hay recursos contiguos y por ello tampoco tienen sentido los controles de avanzar y retroceder recursos. En estos casos, como no se necesita ninguno de los controles de la barra inferior esta misma consecuentemente carece de sentido y se oculta. Sin embargo, como este visualizador también se utiliza para cargar los recursos de la sección *Resources*, que en este caso sí puede disponer de recursos contiguos, se comprueba que la necesidad de la barra inferior en este visualizador va en función de la sección a la que pertenece el recurso que se debe cargar en cada momento.

8.1.3.4 Visualizador de imágenes

Se trata del último de los visualizadores específicos y en este caso no utiliza ninguna biblioteca, pero si una vista propia o personalizada, es decir, una vista (hereda de View) implementada exclusivamente para HepApp.

Así pues, de forma análoga los otros visualizadores, infla el *RelativeLayout* vacío del *container* con las vistas *ImageView* y *PaintingView* (es la vista propia), y dos disposiciones de vistas (*layout*) que son los controles de los dos tipos de interacciones mutuamente excluyentes que soporta este visualizador. Para ello, en esta ocasión también sobrescribe *setContentView()*.

En el método *onCreate()* se extrae del *intent* que ha llamado al visualizador el dato identificador de subsección adjunto en este. Del mismo *intent*, como ya se ha visto antes, ya se encarga de extraer otros datos el método de la superclase de mismo nombre, es decir, el método *onCreate()* del visualizador general.

Cuando se inicia este visualizador, se obtienen las referencias de todas las vistas y controles propios de este visualizador específico y procede a cargar el recurso en la vista *ImageView* a partir de su dirección URL con el método *setImageURI()* de esta vista. Con los datos recibidos del *intent* que llama a este visualizador deduce qué tipo de recurso se está solicitando visualizar (recurso simple o recurso compuesto) y el tipo de interacción que debe permitir para ese recurso (ninguna, *switch* o pintar).

En el momento del diseño de esta parte del sistema se prefería que se pudiera cargar la imagen directamente del recurso remoto para que parecido a lo que sucede con los sitios Web del visualizador Web el contenido se mantuviera siempre actualizado de forma transparente al usuario. Finalmente, como en este caso sí que interesaba que se pudieran descargar los recursos imagen para poder usarse de forma offline (por deseo del usuario o por tener problemas de conexión), y esto implica toda la gestión de versiones y los controles asociados de la barra inferior, se observó que, como ocurre con el visualizador local de PDF, este visualizador sólo puede cargar ficheros de imagen asociados a los recursos que existen en sistema de ficheros, es decir, que se encuentran registrados como locales en el sistema (descargados). Además, como se puede comprobar en la documentación oficial, por una limitación del método *setImageURI()* no se pudiera haber implementado que los recursos de imagen se hubieran obtenido directamente desde el recurso remoto pues este método sólo admite direcciones URI locales (Android Developers, 2018).

A parte, se puede mencionar que los ficheros de imagen que admite el sistema sólo pueden ser de formato JPG o JPEG, o PNG.

Otra cosa particular mencionable de este visualizador es que el control de descarga nunca está disponible porque este visualizador al cargar un recurso, si comprueba que no se encuentra registrado como local en el sistema, realiza un sólo intento de descarga de forma automática sin interacción del usuario. A pesar de esta implementación, el usuario podría forzar que se vuelva a intentar la descarga de un recurso de imagen en caso de que el intento automático fallara saliendo y volviendo a entrar en el visualizador o directamente desde el visualizador cambiando a un recurso contiguo y volviendo al de interés, a fin de cuentas, forzando que se vuelva a intentar cargar el recurso en el visualizador. Como ya se dijo en el

visualizador general, se debe recordar que se carga un recurso en el visualizador cuando se ha obtenido la versión de su análogo remoto o tras fallar todos los reintentos indicados en el contador.

Así pues, en esta ocasión, el método *cargarRecursoEnVisualizador()* sólo carga el recurso en el visualizador cuando este se encuentra descargado y si no se encuentra descargado solicita su descarga que, además, al completarse, por el mecanismo de retrollamada explicado volvería a llamar al método que carga el recurso esta vez sí estando este completamente descargado.

Para lograr que el recurso se cargue automáticamente en el visualizador tras completarse su descarga, se sobrescribe el método *onNewIntent()* para que llame al método *cargarRecursoEnVisualizador()* tras llamar al método de la superclase con el mismo nombre. En el caso de estar visualizando un recurso compuesto (colección de recursos simples) el recurso simple que se carga automáticamente es el que se está visualizando en ese momento.

El comportamiento descrito por el método *cargarRecursoEnVisualizador()* también se lleva a cabo cada vez que se cambia de recurso a visualizar a uno contiguo. Esto lo logra sobrescribiendo algún método del visualizador general. En el caso de estar visualizando un recurso compuesto cuando se carga un recurso compuesto contiguo el recurso simple que se carga automáticamente en el visualizador es el recurso simple representativo de ese recurso compuesto. A parte, cambiar entre recursos simples de un recurso compuesto se puede ver también como cargar un recurso contiguo de una sublista, es decir, se puede ver como una lista de sublistas de recursos simples donde cada sublista es un recurso compuesto. La única particularidad es que en cada sublista hay designado un recurso simple representativo que al cambiar de una sublista a otra contigua se carga este por defecto.

Con respecto a las interacciones, la interacción pintar y la *switch* dan nombre a dos de las subsecciones de la sección imágenes. Como este es el único visualizador que se usa en esta sección, está preparado para cargar todos los recursos imagen de todas las subsecciones a pesar de las particularidades que pueda presentar cada una. De las cinco subsecciones presentes, sólo esas dos relacionadas con los tipos de interacción mencionados tienen particularidades y las demás sólo existen por una cuestión de organización de contenidos pues todas sus imágenes se tratan como recursos simples de imagen sin ningún tipo de interacción.

En referente a la interacción *switch*, hablar de ella y de recursos compuestos se puede entender como un sinónimo porque este tipo de interacción sólo consiste en mostrar unos controles *RadioButton* (cada uno con el título de un recurso simple) que permitan cambiar entre los recursos simples que conforman el recurso compuesto que se ha visualizado. Es más, el concepto de recurso compuesto se puede ver como la forma en la que se ha modelado, organizado y estructurado a nivel de diseño e implementación el concepto o idea a nivel funcional que da nombre a la interacción *switch*. Por ello en este tipo de interacción, el título de los controles *RadioButton* (limitados a un número máximo de 4 controles) se carga de forma dinámica según la información que tiene el modelo (que el visualizador obtiene a través del controlador) sobre los recursos simples que conforman al recurso compuesto. Como ya se ha mencionado, en este caso especial los controles para avanzar y retroceder a recursos contiguos esta vez se refieren a recursos compuestos contiguos en lugar de a recursos simples contiguos. De esta forma, al cambiar de recurso compuesto se actualizan todos los títulos de los controles

RadioButton y el título en la barra superior pues el recurso compuesto tiene un título propio para designar a ese conjunto concreto de recursos simples que lo forman. También se obtiene a través del controlador el recurso simple representativo de cada recurso compuesto.

En el caso de la interacción pintar esta sólo se puede aplicar sobre recursos simples, de forma concreta, a los que pertenecen a la subsección de igual nombre que la interacción. La interacción consiste en que sobre la vista *ImageView* se coloca la vista *PaintingView* abarcando la misma o más superficie que la otra vista. La vista *PaintingView* es la vista personalizada que se ha creado exclusivamente para esta interacción. Esta vista se explica con detalle más abajo en el subapartado *PaintingView*. Como su nombre indica, esta interacción consiste en permitir al usuario pintar sobre el recurso de imagen visualizado para realizar, si se desea con diferentes colores, anotaciones. Los controles de esta interacción son por ello dos botones, uno para abrir el diálogo *ColorPickerDialog* que permite elegir un color de una lista de colores dada y otro para borrar todas las anotaciones hechas (Color Picker, 2018).

Por defecto, en el visualizador la vista *PaintingView* y los controles de ambas interacciones se encuentran ocultos, así, cuando se solicita cargar un recurso de una subsección con interacción se habilitan y hacen visibles estos elementos.

8.1.4 Listar componentes

Son las actividades que se encargan de presentar de forma ordenada y adecuada los recursos y su jerarquía clasificados según secciones, subsecciones y módulos. Aunque se han creado algunas actividades de listado específicas para pantallas cuyo contenido es fijo y conocido de antemano, se ha creado una clase abstracta que va a contener bastante lógica de vista común a todas estas actividades y que de alguna u otra forma en algún momento estas van a necesitar. El contenido de algunas secciones puede ser simplemente una lista de recursos o tener más niveles de jerarquía y ser un listado de módulos siendo cada módulo un listado de recursos. Por ejemplo, en el caso de la sección de imágenes hay cinco subsecciones y una de ellas es un listado de módulos.

8.1.4.1 Actividad de listado general

Esta actividad genérica por lo tanto es una clase abstracta y hereda de la actividad común. Esta recibe del *intent* que la llama algunos de los argumentos que va a necesitar el como los identificadores de la sección, el módulo y la subsección, y el título del listado que se ha solicitado cargar.

Por otro lado, se sobrescribe el método *setContentView()* para extender lo que ya de por sí hace el método de igual nombre de la superclase para que el *container*, se infle con un fichero de disposición de vistas escrito en xml que, al modo de lo que ocurre con la actividad común, lo que hace es añadir en la parte de abajo un par de controles que estarán inicialmente ocultos y servirán para avanzar y retroceder páginas del listado en el caso de que el listado fuera lo suficientemente largo, dejando en este caso un *GridLayout* vacío que ocupa el resto de la

superficie de la pantalla para que sea este a su vez inflado por una subclase (o sea, actividad de listado específica). En relación con estos controles, a parte de las acciones que se realizan al producirse un evento de pulsación sobre los mismos, la actividad de listado general incluye la lógica que realiza las comprobaciones necesarias para mostrar u ocultar cada uno de estos en base a la posición de la página que se está visualizando actualmente en la lista de páginas del listado de elementos. Estos deben aparecer siempre que la página cargada no se encuentre en alguno de los extremos de la lista o la lista sólo esté formada por dicha página.

La actividad de listado deduce a partir de la presencia y el valor, o ausencia de cada uno de los identificadores recibidos en el *intent* que la llama, de qué tipo de listado se trata. Se debe destacar que el método *getIntExtra()* del *intent* permite especificar un valor por defecto en caso de que un identificador no exista adjunto en el *intent* que ha llamado a la actividad de listado. Por ello, los puntos de otras actividades en los que se llama a una actividad de listado pueden no incluir todos los identificadores que espera recibir la actividad de listado sin que esto suponga un problema. Es más, la ausencia de alguno de estos identificadores como adjuntos del *intent* permiten saber a la actividad de qué tipo de listado se trata (pues tomarán el valor por defecto). Entonces, los tipos de listados que se pueden deducir de estos identificadores de más general a más específico respectivamente son listado de recursos de una sección, listado de módulos de una sección, listado de recursos del módulo de una sección, listado de los recursos de una subsección de una sección, listado de módulos de una subsección de una sección y listado de recursos de un módulo de una subsección de una sección. Este orden es importante, porque realmente la deducción anterior se realiza pidiendo al controlador cada vez un listado en el inverso a ese orden (de más a menos específico). El controlador, delegando tareas en otras capas y subsistemas, y según la información presente en el modelo, devolverá o no la lista correspondiente al valor de los identificadores que la vista le ha pasado. Tan pronto como se obtenga una lista se para de pedir los restantes tipos de listas y se sabe que el tipo de lista es aquella cuya petición al controlador ha resultado exitosa.

Dada una lista cargada, por ejemplo, una lista de módulos de una sección, seleccionar uno de los elementos de esa lista podría generar otra nueva lista, en el ejemplo, una lista de los recursos del módulo seleccionado para la sección determinada, y puesto que el controlador podría proporcionar una lista vacía (no hay recursos asociados al módulo), se debe distinguir entre no recibir una lista y recibir una lista vacía. En el caso de no recibir ninguna lista, ni siquiera vacía, tras pedir todos los tipos de listas al controlador se debería notificar la situación de error al usuario. Sin embargo, al recibir una lista vacía se debe parar que se pidan a continuación otros tipos de listas y mostrar al usuario la lista vacía (se muestra una animación que lo indica).

La actividad de listado general restringe del *GridLayout* que ha quedado vacío que este tenga hasta un máximo de ocho elementos *RelativeLayout* con unos identificadores determinados y que cada uno de estos elementos debe tener una vista *ImageView* para el icono del elemento, una vista *TextView* para el título del elemento, una vista *Button* y que pueda tener o no una vista *TextView* para la descripción del elemento. Al igual que con los *RelativeLayout*, todas estas vistas de estar presentes deben tener un identificador único y conocido. El motivo de esto es que exista libertad en la presencia o ausencia de descripción de cada elemento y libertad en el aspecto o estilo de cada elemento como, por ejemplo, los bordes de los elementos,

el tamaño de los elementos, el número de elementos, las separaciones vertical y horizontal entre elementos, la ubicación del icono y título dentro del elemento, etc…

La actividad, al cargar el listado lo que hace es, puesto que conoce los identificadores de todas las vistas indicadas antes, obtiene las referencias de todas. Respecto al número de elementos del *GridLayout*, sabe que si falla al obtener las referencias (método *findViewById()*) de algunos de los elementos es que no están presentes porque sólo existe un máximo en el número de estos. También sabe que de cada elemento para el que obtenga la referencia correctamente va a poder obtener las referencias de las vistas obligatorias en cada elemento que se han indicado antes. Del mismo modo que con el número de elementos del *GridLayout*, en función de si para cada elemento puede obtener una referencia a la vista de la descripción del elemento puede saber si esta está presente o no. El número de elementos en el *GridLayout* va a marcar el número de elementos a mostrar por página de la lista que se obtiene del controlador (a partir de la información del modelo).

También se debe observar que podría ocurrir que la actividad de listar elementos se llamara a si misma al cargar la lista de recursos de un módulo seleccionado en la lista anterior. Aunque se comenta más adelante, lo que se hace es recargar la actividad en lugar de realizar llamadas a si misma si la naturaleza de la lista origen y destino lo permiten, es decir, los elementos de ambas listas tienen el mismo estilo (sólo cambian en las partes dinámicas de cada elemento) y ambas listas tienen el mismo número de elementos por página. Otra forma de verlo es cuando se trataría de sustituir llamadas a sí misma entre una misma actividad de listado específica (subclase de la general). Por lo tanto, la diferencia entre llamar a una nueva actividad de listado y recargar la propia actividad es la necesidad de tomar nuevas referencias de todas las vistas o la posibilidad de reutilización de las referencias tomadas al principio para cada vista porque estas sólo dejan de valer cuando se infla de nuevo el *GridLayout*, algo que sólo ocurre una vez al principio al cargar una nueva actividad de listado.

Por otro lado, al avanzar y retroceder páginas de la lista de elementos puede que no queden suficientes como para llenar toda la página. Por ello, esta actividad tiene implementada la lógica de vista que detecta esta situación y en función de ello oculta, al cargar una página, los elementos del *GridLayout* que se quedan sin elemento de la lista que mostrar. Al retroceder de página también lo detectaría y volvería a mostrar los que se habían ocultado si fuera necesario. Incluso al cambiar entre listados de semejante aspecto o estilo, o sea, compatibles.

También cuenta con la lógica de vista necesaria para deducir las correspondencias entre la lista de elementos y la lista de páginas. Explicado de otra forma, puede calcular cual es el primer elemento que debe listar en función de la página que se va a mostrar y sabiendo cuantos elementos se muestran por página deducir después cuantos elementos a partir del primero debe cargar.

Ejemplificando lo anterior, en un listado de quince recursos con páginas de tamaño de seis elementos por página, si la numeración de los recursos comienza en uno podría deducir que si se desea ver la página dos debe cargar dinámicamente los recursos del siete al doce ambos incluidos. En este caso calcularía que el primer elemento es el siete y como aún quedan suficientes recursos como para llenar la página pues toma cinco más a partir del siete hasta el doce. Si en lugar de la página dos se cargara la página tres se tomarían los recursos del trece al

quince inclusive. En esta ocasión, de nuevo calcula que es el trece el primero que debe mostrar y como no tiene suficientes recursos para completar la página toma todos los elementos restantes hasta completar la página ocultando los elementos del *GridLayout* que se hayan quedado sin recursos que mostrar.

La idea es que al cargar una página son precisamente todas esas vistas obligadas y con identificadores conocidos en cada elemento del *GridLayout* las que cambian y se cargan de forma dinámica en tiempo de ejecución según lo que se obtiene del controlador que depende de la información que tiene el modelo. Cuando se listan los módulos, estos tienen una imagen-icono propio y similar para todos independientemente de la sección. Cuando se listan recursos de una sección, estos toman como imagen-icono el de la sección y todos el mismo.

Todas las vistas *Button* que en un listado reciben el evento de pulsación, llaman a un mismo método que primeramente deduce qué botón se ha pulsado a partir del identificador de la vista que ha provocado la llamada a dicho método. Esto, que es el equivalente a saber el índice del elemento que se ha pulsado en una página, permite deducir qué elemento del listado de elementos (listado de recursos, por ejemplo) ha sido seleccionado por el usuario. Sabiendo qué elemento se ha seleccionado y sabiendo el tipo de listado que se ha deducido como se ha explicado anteriormente, se puede deducir qué acción llevar a cabo. Básicamente las acciones son de dos tipos. Si es una lista de recursos la acción es generar y lanzar un *intent* con los datos necesarios al visualizador de recursos específico correspondiente según el tipo de recurso. Para cualquier otra lista, la acción o es recargar la actividad o es generar y lanzar un *intent* con los datos necesarios para llamar a otra actividad de listado específica. En este segundo caso, la actividad de listado general no puede aportar una implementación porque no conoce lo suficiente como para saber si deberá recargar la actividad o llamar a otra actividad de listado. Así pues, esta implementación (decisión a final de cuentas) le corresponderá ponerla a la actividad de listado específica que herede de esta general.

Finalmente, la actividad de listado general también plantea que sobrescribiendo el método *onBackPressed()* se puede lograr que, en una actividad de listado que muestra una lista de elementos la cual se ha cargado tras seleccionar un elemento de otra lista de elementos anterior ofrecida por la misma actividad de listado, al pulsar el control volver o atrás se retroceda a la lista anterior en lugar de retroceder en la pila de actividades que supondría el cierre de la actividad de listado actual. Esto se puede lograr implementando manualmente una pila de llamadas a la propia actividad desde que su instancia existe, es decir, guardando manualmente en una pila los datos adjuntos del *intent* que llamó primeramente a la actividad y posteriormente guardando en esta los datos empleados en cada recarga de la misma actividad. La otra opción es que sólo se sobrescribiera el método añadiendo que este llame a otro método abstracto (sólo se ha declarado y carece de implementación) para forzar a una subclase que herede de esta que proporcione una implementación para dicho método que, como ocurría antes, por tratarse de un nivel más específico supiera como actuar para lograr este efecto sin tener que necesitar de una pila. Si la actividad de listado más específica (subclase) no supiera como hacer esa implementación alternativa porque no conoce los suficientes datos o porque simplemente no es posible realizarla de otra forma siempre podría implementar el mecanismo de la pila en ese momento o ese nivel. Es más, siempre se podría implementar el mecanismo de la pila en la actividad de listado general y dejar elegir a la subclase si utilizar este mecanismo o poner uno propio alternativo.

8.1.4.2 Actividad de listado de módulos

Es una actividad específica de listado de elementos y por ello hereda de la actividad de listado de elementos general.

Esta actividad infla el *GridLayout* de la actividad de listado de elementos general que se había quedado vacío con ocho *RelativeLayout* cada uno con las vistas de la imagen-icono, título, descripción y botón de acción. Para realizar esto, también sobrescribe el método *setContentView()*. Estos elementos *RelativeLayout* tienen su estilo propio de bordes, sombreado, tamaño y posiciones de las vistas que lo conforman entre otros.

A pesar de que la superclase es compatible con más tipos de listas, se espera utilizar esta actividad para mostrar listas sólo de los tipos listado de recursos de una sección, listado de módulos de una sección, listado de recursos del módulo de una sección y listado de módulos de una subsección de una sección.

El método que gestiona la pulsación de una de las vistas *Button* de los elementos de la lista, llevará a cabo una acción distinta según del tipo de lista del que se trate. Si se trata de un listado de módulos de una subsección de una sección, la acción se debe implementar y será llamar a la actividad específica de listado de imágenes. Si se trata de un listado de recursos del módulo de una sección la acción ya está implementada en la superclase y será llamar al visualizador de recursos correspondiente. Si se trata de un listado de módulos de una sección, la acción se debe implementar y será recargar la actividad porque la acción implica mostrar otro tipo de listado del que también se encarga esta misma actividad. Se recarga la actividad simplemente teniendo en cuenta que ahora ya sí se cuenta con un identificador de módulo. Si se trata de un listado de recursos de una sección la acción ya está implementada en la superclase y será llamar al visualizador de recursos correspondiente.

En referencia al método *onBackPressed()* este se sobrescribe sin necesitar hacer uso del mecanismo de la pila. Sólo en uno de los cuatro tipos de listados que gestiona esta actividad (respectivamente el segundo de los mencionados antes) se lleva a cabo como acción el de recargar la actividad en lugar de llamar a otra actividad. Por ello, en este método sólo se debe distinguir si se trata de una lista del tipo listado de recursos del módulo de una sección. Si se trata de otro tipo distinto sólo se debe llamar al método de la superclase de igual nombre que a fin de cuentas producirá volver a la actividad anterior según la pila de actividades. Si se trata de este tipo concreto la acción a realizar se debe implementar y consiste en asignar el valor *-1* al identificador del módulo y tras esto, recargar de nuevo la actividad.

El comportamiento para la decisión de la imagen-icono de los elementos descritos en la actividad de listado de elementos general es realmente el que sigue esta actividad específica de listado y sólo esta. Aunque se podría trasladar esta lógica de decisión de la actividad general a esta actividad específica se mantiene en la general pues se considera un comportamiento por defecto o común a todas las actividades de listado de elementos, a pesar de que se sepa que sólo una de las dos actividades de listado específicas que existen la use.

8.1.4.3 Actividad de listado de imágenes

Es una actividad específica de listado de elementos y por ello hereda de la actividad de listado de elementos general.

Esta actividad infla el *GridLayout* de la actividad de listado de elementos general que se había quedado vacío con seis *RelativeLayout* cada uno con las vistas de la imagen-icono, título y botón de acción. Para realizar esto, también sobrescribe el método *setContentView()*. Estos elementos *RelativeLayout* tienen su estilo propio de bordes, sombreado, tamaño y posiciones de las vistas que lo conforman entre otros. En este caso, no se utiliza el máximo posible de elementos del *GridLayout* a cambio de poder asignar más superficie de pantalla (más tamaño) a cada uno de los seis elementos.

A pesar de que la superclase es compatible con más tipos de listas, se espera utilizar esta actividad para mostrar listas sólo de los tipos listado de los recursos de una subsección de una sección y listado de recursos de un módulo de una subsección de una sección (los dos tipos restantes que no gestionaba la otra actividad de listado específica).

El método que gestiona la pulsación de una de las vistas *Button* de los elementos de la lista, llevará a cabo siempre la acción ya implementada en la superclase de llamar al visualizador de recursos correspondiente.

En referencia al método *onBackPressed()*, en este caso no es necesario sobrescribirlo pues no debe hacer uso del mecanismo de la pila en caso de que estuviera implementado y siempre interesa que finalice la actividad de listado para retroceder a la actividad que la llamó, es decir, a la actividad anterior según la pila de actividades.

El comportamiento para la decisión de la imagen-icono de los elementos que se listan en esta actividad de listado específica es la imagen-icono de la sección de imágenes por defecto. Esta se sustituirá por la imagen propia de cada recurso simple cuando este se encuentre registrado como local en el sistema (descargado). Para los recursos compuestos ocurre lo mismo, pero sólo para el recurso simple representativo de dicho recurso compuesto. Con este comportamiento es posible encontrar mezclados en un listado algunos elementos de la lista con la imagen propia del recurso (o del recurso simple representativo) y otros con el icono de la sección de imágenes.

8.1.4.4 Actividad de listado de subsecciones de imágenes

Aunque se trata de una actividad de listado, dado que la lista que se muestra en esta actividad es fija y conocida de antemano (también lo es el contenido de los elementos listados, es decir, la imagen-icono, título y descripción si la hubiera) se ha decidido crear una actividad propia que no es subclase de la actividad de listado de elementos general y cuenta con un fichero de recurso xml de disposición de vistas propio.

Sin embargo, sí es subclase de la actividad común y por ello infla el *container* con su mencionada disposición de vistas propia a través de sobrescribir el método *setContentView()*.

Esta actividad va a listar las subsecciones de la sección imágenes. Por ello, en esta actividad la imagen-icono de cada elemento será el de la sección de imágenes.

El método que gestiona la pulsación de una de las vistas *Button* de los elementos de la lista, llevará a cabo siempre la acción de llamar a la actividad específica de listado de imágenes

salvo para la subsección de figuras de capítulos que llamará a la actividad específica de listado de módulos.

8.1.4.5 Actividad de listado de calculadoras

Aunque se trata de una actividad de listado, dado que la lista que se muestra en esta actividad es fija y conocida de antemano (también lo es el contenido de los elementos listados, es decir, la imagen-icono, título y descripción si la hubiera) se ha decidido crear una actividad propia que no es subclase de la actividad de listado de elementos general y cuenta con un fichero de recurso xml de disposición de vistas propio.

Sin embargo, sí es subclase de la actividad común y por ello infla el *container* con su mencionada disposición de vistas propia a través de sobrescribir el método *setContentView()*.

Esta actividad va a listar las distintas calculadoras disponibles en la sección calculadoras. Por ello, en esta actividad la imagen-icono de cada elemento será el de la sección de calculadoras.

El método que gestiona la pulsación de una de las vistas *Button* de los elementos de la lista, llevará a cabo siempre la acción de llamar a la actividad específica de calculadora correspondiente. Aunque se ve más adelante, para las calculadoras también se han utilizado mecanismos de herencia similares a los utilizados hasta el momento y cada calculadora se implementa con una actividad específica propia que es subclase de una más general.

8.1.4.6 Actividad principal

Aunque ya se ha hablado sobre ella antes en el subapartado *Actividad principal*, simplemente se vuelve a mencionar aquí desde el punto de vista de las actividades de listado de elementos pues a fin de cuentas se trata también de una actividad de ese tipo donde la lista es fija y conocida de antemano (también lo es el contenido de los elementos listados, es decir, la imagen-icono, título y descripción si la hubiera). Constituye una actividad propia que no es subclase de la actividad de listado de elementos general.

Esta actividad lista las distintas secciones con las que cuenta HepApp. Por ello, en este caso la imagen-icono de cada elemento será el de cada sección.

El método que gestiona la pulsación de una de las vistas *Button* de los elementos de la lista, como ya se indicó llevará a cabo siempre la acción de llamar a otra actividad correspondiente para gestionar cada sección de la forma adecuada.

8.1.5 Calculadoras

Son las actividades que se encargan de presentar al usuario un formulario para introducir datos médicos sobre un paciente de forma que la aplicación los use para calcular diferentes algoritmos y mostrar después los resultados. Esta sección de la aplicación es

completamente fija y de contenido conocido de antemano. Sirve para ayudar al diagnóstico de hepatología y se organiza en varias calculadoras parciales y una calculadora completa. Cada una de las calculadoras cuenta con una actividad propia y una o varias disposiciones de vistas propias. En este apartado se habla sólo de las actividades de las calculadoras pues la actividad de listado de las calculadoras ya se ha tratado en apartados anteriores. De nuevo, como para otras partes de la vista se ha creado una clase abstracta que va a contener bastante lógica de vista común a todas estas actividades y que de alguna u otra forma en algún momento estas van a necesitar. También se ha creado una segunda clase abstracta para enfocar más a la primera hacia las calculadoras parciales. En esta ocasión hay presencia a veces de dos niveles de herencia a partir de la actividad común. Primeramente, se realizó el desarrollo de la calculadora completa y a partir de esta el de las calculadoras parciales. Como las calculadoras utilizan varias estructuras *Map* de colección de información, se hace uso de un gran número de constantes y estas se han ubicado en una interfaz en el paquete de las calculadoras. Aunque esto no está considerado una buena práctica se ha implementado de esta forma por simplicidad puesto que de esta forma se logra centralizar las constantes en un único punto y se pueden utilizar directamente sin necesidad del acceso estático en caso de haberse utilizado una clase para el mismo fin (tampoco sería mucha mejor práctica). Esta interfaz es suficiente con que la implemente la actividad abstracta de la calculadora general porque todas las actividades que heredaran de ella tienen inmediatamente acceso directo también a dichas constantes.

8.1.5.1 Actividad calculadora general

Esta actividad genérica por lo tanto es una clase abstracta y hereda de la actividad común. Implementa la interfaz de constantes de las calculadoras y que contiene las claves utilizadas en las estructuras *Map* (es una interfaz) utilizadas. Esta clase se ha desarrollado teniendo más en mente la calculadora completa que las calculadoras parciales, por ello que se haya creado una segunda clase abstracta genérica que se enfoque más a las calculadoras parciales.

Al empezar, inicializa varios *HashMap*, carga algunos valores por defecto, inicializa el diálogo *AlertDialog* que se utilizará para mostrar al usuario mensajes de error de validación con los valores introducidos en los campos del formulario y llama al método *setContentView()*.

Por otro lado, se sobrescribe el método *setContentView()* para extender lo que ya de por sí hace el método de igual nombre de la superclase para que el *container*, se infle con un fichero de disposición de vistas escrito en xml que, al modo de lo que ocurre con la actividad común, lo que hace es proporcionar una estructura visual común a todas las calculadoras pues añade cuatro *RelativeLayout* vacíos que deberán ser después inflados por las subclases. Dos de estos, por defecto, no serán visibles y serán utilizados para contener las capas ocultas de los ajustes de la calculadora y la capa de más información. Otro es para añadir en la parte de abajo una barra inferior con algunos controles (botones) que entre otros sirven para mostrar alguna de las capas ocultas. El último es el que ocupa el resto de la superficie de la pantalla y se dedicará a contener una de las pantallas (formulario) de la calculadora (la calculadora completa tiene varias pantallas y las calculadoras parciales sólo una). Estos *RelativeLayout* se encuentran vacíos pues la disposición de vistas que van a contener podrían ser diferentes en cada calculadora.

Esta clase implementa dos métodos genéricos para inflar disposiciones de tipo *RelativeLayout* y de tipo *LinearLayout*. Reciben como argumento el identificador del *layout* a inflar, el identificador del fichero de recurso xml de disposición de vistas con el que se va a inflar y la clave con la que se va a guardar en un mapa la referencia al *layout* inflado. Las subclases utilizarán estos métodos con los argumentos adecuados para inflar los cuatro *RelativeLayout* de antes y otros vacíos que pueda haber en los ficheros de recurso xml que se utilicen para inflar a estos.

Se declara un método abstracto (sólo declarado y cuya implementación deberá realizar la subclase) de nombre *cargarPantalla()* que tendrá por argumento la pantalla que debe cargarse. Este argumento será del tipo de un *Enum* que se ve más adelante.

El formulario de cada posible pantalla de una calculadora está formado por varias entradas de datos de dos tipos: el de la selección de sólo un control *Button* de un grupo de controles de este tipo, y el de un campo de introducción de texto.

En este punto, se debe mencionar que la clase cuenta con varios arrays destinados a almacenar para cada pantalla las constantes de la interfaz que se refieren a esa pantalla. Por ejemplo, existe un array que tiene en un orden concreto todas las constantes de la interfaz con las claves del mapa que almacena las referencias a las vistas de todos los controles de tipo *Button* de una pantalla concreta. También hay un array asociado al mapa de referencias de vistas de campos de introducción de textos (*EditText*) de una pantalla. Un array asociado al mapa de referencias de vistas de etiquetas de textos (*TextView*) de una pantalla. Existe un mapa que guarda de forma acumulativa los valores introducidos hasta el momento en cada entrada de datos (campos del formulario) de cada formulario de cada pantalla en la que se han ido introduciendo datos. Asociado a este mapa hay un array con las claves de cada grupo de controles *Button* que conforman un campo de un formulario de una pantalla y lo análogo con un array con las claves de cada campo de entrada de texto del formulario de una pantalla. La idea es que un método con una implementación propia en cada calculadora específica será llamado por el método *cargarPantalla()* para que en función de la pantalla que se desea cargar cambie todos estos array mencionados. Lo importante es que todos estos array cambian completamente con cada pantalla que se visualiza y aunque siempre se conforman con constantes de la interfaz, en cada pantalla sólo tienen las constantes que tienen vistas asociadas en dicha pantalla. Sin embargo, el mapa que guarda de forma acumulativa todos los datos médicos introducidos hasta el momento en las diferentes pantallas se mantiene invariante a lo largo de todas ellas.

Se implementa un método llamado *cargarValoresGuardadosPantalla()* que sirve para cargar en el formulario de una pantalla los valores introducidos alguna vez anterior en dicha pantalla. El método sirve tanto para cargar los valores introducidos alguna vez en los campos del formulario de una pantalla anterior cuando se vuelve a esta, como para lo mismo pero al cargar los valores introducidos en esa pantalla para un cálculo anterior. Realmente existen dos mapas como el descrito en el párrafo anterior, es decir, que guardan los valores acumulados a lo largo de las pantallas. Uno es el descrito que se refiere al cálculo actual, que es el que se utiliza al retroceder a una pantalla, y el otro no es más que el duplicado del primero que se hace en el momento en el que se solicita al controlador realizar el cálculo a partir de todos los datos introducidos hasta el momento. Este segundo es el que se utiliza cuando, estando ya en una

pantalla (incluso con valores introducidos), se desean cargar todos los valores que se introdujeron en esa pantalla en un cálculo anterior. Se debe tener en cuenta que en el momento de realizar el cálculo el segundo mapa pasa a ser un duplicado del primero y el primero se vacía (inicializa) para volver a ir almacenando los valores que se vayan introduciendo para el nuevo cálculo (esto, realmente, aunque no se explica, por motivos de optimización, sólo es así conceptualmente). Este método entonces recibe como parámetros alguno de estos dos mapas según los valores que se deseen restablecer (retroceder a una pantalla del cálculo actual o cargar los valores de un cálculo anterior), el array con las claves de los grupos de controles *Button* y el array con las claves de los campos de introducción de texto. Es importante recordar que estos array cambian con cada pantalla por lo que se refieren cada vez sólo a la pantalla que se visualiza.

Entonces, aunque no se implementa en esta clase y esto es deber de la subclase, se declara un método abstracto llamado *obtenerReferenciasVistasPantalla()* cuya implementación será una estructura *switch* que llame a un método distinto para cada pantalla. En las calculadoras parciales, como sólo tienen una pantalla, en lugar de la estructura *switch* con una entrada *case* que llame a un método puede contener directamente la implementación de ese método que acabaría llamando siempre. ¿Para qué entonces un método por pantalla? Para una determinada pantalla, este método es el que declara inicializados y de forma local todos los array con claves de las vistas de la pantalla que se han mencionado antes (no son los array con claves asociadas al mapa que guarda acumulativamente los valores introducidos). De forma adicional, por cada array se crea otro con un orden coherente para los identificadores de cada vista a la que se refieren las claves del otro array. Resumiendo, se tienen seis array agrupados en parejas de dos. Las parejas son para las vistas *Button*, las vistas *EditText* y las vistas *TextView* propias de cada pantalla. Como para cada tipo existe un mapa que almacena cada vez para una pantalla concreta las referencias a todas las vistas de ese tipo, las claves de este mapa son las del primer array de cada pareja y se usan los identificadores correctamente ordenados del segundo array de cada pareja para obtener las referencias de cada vista de ese tipo. Esto se tiene por cada método que la estructura *switch* puede llamar. Cada uno de estos métodos con sus seis array cada uno, simplemente llama por cada tipo de vista a una vez método *obtenerReferenciasVistasGenerico()* con los argumentos correctos. La gracia de cada uno de estos métodos propios por pantalla que se pueden llamar desde la estructura *switch* es simplemente que es aquí donde se establece el valor diferente para cada pantalla de los array porque cada pantalla tiene diferentes vistas aunque sean siempre de los mismos tipos.

El método *obtenerReferenciasVistasGenerico()* cuenta con cuatro argumentos. Dos array de claves, un array de identificadores de vistas y un mapa para coleccionar referencias de vistas. Así pues, cada una de las tres veces que es llamado en cada método propio de cada pantalla llamado por la estructura *switch* recibe en orden el primer array de calves de cada pareja de array locales, el array análogo que es atributo de la clase, el segundo array de identificadores de vistas de cada pareja de array locales y el mapa que almacena referencias a las vistas del tipo para el que se ha llamado a este método (se le llama una vez por tipo de vistas). Con estos argumentos, este método sólo duplica el primer argumento en el segundo, y rellena el cuarto argumento con las claves del primer argumento y las referencias a vistas que obtiene a partir del tercer argumento. Para que este método funcione independientemente del tipo de vista para el que es llamado, se ha utilizado *Java Generics*. En las figuras Figura 69 y Figura 27 se muestra un ejemplo de implementación de algunos métodos que ayudan a comprender toda esta explicación.

```
private void obtenerReferenciasVistasPantallaDiagnostico()
{
    String[] clavesVistasBotones = {KEY_SELECCION_NUMERO_TUMORES_0, KEY_SELECCION_NUMERO_TUMORES_1, KEY_SELECCION_NUMERO_TUMORES_2, K...
            KEY_SELECCION_TUMOR_EXTENT_INF, KEY_SELECCION_TUMOR_EXTENT_SUP,
            KEY_SELECCION_PVI_SI, KEY_SELECCION_PVI_NO,
            KEY_SELECCION_NODES_SI, KEY_SELECCION_NODES_NO,
            KEY_SELECCION_METASTASIS_SI, KEY_SELECCION_METASTASIS_NO,
            KEY_SELECCION_PORTAL_HIPERTENSION_SI, KEY_SELECCION_PORTAL_HIPERTENSION_NO,
            KEY_SELECCION_PVT_SI, KEY_SELECCION_PVT_NO};

    int[] idsVistasBotones = {R.id.seleccionNumeroTumores0, R.id.seleccionNumeroTumores1, R.id.seleccionNumeroTumores2, R.id.seleccion...
            R.id.seleccionTumorExtentInf, R.id.seleccionTumorExtentSup,
            R.id.seleccionPVIsi, R.id.seleccionPVIno,
            R.id.seleccionNodesSi, R.id.seleccionNodesNo,
            R.id.seleccionMetastasisSi, R.id.seleccionMetastasisNo,
            R.id.seleccionPortalHipertensionSi, R.id.seleccionPortalHipertensionNo,
            R.id.seleccionPVTsi, R.id.seleccionPVTno};

    String[] clavesVistasCamposTexto = {KEY_ENTRADA_DATO_TUMOR1_TAMANO, KEY_ENTRADA_DATO_TUMOR2_TAMANO, KEY_ENTRADA_DATO_TUMOR3_TAMANO,...
    int[] idsVistasCamposTexto = {R.id.entradaDatoTumor1Tamano, R.id.entradaDatoTumor2Tamano, R.id.entradaDatoTumor3Tamano, R.id.entrad...

    String[] clavesVistasTextos = {};
    int[] idsVistasTextos = {};

    this.clavesVistasBotones = new String[clavesVistasBotones.length];
    this.clavesVistasCamposTexto = new String[clavesVistasCamposTexto.length];
    this.clavesVistasTextos = new String[clavesVistasTextos.length];

    obtenerReferenciasVistasGenerico(clavesVistasBotones, this.clavesVistasBotones, idsVistasBotones, vistasBotones);
    obtenerReferenciasVistasGenerico(clavesVistasCamposTexto, this.clavesVistasCamposTexto, idsVistasCamposTexto, vistasCamposTexto);
    obtenerReferenciasVistasGenerico(clavesVistasTextos, this.clavesVistasTextos, idsVistasTextos, vistasTextos);
}
```

Figura 26. Implementación para la pantalla diagnóstico de uno de los métodos llamados desde la estructura switch de la actividad de la calculadora completa.

```
private <T extends View> void obtenerReferenciasVistasGenerico(String[] clavesVistasSrc, String[] clavesVistasDest,
    int[] idsVistas, Map<String, T> coleccionVistas)
{
    int i;

    System.arraycopy(clavesVistasSrc, 0, clavesVistasDest, 0, clavesVistasSrc.length);

    for(i = 0; i < idsVistas.length; i++)
    {  coleccionVistas.put(clavesVistasSrc[i], ((T) findViewById(idsVistas[i])));  }
}
```

Figura 27. Implementación del método obtenerReferenciasVistasGenerico() de la clase de la calculadora genérica.

Además de todo esto y siguiendo un desarrollo análogo, se sigue la misma idea para los grupos de controles *Button* que conforman cada uno un campo del formulario de una pantalla. Se cuenta con un método propio por pantalla con el array de las claves que se van a utilizar para guardar el valor introducido por cada grupo de controles *Button* en el mapa correspondiente (el de los valores acumulativos del cálculo actual). No se debe confundir el array con las claves del mapa que guarda las referencias a todas las vistas de los controles *Button* que forman un grupo

con el array con las claves del mapa que guarda un sólo valor de todo ese grupo formado por varias vistas. Por ejemplo, en un grupo de controles formado por dos *Button* excluyentes para indicar si o no, el primer array tendría dos calves procedentes de este grupo (una por cada vista) y el segundo array tendría sólo una clave procedente del grupo en su conjunto (una clave por grupo de vistas *Button* en la pantalla actual). Aunque se explica más adelante su necesidad y uso, en este método se crea otro array de enteros que contiene la relación entre los dos array anteriores. Como el primero tiene seguidas en orden todas las claves de todas las vistas de tipo *Button* de la pantalla, este array de enteros realmente sólo indica cuáles de las claves del primer array forman un grupo y por ello se corresponde con una clave del segundo array. Este método propio de cada pantalla, como antes, tras declarar los array locales llama a un método genérico. Este método genérico aparte de copiar el array local de claves declarado en el método al correspondiente array que es atributo de la clase, rellena un mapa con la relación entre estas claves y el array de enteros, es decir, almacena en un mapa ambos array locales evitando tener que implementar lógica de vista adicional cada vez que se necesite utilizar la relación entre un grupo de controles *Button* y las vistas que lo forman. En el caso particular de los campos de introducción de texto es siempre una sola vista la que conforma el grupo, o lo que es lo mismo, no existen grupos porque la clave de la referencia a una de estas vistas se corresponde sólo con una clave del mapa que acumula los valores introducidos a lo largo de las pantallas en el cálculo actual. Por este motivo, aunque se podrían crear claves distintas para guardar las referencias a las vistas y claves distintas para guardar los valores introducidos en estas vistas, dado que en este caso esas hipotéticas claves distintas tienen siempre una relación una a una que es siempre la misma se ha decidido sólo crear un único *set* (conjunto) de claves y utilizarlo para rellenar los dos array de calves diferentes que son atributo de la clase. Este método genérico vuelve a copiar el hipotético array local que no se ha declarado porque el array atributo de la clase ya está correctamente asignado al otro array atributo de la clase aún sin asignar. No confundir que, puesto que ambos array tengan exactamente el mismo valor (las mismas claves), ambos no sean necesarios y no deba existir más que uno, porque cada cual tiene una función y será utilizado en distintos métodos. Esto es a nivel conceptual, a nivel de implementación sí podría ahorrarse uno de ellos.

Los primeros métodos creados para cada pantalla obtienen las referencias de todas las vistas de los tres tipos de cada pantalla y los segundos métodos creados para cada pantalla obtienen las relaciones entre grupos de vistas (o vistas cuando se trata de grupos compuestos por una sola vista) y el mapa que almacena todos los valores de los datos médicos del paciente introducidos a lo largo de las pantallas de la calculadora para el cálculo actual. Ambos grupos de métodos se podrían haber fusionado para tener sólo uno de cada par de métodos pero se ha decidido mantenerlos porque facilita bastante a la separación conceptual de conceptos y por ello a la claridad del código. En el segundo grupo de métodos propios de cada pantalla el método abstracto con la estructura *switch* se llama *establecerClavesValoresGuardadosPantalla()*.

El efecto de realizar de esta forma toda esta implementación descrita, a parte de la correcta estructuración y optimización del código es la gran flexibilidad que proporciona y sobre todo la posibilidad de adaptación a cambios futuros para que, con pocos cambios del código, se puede cambiar fácilmente el formulario de una pantalla añadiendo, editando o eliminando campos del formulario, y crear fácilmente nuevas pantallas si se necesitara. Todo esto, claro, siempre que se tratara de campos del formulario basados en los tres tipos básicos de vistas con los que se ha trabajado. Igualmente, como efecto de esta implementación se sabe que sin

excesiva complejidad se podría permitir el uso de nuevos tipos de vistas, aunque no es lo que se tuvo en mente explícitamente al pensar este diseño e implementación software.

Para implementar la acción que se debe llevar a cabo al detectarse un evento de pulsación sobre alguno de los controles *Button* de un grupo de vistas, se debe crear de nuevo un método propio por cada pantalla. Estos métodos serán los métodos de acción asociados con todas las vistas *Button* de una misma pantalla independientemente de los grupos a los que pertenezcan, es decir, usarán el atributo xml *onClick* para enlazarse a estos métodos que por ello deberán tener una firma de visibilidad *public*, retorno *void* y un único argumento del tipo *View*. Así pues, estos métodos que se implementan en las subclases, primeramente deben obtener del argumento el identificador de la vista *Button* que ha accionado al método que tiene asociado. Con este identificador llaman a un par de métodos de la clase genérica (esta) que tienen la lógica para deducir a partir del identificador de la vista y varios de todos los mapas y array explicados antes, cuál es el subarray con sólo las claves de las referencias a las vistas *Button* que conforman el grupo de controles al que el accionado pertenece. Después, también a partir del identificador de la vista recibida como argumento, con una estructura *switch* que tiene una entrada *case* por cada control *Button* que existe en el formulario de esa pantalla, se llama al método *configurarSeleccionBotones()* implementado en la clase general con los argumentos adecuados para cada caso de la estructura *switch*.

El método *configurarSeleccionBotones()* recibe cuatro argumentos, la clave del mapa acumulativo de valores introducidos asociado al grupo de controles al que pertenece la vista accionada, el valor que supone haber pulsado ese control en lugar de otro distinto del mismo grupo, la clave del mapa con las referencias a las vistas *Button* que corresponde con el control accionado y el subarray de las claves de las vistas de los controles que formar el grupo y que se ha deducido antes. A partir de estos argumentos guarda correctamente en el mapa de valores acumulativos introducidos el nuevo valor asociado al dato médico que representa ese grupo de controles y actualiza la apariencia de todas las vistas de los controles que forman el grupo dejando el control accionado con un aspecto diferente. Aunque no se describe cómo, realmente por temas de optimización, existe la lógica necesaria que permite detectar cuál era hasta ahora el control accionado del grupo (si es que lo había) para sólo devolver este al mismo aspecto que el resto de controles no accionados. Si el grupo se conforma por veinte controles esto provoca que sólo se actualice el aspecto de un máximo de dos controles en lugar de tener que actualizar el aspecto de todos los controles que forman el grupo.

Vista toda la lógica de vista que controla el comportamiento de los formularios de cada pantalla, se menciona que para la capa oculta de ajustes se ha replicado toda esta lógica adaptada (realmente se ha creado una análoga basada en la otra, no se ha duplicado) al formulario concreto de los ajustes (que es muchísimo más simple y corto) teniendo en cuenta que este dentro de una calculadora no cambia nunca a través de las diferentes pantallas. También se ha tenido en cuenta que este formulario sí puede cambiar en las diferentes calculadoras, y por ello, y por mejora de la claridad del código, se ha mantenido separada de la lógica de vista que controla los formularios de las pantallas.

A la hora de solicitar al controlador realizar los cálculos a partir de los datos médicos introducidos, la clase general implementa un método *realizarCalculo()* que junta (*append*) dos mapas, aquel con los valores acumulativos de datos médicos introducidos a lo largo de las pantallas anteriores y el análogo con los valores de los ajustes introducidos, para pasar el mapa resultante como argumento de la solicitud de cálculo al controlador. Como resultado de esa solicitud se obtiene o el mapa de resultados calculados o un error. Se debe distinguir que en los campos de introducción de texto le corresponde a la lógica de vista comprobar el tipo primitivo de dato que se ha introducido y la falta de valor introducido si se trata de un campo requerido entre otros, y le corresponde a la lógica de negocio comprobar si el valor introducido está dentro del rango permitido o es correcto para en caso contrario poder comunicar a través del controlador a la vista que esta debe informar al usuario y de qué estado concreto. Por ejemplo, para un campo requerido que espera un valor numérico entero, le corresponde a la vista comprobar que se ha introducido un valor numérico y que es de tipo entero. A la lógica de negocio le tocará comprobar si el valor introducido que ya se sabe que es numérico y entero vale menos que cierto valor, si es distinto de cero porque se utiliza en un cociente, o las condiciones pertinentes. Después, este método llama a otro abstracto de nombre *cargarResultadosAlgoritmos()* que se encargará de presentar ese mapa de resultados recibido al usuario de forma visual.

El método *cargarResultadosAlgoritmos()*, al estilo de lo anterior, cuenta con dos array locales en orden coherente para relacionar las claves del mapa de resultados de los cálculos con las calves del mapa con referencias a las vistas *TextView* en las que se van a mostrar los resultados. Este método es abstracto porque cada calculadora va a mostrar el resultado de un número diferente de algoritmos. Sin embargo, como la lógica de representación de los resultados obtenidos independientemente de lo anterior es siempre la misma, esta se ha implementado en otro método de la clase general el cual se espera que el método abstracto llame pasando los array locales que el método abstracto declara en cada una de sus implementaciones distintas que tenga en las subclases.

Respecto a la validación, se observa que utilizando adecuadamente el atributo xml *inputType*, como los datos médicos que se introducen son siempre valores numéricos, la lógica de vista sólo debe comprobar que en los campos requeridos se haya introducido un valor y que este es de tipo entero o decimal. Respecto a la validación de los controles *Button*, no debe realizarse ninguna validación porque cada control tiene mapeado un valor entero. De hecho, el que un campo de un formulario fuera requerido o no, no tendría que ser función de la lógica de vista que no sabe para qué es cada campo. Se debería encargar de ello la lógica de negocio y notificarlo al usuario en el momento de solicitar que se realicen los cálculos. Como en el caso concreto de la calculadora completa, sólo los campos de formulario que se refieren al número y extensión de los tumores son los únicos requeridos en caso de estar presentes en la pantalla, y para las calculadoras parciales todos los campos son obligatorios independientemente del tipo, se ha podido implementar la lógica de vista de validación en la clase general (aunque desempeñe comprobaciones que no son de su incumbencia). Esto se ha implementado así para que la calculadora completa, que cuenta con varias pantallas, no deba esperar hasta pasar varias para que se notifique al usuario que un campo requerido de una pantalla anterior carece de valor introducido.

La clase general también implementa un método para borrar todos los campos del formulario de la pantalla actual. Para ello restaura a su valor inicial o por defecto tanto el aspecto de todas las vistas como el valor de cada campo en el mapa de valores acumulativos introducidos a lo largo de las pantallas. Además, cuenta con un método que restaura el estado de todos los campos del formulario de una pantalla al del cálculo anterior realizado si existiera. Para ello, reutiliza parte de la lógica de vista del método anterior que borra los campos del formulario aunque con otro propósito (adaptada) y llama al método *cargarValoresGuardadosPantalla()*.

Como la capa oculta de ajustes admite que se introduzcan valores y se guarden, y se introduzcan valores, pero no se guarden (cerrar la capa sin guardar) también utiliza dos mapas que almacenan los valores que se van introduciendo en este. No está implementada la funcionalidad de cargar los valores de los ajustes para un cálculo anterior, pero de necesitarse, pasaría por usar un tercer mapa. La necesidad del segundo mapa temporal dada la opción de poder cerrar la capa sin guardar se ve justificada con que se pueden introducir los valores de los ajustes abriendo y guardando varias veces la capa por lo que podría ocurrir que al cerrar la capa no siempre hubiera que hacer volver a todos los campos a su valor inicial.

La capa de más información consiste generalmente en mostrar una imagen dependiendo de la pantalla en la que se está, dependiendo de la calculadora en la que se está, dependiendo del resultado obtenido de los algoritmos, etc., por lo que cerrarla no requiere ninguna acción más allá de establecer su visibilidad como oculta de nuevo. Sabiendo de antemano que va a ser siempre una imagen se podría evitar que su *layout* se inflara y limitarse sólo a ir cargando en su lugar la imagen dinámicamente (esto ya se hace ahora), pero se ha decidido implementar así (inflar además de cargar la imagen dinámicamente según una consulta al controlador) para dejar abierta la posibilidad de que en el futuro no sólo consistiera en mostrar una imagen. Así pues, la clase general cuenta con los métodos necesarios para implementar la lógica de vista necesaria que requieren estas funcionalidades con ambas capas ocultas.

La clase general implementa un método para cargar la siguiente pantalla. Este método sólo llama al de validación de los campos del formulario de la pantalla que se ha visto antes, después llama al método *guardarValoresCamposTexto()* y finalmente al método *cargarPantalla()*. El método *guardarValoresCamposTexto()* se encarga de a partir de los array y mapas descritos en este apartado extraer el texto introducido en las vistas *EditText*, convertirlo al tipo numérico decimal (si ha pasado la validación es porque son convertibles a este tipo) y almacenar los valores en el mapa acumulativo de valores introducidos a lo largo de las pantallas. Respecto al argumento que requiere el método *cargarPantalla()* este ya se ha establecido en dicho método la última vez que se llamó. La primera vez que se llama a ese método no ha habido una vez anterior, por eso el valor inicial del argumento que necesita se establece en el método *onCreate()* antes de que sea llamado por primera vez.

Como ya se ha indicado antes, de nuevo, parte de la última lógica de vista explicada se ha replicado adaptada (se ha creado una análoga basada en la otra, no se ha duplicado) para la capa oculta de ajustes. Por ello, el método análogo al descrito antes para cargar una siguiente pantalla que en este caso se activa cuando el usuario desea guardar los valores de ajustes introducidos, realiza en orden la validación de los valores, el guardado de los valores y ocultar la capa de ajustes. Todo esto se menciona para explicar que justo antes de ocultar la capa de ajustes llama al método que actualiza las etiquetas de texto (*TextView*) con las unidades de los

campos del formulario de la pantalla que se está visualizando en ese momento. Las unidades se actualizan sólo si el campo del formulario de los ajustes que se refiere a las unidades internacionales ha cambiado. Si tiene el mismo valor no se realiza ningún cambio, pero si el valor es distinto se actualizan consecuentemente todas las unidades correctamente.

Volviendo a un razonamiento anterior (que la vista no debe conocer qué campos son obligatorios), la lógica de vista que hace estos cambios de unidades no debería por qué comprender qué significa cada unidad para poder deducir la relación presente entre cada par de unidades (unidad internacional y unidad no internacional) que pueda existir entre todos los campos de los formularios, siendo esta tarea algo que debiera recibir del controlador. Aunque así sea, sí se le pueden dar todos los pares de relaciones entre unidades ya deducidos porque esto no implica que deba comprender el significado de cada unidad. La lógica de vista de esta forma no debe realizar deducciones porque ya parte de tener estas hechas. Por ello, no es su tarea deducir las relaciones, pero si lo es, una vez conocidas estas, saber gestionar los cambios de unidades correctamente a partir de estas relaciones dadas. Bien, lo que queda por decidir es si estas relaciones las pide al controlador o si dado que son relaciones fijas y que no pueden cambiar más adelante se integran en una matriz de enteros directamente en la lógica de vista. La decisión de diseño tomada es quedarse con la segunda opción de las dos anteriores.

Sin embargo, no se ha podido aplicar la idea explicada hasta ahora tan estrictamente porque para el caso concreto de HepApp existe la particularidad de que al menos una unidad internacional se corresponde con dos o más unidades no internacionales diferentes. Esta situación fuerza a que la lógica de vista rompa el principio anterior pues implica que esta acabe conociendo el significado de alguna unidad para poder distinguir cuando una unidad internacional debe corresponderse con una u otra unidad no internacional. En el momento de la implementación esta violación de independencia de capas se reduce porque en lugar de hacer tener que comprender a la vista el significado de algunas unidades realmente se la dice qué campos concretos (en este caso sólo son dos) deben hacer la correspondencia de la unidad internacional con una unidad no internacional *b*, y que para el resto de campos la unidad internacional siempre se corresponde con la unidad no internacional *a*. De esta forma no acaba teniendo que comprender el significado de las unidades, pero sí se la hace conocer alguna relación entre unidades y campos concretos.

De nuevo, que estas relaciones entre unidades y campos se las de, puesto que no supone que tenga que deducirlas, no implica tener que comprender ni el significado de las unidades ni el de algunos campos. Esto soluciona el problema, pero complica el diseño e implementación del sistema. Se podría haber evitado si directamente se tomaran las relaciones pidiéndoselas al controlador. ¿Por qué no se ha hecho esto desde un principio? El único motivo es el de que las cadenas de texto con las unidades son recursos *string* en el fichero *strings.xml*. El diseño de Android está pensado para que siempre que se pueda sea mejor tomar este tipo de recursos desde una actividad, y las actividades sólo se usan en la capa de la vista del sistema. Aunque pasando un objeto de tipo *Context* se puede lograr acceder a estos recursos (y a otros) en otras capas desde clases que no son actividades, si esto se puede evitar y realizarse en actividades mucho mejor. El motivo por el que los textos de las unidades están declarados como recursos *string* es el de aprovechar el mecanismo de Android para crear directorios de recursos alternativos, en este caso de cara a las traducciones y el soporte de múltiples idiomas. Es cierto que en este caso concreto se trata de unidades que seguramente sean numéricas o cuenten con una abreviatura que pueda ser común para los diferentes idiomas, pero, aunque no es este caso, si en castellano se denota una unidad como centímetro y en inglés la misma unidad se denota

como centimeter, puesto que a priori se desconoce cuándo y con qué idioma concreto al que se pudiera dar futuro soporte una unidad va a tener una representación textual diferente en función del idioma se pensó que esta opción era la mejor a pesar de los inconvenientes que plantea. Evidentemente se ha podido aplicar porque en este caso sólo son dos las relaciones entre unidades y campos las que se deben dar deducidas de antemano, de otro modo este planteamiento podría no ser escalable.

La clase general cuenta con un método *cargarAjustesCalculadora()* que restaura a partir del mapa de valores que guarda los ajustes introducidos, y el estado y aspecto de las vistas del formulario de ajustes. Se ha dicho que se conoce que este formulario de ajustes puede ser distinto en cada calculadora, algo que en principio impediría la implementación completa de este método en la clase general. Lo que permite realizar la implementación de este método es que, aunque lo anterior sea cierto, se conoce que en otras calculadoras el formulario de la capa de ajustes de ser diferente es simplemente porque con respecto al formulario de ajustes de la calculadora completa, en las parciales sólo se eliminarían campos de este formulario completo. Así pues, este método actúa suponiendo de partida que siempre se tiene el formulario completo inflado por lo que intentará restaurar el estado y aspecto de todas las vistas. Para saber si un campo del formulario completo no está presente (caso del formulario reducido de una calculadora parcial) y que por lo tanto debe omitir intentar restaurar el estado y aspecto de las vistas asociadas lo que hace es comprobar en el mapa correspondiente con referencias a estas vistas si este contiene una entrada para la clave de la vista cuya referencia pretende tomar de dicho mapa para utilizarla. Si no existe el campo del formulario, este mapa no contendrá entradas para estas claves.

Sobrescribir el método *onBackPressed()* no tendría sentido hacerlo en esta clase, pero sí en alguna subclase que implemente una calculadora con varias pantallas en la que, si se desea, se pueda implementar la lógica de vista necesaria que permita establecer el comportamiento del botón atrás para retroceder desde una pantalla concreta de la calculadora a otra diferente concreta en lugar de ejecutar la acción predeterminada para ese botón que implicaría retroceder a una actividad anterior según la pila de actividades, o lo que es lo mismo, implicaría cerrar la actividad de la calculadora.

El control tipo *Button* que sirve para mostrar la capa oculta de ajustes es el control que en la actividad común se encuentra por defecto oculto (barra superior) y se reservaba para futuros usos. Como el aspecto de este control es similar para todas las calculadoras es en esta clase (que es subclase de la actividad común) en la que se implementa el método abstracto que configura el recurso *drawable* con el icono de la rueda dentada de ajustes para ese control. De esta forma, dado que este control cuenta en la actividad común con un método que permite fácilmente cambiar entre mostrarlo y ocultarlo, se puede cambiar su visibilidad bajo demanda en cada pantalla distinta de cada calculadora.

El control siempre que sea accionado debe llevar a cabo la misma tarea, por lo que se implementa también en la clase abstracta el método *accionBotonExtra()* que simplemente llamará a otros métodos propios de las calculadoras (que ya se han explicado) para mostrar la capa de ajustes y con los valores introducidos anteriormente si es que los hubiera.

Para implementar la capa oculta de ajustes, aunque no se han utilizado, se ha sido en su momento consciente de la existencia y posibilidad del uso de actividades de preferencias y la API de preferencias. No se han utilizado por las necesidades de diseño (visual) de la interfaz gráfica de la aplicación, pues en esta capa en vez de mostrarse con un menú y una actividad propia para preferencias se tenía que mostrar como un *popup* y con un icono concreto para el control de la barra superior. A parte, se puede interpretar que el uso pensado para las preferencias (ajustes generales de la aplicación, no sólo de la calculadora) es otro distinto al uso que se las hubiera dado para implementar la capa oculta de ajustes sólo en las calculadoras. También, dado que el funcionamiento de la capa de ajustes se ha basado en una lógica que se tendría que haber implementado igualmente para las calculadoras, su implementación actual no ha supuesto apenas un sobresfuerzo adicional.

Posteriormente al diseño e implementación de las calculadoras se cayó en la cuenta de que para las calculadoras con varias pantallas se podrían haber usado fragmentos y FragmentActivity para implementar mejor el uso de diversas pantallas en una misma actividad. El motivo de no haberse dado cuenta de esto es la falta de familiarización y costumbre de uso de estos elementos por parte del desarrollador. Puesto que esto fue una consideración a posteriori, que sólo había una calculadora con varias pantallas y que finalmente, aunque fuera una mejor práctica, como tampoco simplificaría tantísimo la implementación que ya se tenía, se decidió no tomarse las molestias de aplicarlo dejándolo como una tarea pendiente de realizar en un futuro.

8.1.5.2 Actividad calculadora completa

Esta actividad implementa la calculadora completa que permite la entrada de todos los posibles datos médicos de un paciente para calcular todos los posibles algoritmos si se han completado todos los campos de todos los formularios. Hereda de la actividad calculadora general. Esta calculadora es la única que va a contar con varias pantallas.

Esta actividad sobrescribe el método *onCreate()*, aparte de para especificar la pantalla inicial que debe cargarse, para llamar al método de la superclase que va a habilitar o mostrar el control extra de la barra superior (control de la capa oculta ajustes).

Se sobrescribe el método *setContentView()* para inflar el *RelativeLayout* de la capa oculta ajustes de los que en la superclase se han dejado vacíos. También realiza una llamada al método *cargarPantalla()* esta vez sí implementado en la clase y que se encargará de inflar los otros tres *RelativeLayout* en cada nueva pantalla que se cargue.

El método *cargarPantalla()* recibe como argumento un *Enum*. Para cada calculadora con varias pantallas existe un *Enum* propio que lista todas las pantallas (una entrada por pantalla posible) que puede tener esa calculadora. Las instancias (*Singleton*) de este tipo *Enum* tienen como atributos de clase, el identificador del recurso, de un recurso *string* del título de la pantalla y los tres identificadores (cada uno como un atributo de clase) del fichero de recursos xml con

las disposiciones de vistas (*layout*) que se van a usar en cada pantalla para inflar los tres *RelativeLayout* que se han quedado vacíos. Así pues, este método cuenta con una estructura *switch* en función del *Enum* y cuenta con un *case* por cada elemento del *Enum*, es decir, por cada pantalla. En el *case* de cada pantalla que sí deba tener una siguiente pantalla se cambia el valor del atributo de la clase de la calculadora del tipo *Enum* que almacena la siguiente pantalla a cargar al de dicha siguiente pantalla. En las pantallas que no deban tener una siguiente pantalla esto no se hace. En ese método, antes de la estructura *switch* a partir del *Enum* recibido como argumento, aunque aún no haya pasado por la estructura *switch* para saber a qué pantalla se refiere, se usan los métodos *getter* del *Enum* para obtener los identificadores de los *layout* xml y utilizarlos para inflar los *RelativeLayout* usando los métodos que tiene implementados la superclase. Esta parte se ha colocado antes de la estructura *switch* porque es un código común a todas las pantallas y no debe repetirse en cada *case*. Sin embargo, en cada *case* sí se deben llamar directamente a los métodos propios de cada pantalla que obtienen las referencias a las vistas y que establecen las relaciones entre los grupos de controles *Button* y las claves con las que se guardan los valores en el mapa. A fin de cuentas, llamar a los métodos *obtenerReferenciasVistasPantalla()* y *establecerClavesValoresGuardadosPantalla()* que en la superclase eran abstractos y ahora cuentan con una implementación. En el momento de la implementación, como en el caso de haber varias pantallas cada uno de esos métodos volvería a implementar la misma estructura *switch* ya presente en *cargarPantalla()* se ha decidido llamar directamente a los métodos desde los *case* de la estructura *switch* de *cargarPantalla()* e implementar esos métodos abstractos como vacíos (sólo retornan). Estos métodos abstractos de la superclase siguen teniendo sentido enfocados a aquellas calculadoras que sólo tendrán una única pantalla y porque son convenientes a nivel conceptual para la comprensión de la superclase.

Esto es lo que se hace con las tres primeras pantallas de esta calculadora que son en las que se introducen datos médicos del paciente en los formularios. La calculadora tiene aún otras tres pantallas más. Una en la que se muestran los resultados (*RESULTADOS*), otra en la que se muestra un resumen de los datos médicos introducidos (*RESUMEN*) y otra en la que se muestra el esquema de *Alberta* según el cálculo realizado (*ALBERTA*). La última de las tres pantallas con formulario marca como siguiente pantalla la que muestra los resultados, pero estas tres últimas pantallas nombradas en este párrafo al cargarse no marcan ninguna pantalla como siguiente. Esto se debe porque la pantalla de resultados tiene un botón exclusivo para cargar a las otras dos restantes y estas dos restantes son pantallas finales que no tienen una pantalla siguiente y sólo pueden regresar a la pantalla de resultados. Sin embargo, la pantalla de resultados también tiene un botón exclusivo a cada pantalla con formulario para poder retroceder por si se quisiera cambiar de valor un dato médico antes introducido. En caso de retroceder a una pantalla de formulario, desde estas no es posible volver directamente a la pantalla de resultados y se debe pasar por todas las pantallas con formulario que queden. En el case de la pantalla resultados se llama al método *realizarCalculo()* para lograr que los cálculos se realicen automáticamente ya que esta calculadora carece de un botón propio para solicitar que se hagan los cálculos. Al retroceder desde las pantallas *RESUMEN* y *ALBERTA* no se vuelve a llamar al método *realizarCalculo()* porque desde estas pantallas no se han podido cambiar ninguno de los valores de los datos médicos introducidos ya que carecen de formularios. Sin embargo, cuando se retrocede desde la pantalla de resultados a una de las que tienen formularios, al volver a la pantalla de resultados sí se llama de nuevo a *realizarCalculo()* pues podría haberse modificado algún dato médico introducido. Se debe pensar que no tiene sentido retroceder a una pantalla

con formulario sólo para observar los valores introducidos porque ese es exclusivamente el cometido de la pantalla *RESUMEN*.

En el *case* de la pantalla *RESUMEN* se llama a un método *cargarResumenValores()* exclusivo de esta calculadora (no está en la superclase ni implementado ni declarado como abstracto) que se encarga de cargar en las vistas *TextView* todos los datos médicos introducidos a partir del mapa de la superclase en el que se tienen todos estos valores almacenados. Este método también va tomando sobre la marcha las referencias a los *TextView* porque a diferencia lo que ocurre en las pantallas con formulario estas referencias son de un único uso porque no se van a necesitar utilizar de nuevo y de hecho no son almacenadas.

Se debe observar que cada vez que se carga una pantalla diferente al menos se vuelve a inflar el *RelativeLayout* en el que se infla el formulario de la pantalla y por ello las referencias a las vistas deben volver a tomarse quedando invalidas las que se hayan tomado en veces anteriores. También considerar que ese *RelativeLayout* que se supone que sería sólo para formularios se puede utilizar para lo que se quiera como es el caso de las pantallas de resultados, *RESUMEN* y *ALBERTA* en las que no hay formularios.

El *case* de la pantalla *ALBERTA* llama a un método *cargarEsquemaAlberta()* que solicita al controlador cuál es el esquema de *Alberta* que debe cargar y después lo carga en la vista *ImageView* del *layout* que esta pantalla infla.

Se debe caer en la cuenta de que la pantalla resultados permite ir a cualquiera de todas las demás pantallas. También, los valores de los identificadores de los *layout* almacenados en los atributos de clase del *Enum* de las pantallas de esta calculadora (recordar que es un *Enum* distinto por cada calculadora con varias pantallas) pueden ser distintos porque se refieran a diferentes *layout*. Esto siempre ocurre con el *layout* principal de cada pantalla, pero se debe tener en cuenta que también puede ocurrir para la barra inferior y la capa oculta de más información. Respecto a la barra inferior, las pantallas *RESUMEN* y *ALBERTA* no tienen y por ello el valor de este atributo es *-1* que indica que en vez de vaciar e inflar sólo se debe vaciar el *RelativeLayout* de la barra inferior (siempre hay que vaciar antes de volver a inflar porque podría estar ya inflado de alguna pantalla anterior). La pantalla de resultados como se ha dicho tiene su propio *layout* para la barra inferior que consiste en un botón para cada una de las pantallas con formulario. Todos estos botones de la barra inferior llaman a un mismo método con el atributo *onClick* de xml y este a partir de la vista que recibe como argumento (con una estructura *switch*) detecta qué botón se ha pulsado y en función de eso deduce con qué argumento debe llamar al método *cargarPantalla()*. En el caso de las pantallas con formulario todas estas tienen la barra inferior común con cuatro controles *Button*. De nuevo todos estos controles tienen asociados un mismo método con el atributo xml *onClick* que según la vista que recibe como argumento (con una estructura *switch*) deduce cuál de los botones ha sido pulsado para llevar a cabo la acción pertinente. Uno de estos controles provocará que se llame al método *cargarPantalla()* para pasar a la siguiente pantalla, otro hará que se muestre la capa de más información de esa pantalla (puede ser distinta en cada pantalla), otro servirá para llamar al método de la superclase que restaura los valores introducidos sólo en el formulario de esa pantalla para un cálculo anterior y el último para llamar al método de la superclase que borra todos los valores introducidos en el formulario de sólo esa pantalla.

Respecto a la capa oculta de más información esta se establece a partir del layout (que tiene una vista *ImageView*) cuyo identificador está en el atributo del *Enum*. Como caso distinto, en la pantalla de resultados la capa oculta de más información muestra una imagen u otra

dependiendo de los resultados del cálculo realizado por lo que el *layout* del *Enum* para esa pantalla infla la vista *ImageView* pero sin una imagen predeterminada establecida como ocurre en el resto de pantallas (las pantallas *RESUMEN* y *ALBERTA* aparte de no tener barra inferior tampoco tienen capa oculta de más información). Así pues, esta calculadora implementa un método exclusivo llamado *establecerImagenCapaMasInformacion()* que solicita al controlador cuál es esa imagen que debe mostrar cada vez en la pantalla resultados según los resultados de los cálculos realizados. Para ello se ha implementado otro *Enum* que representa las posibles imágenes que se pueden mostrar en la capa oculta más información para esta pantalla concreta. Lo que devuelve el controlador como resultado a la consulta que se le ha hecho es un elemento de este *Enum*. Este *Enum* tiene un atributo de clase que almacena el identificador de un *drawable* que tiene la imagen correspondiente. Esto se ha implementado de esta forma para que el controlador no tenga que devolver directamente el identificador del recurso *drawable* puesto que como ocurría con la clase *Context*, aunque se puede implementar no es para nada una buena práctica ni el uso pensado para la clase *R*. Con el valor del *Enum* se usa un método *getter* en la actividad de la calculadora, se obtiene la referencia de la vista *ImageView* sin imagen asignada y se usa el método *setImageResource()* para establecer la imagen de la capa oculta más información.

También en la estructura *switch* del método *cargarPantalla()* en los *case* de las pantallas de resultados, *RESUMEN* y *ALBERTA* se debe llamar al método de la superclase que sirve para establecer la visibilidad del control reservado para futuros usos de la barra superior que en este caso se ha utilizado para mostrar la capa oculta ajustes pues en estas pantallas no se desea que se pueda mostrar la capa de ajustes.

La clase también debe implementar todos los grupos de métodos propios de cada pantalla para obtener vistas y establecer relaciones entre campos de los formularios (formados por grupos de vistas) y las claves utilizadas en el mapa que guarda los valores médicos introducidos. Hasta ahora en esta clase sólo se había dicho cuándo se llaman (en *cargarPantalla()*). Donde realmente ya se ha explicado cómo deben implementarse es en el subapartado *Actividad calculadora general*.

También debe implementar esta clase el grupo de métodos cada uno propio de una pantalla que va a gestionar la acción que se debe llevar a cabo cada vez que se acciona un control *Button* de todos los presentes en el formulario de una pantalla. También se ha descrito cómo debe ser su implementación en el subapartado *Actividad calculadora general*.

En el caso del método *cargarEsquemaAlberta()* antes mencionado pasa como en el método *establecerImagenCapaMasInformacion()*.

Dado que en la pantalla de resultados se muestran también si están disponibles hasta dos tratamientos para el paciente que se deducen de los resultados obtenidos, esta calculadora sobrescribe el método *realizarCalculo()* para que después de que se haga lo que hace el de la superclase de igual nombre se implemente esta funcionalidad. Los tratamientos se obtienen del controlador como cualquier otro resultado de cálculo de cualquiera de los algoritmos.

Precisamente el método *realizarCalculo()* en la superclase llamaba a un método *cargarResultadosAlgoritmos()* de esta abstracto que se debe implementar en esta calculadora. En el subapartado *Actividad calculadora general* ya se ha indicado cómo debe ser la implementación del método. Lo que no se explica es cómo son esos dos array locales. Uno es un

array formado por algunas de las constantes de la interfaz que implementa la superclase. Estas constantes son las calves utilizadas en el mapa que almacena las referencias de las vistas *TextView* en cada pantalla. El otro array está formado por las distintas entradas de un *Enum* que sólo lista los tipos de algoritmos con los que cuenta el sistema. Esta vez, la clase *Enum* no cuenta con atributos de clase, por lo que es un *Enum* en si mínima expresión. Por ello, como en veces anteriores, estos dos array locales son de la misma longitud y van en orden coherente para declarar la relación entre cada elemento de ambos array, es decir, el orden de los array indica qué *TextView* debe mostrar el resultado de cada algoritmo. Como ya se dijo (con las unidades de algunos campos de los formularios), de esta forma a la vista sólo se la están dando las relaciones ya deducidas y por ello sigue sin necesitar conocer detalles propios de la lógica de negocio, aunque esté el controlador actuando como intermediario entre ambos.

Finalmente, esta calculadora implementa el método *onBackPressed()* para que, según lo que se ha dicho en el subapartado *Actividad calculadora general* al respecto, aporte la lógica para que pulsar el botón atrás desde las pantallas *RESUMEN* y *ALBERTA* suponga volver a la pantalla de resultados en vez de retroceder en la pila de actividades que es el comportamiento por defecto y el que se realiza desde el resto de pantallas de la calculadora.

Se debe mencionar que se crea otro *Enum* que lista todas las calculadoras disponibles en la aplicación, es decir, la completa y las otras cuatro parciales. Ese *Enum* cuenta con un atributo de clase para almacenar el identificador del recurso *string* xml que es el título de la calculadora. A este título, si la calculadora se compone de varias pantallas se le va concatenando al final (*append*) el título de la pantalla concreta en cada caso.

8.1.5.3 Actividad calculadora parcial general

Esta actividad genérica por lo tanto es una clase abstracta y hereda de la actividad calculadora general. Lo que pretende esta clase es particularizar algunas cosas de su superclase teniendo en mente las calculadoras parciales que sólo tienen una única pantalla. De esta forma se simplifica y acorta la implementación de las cuatro calculadoras parciales.

Esta actividad sobrescribe el método *onCreate()*, para llamar al método de la superclase que va a habilitar o mostrar el control extra de la barra superior (control de la capa oculta ajustes). Esto se debe a que en todas las calculadoras parciales se va poder mostrar la capa oculta ajustes.

Se sobrescribe el método *setContentView()* para inflar el *RelativeLayout* de la capa oculta ajustes de los que en la superclase se han dejado vacíos. Esto se debe a que todas las calculadoras parciales van a utilizar el mismo formulario reducido para la capa oculta ajustes. También realiza una llamada al método *cargarPantalla()* esta vez sí implementado en esta clase.

En esta ocasión el método *cargarPantalla()* primeramente llama a un nuevo método abstracto llamado *cargarTituloCalculadoraParcial()* cuya implementación debe ser distinta en cada calculadora parcial porque debe cargar un recurso xml *string* distinto. Después, se llama a otro nuevo método abstracto *cargarDisposicionCalculadoraParcial()* que como el anterior su

implementación será distinta en cada calculadora parcial sólo porque debe utilizar cada vez un recurso xml de disposición de vistas (*layout*) diferente. A continuación, infla los *RelativeLayout* de la barra inferior y de la capa oculta más información. La barra inferior en este caso tendrá todos los controles descritos en la calculadora completa para las pantallas con formulario, es decir, el control para borrar el formulario, cargar los valores anteriores y mostrar la capa oculta de más información. Por ello esta clase implementa todos los métodos de acción asociados a esos botones como ya se ha descrito en el apartado *Actividad calculadora completa*. Respecto al *layout* que infla la capa oculta de más información se establece una vista *ImageView* sin una imagen asociada.

Finalmente, el método también llama a los métodos abstractos *obtenerReferenciasVistasPantalla()* y *establecerClavesValoresGuardadosPantalla()* declarados en la superclase que seguirán sin ser implementados en esta.

La clase implementa el método con firma *public void seleccionBotonGrupoBotonesPantalla(View vista)* que equivaldrá al método que en otras calculadoras de varias pantallas había que implementar por cada pantalla para que gestione las acciones de pulsación sobre los controles *Button* del formulario de dichas pantallas. Así pues, la implementación de este método, como se hacía en el subapartado *Actividad calculadora general*, obtiene el identificador de la vista *Button* del control accionado y también llamando a otros métodos de la superclase el array con las claves que usa el mapa que guarda las referencias de las vistas *Button* para el grupo de controles al que pertenece el control accionado. Tras realizar esto llama a un nuevo método abstracto con firma *protected void configurarBotonSeleccionado(int idVista, String[] listaKeyBotonesGrupo)* que básicamente implementará la estructura *switch* con un *case* por cada vista *Button* que haya en el formulario para determinar la acción de cada una de estas vistas.

También sobrescribe el método *realizarCalculo()* de la superclase para que antes de llevar a cabo las acciones que hace la implementación de este mismo método de la superclase haga sólo parte de las tareas que hacía el método *cargarSiguientePantalla()* de la superclase. Esto se debe a que en la calculadora completa se iba haciendo la validación que hace la vista en cada pantalla antes de llamar a la siguiente de forma que cuando se llamaba a *realizarCalculo()* automáticamente ya se había validado antes todo por parte de la vista. En las calculadoras parciales se cuenta con un botón exclusivo para llamar a *realizarCalculo()* lo cual obliga a la implementación que se acaba de describir.

Como ocurría en la calculadora completa con la pantalla de resultados, se declara un método abstracto *establecerImagenCapaMasInformacion()* que servirá para establecer en cada calculadora en la vista *ImageView* inflada antes el recurso *drawable* que podría ser distinto en cada calculadora y por ello que este método vaya a tener una implementación diferente en cada calculadora parcial. Este método también es llamado desde *cargarPantalla()* tras inflar el *layout* que tiene la vista *ImageView*. Esta implementación presenta el inconveniente de que asume que cada calculadora parcial sólo va a tener siempre máximo una misma imagen para la capa de más información. Sería incompatible entonces con un caso como el de la pantalla de resultados de la calculadora completa. Se era consciente de esto al realizar la implementación y se decidió hacerlo así porque se consideró que no se iba a poder dar un caso como ese en una calculadora parcial.

Cualquier otro método declarado como abstracto en la superclase que no se haya mencionado hasta el momento no tendría su implementación en esta clase por lo que quedaría ahora pendiente su implementación para las subclases de esta clase.

8.1.5.4 Actividad calculadora parcial CLIP

Esta es la actividad que implementa una de las calculadoras parciales. Hereda por ello de la actividad calculadora parcial general. Por regla general, las calculadoras parciales sólo tendrán que implementar los métodos abstractos nuevos que se han declarado en la superclase y aquellos abstractos de la superclase de la superclase que la superclase no ha implementado. A parte, se implementarán los métodos exclusivos que cada calculadora particular necesitara. Las calculadoras parciales también tendrán que crear sus ficheros de recurso xml con el *layout* del formulario de la única pantalla. El aspecto general de las calculadoras parciales es similar y sus formularios se componen reutilizando código de los ficheros xml *layout* de las pantallas de la calculadora completa.

En este apartado sólo se explicarán los detalles propios y exclusivos de esta calculadora parcial porque lo demás ya se ha explicado en subapartados anteriores.

Como caso particular, esta calculadora debe tener un campo en su formulario que es nuevo y no aparece en ninguno de los formularios de las pantallas de la calculadora completa. Esto se debe a que el campo sirve para introducir directamente el resultado de un algoritmo, concretamente el algoritmo *CPS*, en lugar de servir para introducir un dato médico como ocurría siempre hasta ahora en todos los campos de los formularios de las calculadoras. Alternativamente, se podrían haber puesto en el formulario todos aquellos campos de datos médicos que se necesitan y no estuvieran ya para calcular el algoritmo cuyo resultado se introduce directamente a través del formulario en esta calculadora.

Con esta particularidad se debe implementar un método exclusivo de esta calculadora para gestionar sólo las acciones que debe desempeñar cada control del grupo de controles *Button* que forman este campo del formulario. También se debe tener en cuenta esto para la implementación de los métodos *obtenerReferenciasVistasPantalla()* y *establecerClavesValoresGuardadosPantalla()* porque aunque su tipo de implementación no va a variar, los array locales que estos métodos declaran tendrán nuevas constantes no utilizadas aún hasta el momento.

8.1.5.5 Actividad calculadora parcial CPS

Esta es la actividad que implementa una de las calculadoras parciales. Teniendo en cuenta lo que se ha explicado en el subapartado *Actividad calculadora parcial CLIP* como regla general para las calculadoras parciales, dado que esta calculadora parcial no tiene ningún elemento ni método exclusivo sólo tendrá que implementar los métodos ya explicados en

subapartados anteriores de la forma también ya explicada en estos, pero particularizados a esta calculadora.

8.1.5.6 Actividad calculadora parcial MELD

Esta es la actividad que implementa una de las calculadoras parciales. Teniendo en cuenta lo que se ha explicado en el subapartado *Actividad calculadora parcial CLIP* como regla general para las calculadoras parciales, dado que esta calculadora parcial no tiene ningún elemento ni método exclusivo sólo tendrá que implementar los métodos ya explicados en subapartados anteriores de la forma también ya explicada en estos, pero particularizados a esta calculadora.

8.1.5.7 Actividad calculadora parcial Okuda

Esta es la actividad que implementa una de las calculadoras parciales. Teniendo en cuenta lo que se ha explicado en el subapartado *Actividad calculadora parcial CLIP* como regla general para las calculadoras parciales, dado que esta calculadora parcial no tiene ningún elemento ni método exclusivo sólo tendrá que implementar los métodos ya explicados en subapartados anteriores de la forma también ya explicada en estos, pero particularizados a esta calculadora.

8.1.6 Otros

Aquí se van a tratar otros aspectos y consideraciones que no se han comentado sobre la vista en apartados anteriores o que si se ha hecho ha sido de pasada o no entrando demasiado en detalle.

8.1.6.1 PaintingView

La vista personalizada *PaintingView* extiende a *View* utilizando instancias de las clases *Paint*, *Path*, *Canvas* y *Bitmap* del paquete *android.graphics*. Utiliza *Paint* para definir el estilo, propiedades y características del trazado que se va a realizar sobre el lienzo *Canvas*. Se utiliza *Path* para almacenar el trazado realizado por el usuario del tipo configurado con *Paint*.

Entrando más a nivel de implementación, cuando se instancia esta vista se carga la configuración por defecto del trazado con el objeto *Paint*. Posteriormente, con el diálogo para elegir color sólo se cambia la propiedad color del objeto *Paint*. El resto de sus propiedades no son configurables para el usuario y se quedan como se han configurado inicialmente al instanciar la vista.

Con el método *onDraw()* se aplica (pinta) sobre el lienzo (*Canvas*) el trazo (*Path*) que se ha creado durante los eventos táctiles producidos por el usuario con *drawPath()*. El método

onTouchEvent() se encarga de manejar los eventos táctiles del usuario que sirven para conformar el trazo (*Path*). En este método se implementa el comportamiento para los tres eventos detectados que interesan, el de pulsación de la pantalla (marca el comienzo de un trazo), el de deslizamiento por la pantalla y el de dejar de pulsar la pantalla (marca el final de un trazo).

Cuando se detecta el primer evento, que se produce una vez por trazo que realiza el usuario, se establece el punto tocado de la pantalla como punto inicial de *Path* con *moveTo()*.

El segundo evento, que se produce múltiples veces mientras el usuario realiza el trazo, consiste en ir tomando los puntos de la pantalla por los que se produce el deslizamiento para indicar a *Path* que guarde líneas de un punto al siguiente con *lineTo()*. Cada punto de la pantalla tomado viene marcado por cada vez que se detecta un evento táctil de deslizamiento, es decir, cada vez que el sistema operativo llama a *onTouchEvent()* se está tomando un siguiente punto en *Path* y la línea que se forma entre el este y el inmediatamente anterior. Sin embargo, visualmente no se aprecian ni los puntos ni las líneas entre ellos. Esto se debe a la alta frecuencia de detección de eventos del sistema operativo que consecuentemente provoca que sean tantos los puntos tomados que se pierda la noción de conjunto de puntos y rectas que los unen que finalmente forman el trazado.

El último evento, que se produce una vez por trazo que realiza el usuario, aplica el trazo (*Path*) generado hasta ese momento en el lienzo *Canvas* y después lo resetea para poder conformar uno nuevo. El método finalmente usa *postInvalidate()* cada vez que *onTouchEvent()* es llamado (un evento de los anteriores se ha detectado) para forzar que se aplique el trazado generado aún en su estado parcial sobre el lienzo. El método *onDraw()* no es llamado explícitamente, sino que es el sistema operativo el que le llama indirectamente cuando al final de *onTouchEvent()* se usa *postInvalidate()*. Si no se realizara este fuerce el trazo dibujado por el usuario no se pintaría hasta estar este completo, es decir, hasta que se produce el evento de dejar de pulsar la pantalla (levantar el dedo de la pantalla). De esta forma se logra que el trazo se dibuje según se va realizando pues tras cada evento de deslizamiento por la pantalla detectado se redibuja completamente el trazado (todos los puntos almacenados y líneas calculadas entre puntos de *Path*).

El control del visualizador para borrar las pintadas hechas sobre el lienzo lo único que hace es solicitar a esta vista personalizada que borre su lienzo *Canvas* (también resetea *Path*).

Finalmente, por cuestiones de rendimiento y optimización, se utiliza *Bitmap* a modo de caché para mejorar sustancialmente los tiempos de renderizado.

8.1.6.2 Clases Enum

La aplicación cuenta con un paquete fuera de la vista que contiene otras clases *Enum*. Este paquete está fuera de cualquier otra capa del proyecto por lo que contiene mezclados varios *Enum* referidos a distintas capas. En lo que se refiere a la vista se cuenta en este paquete con varias clases *Enum*.

Existe un *Enum* que lista los tipos de interacciones con las que cuenta el visualizador particular de imágenes. Se considera también la interacción *NINGUNA*. Esta clase no tiene atributos de clase por lo que se trata de un *Enum* en su mínima expresión.

Otro *Enum* define los tipos mime de los tipos de recursos que maneja la aplicación. Estos básicamente son PDF, imágenes PNG y JPEG (o JPG) y HTML. Asocia varias extensiones de ficheros a un mismo tipo mime. También aporta varios métodos útiles de acceso estático para obtener el tipo mime (instancia *Singleton* de este *Enum*) de un recurso a partir de su dirección URL. Aunque esto pueda estar más relacionado con la lógica de negocio se menciona aquí porque conceptualmente está ligado a los visualizadores de la vista.

Algo semejante al anterior ocurre con otro *Enum* que sí tiene atributos de clase. En esta ocasión a través de sus atributos que son un array de elementos del *Enum* anterior y la información que necesita *Java Reflection* para encontrar la clase de un visualizador, este *Enum* asocia varios elementos del *Enum* anterior a un tipo de recurso que es precisamente lo que define este *Enum*. Estos tipos son básicamente *PDF*, *IMAGEN* y *WEB*. De hecho, hay un tipo de recurso en este *Enum* por visualizador que existe. Implementa también métodos de acceso estático que usando *Java Reflection* y también los métodos de acceso estático definidos en el *Enum* anterior proporciona métodos a quién deba usarlos que permiten obtener el tipo de recurso (instancia *Singleton* de este *Enum*) tanto a partir del tipo mime del recurso como a partir de la dirección URL del recurso. Obtener el tipo de recurso es directamente proporcional a poder obtener el visualizador que se debe utilizar para visualizar dicho recurso.

Existe otro *Enum* que lista las secciones de la aplicación y cuenta con atributos de clase que almacenan identificadores a recursos en ficheros xml para el título de la sección y el *drawable* imagen-icono correspondiente a cada sección.

Finalmente existe otro *Enum* análogo al anterior para las subsecciones de la sección imágenes que en este caso sólo cuenta con un atributo de clase para el título de cada subsección pues la imagen-icono que utiliza es la de la sección imágenes.

8.1.6.3 Estructuración del estilo de las vistas y los layout

Al implementar las disposiciones de vistas y las vistas en estas, en definitiva, las partes de las que se compone la interfaz gráfica, ha sido frecuente que se aplicaran repetidamente bastantes atributos con valores repetidos. Esto se refiere a tamaño de las vistas, colores, posiciones, tamaño del texto, imágenes, en definitiva, lo que se conoce como estilo o aspecto. Por ello, siguiendo la idea de las hojas externas de estilos *CSS* que se usan en las tecnologías Web se han estructurado las vistas y los *layout* usando recursos xml para definir estilos en el fichero *styles.xml*.

De esta forma se ha evitado repetir numerosísimas líneas de código y se ha centralizado el estilo o aspecto de todos los elementos gráficos en un único fichero. Esto tiene varias ventajas claras como que se evitan errores por el copiado y pegado del código, ayuda a la mantenibilidad del código pues un futuro cambio sólo debe cambiarse una vez en una línea, mejora la claridad y legibilidad del código y, entre otros, permite crear versiones alternativas del fichero *styles.xml* y por ello, versiones alternativas de todos los estilos que definen de forma visual a la interfaz gráfica.

Los estilos en este fichero se definen declarando un elemento *item* para cada atributo o propiedad xml de una vista o *layout* para la que ese estilo especifica un valor concreto. Los estilos admiten herencia de otros recursos *style* aunque no es una herencia múltiple. Esto es muy útil

para que de igual forma que se ha hecho en *Java* se puedan estructurar correctamente todos los estilos declarando estilos más generales al principio y luego declarando estilos más específicos que heredan de los generales.

Por otro lado, al definir los estilos, en lo que se refiere a la parte en la que se especifica el valor para cada propiedad o atributo xml definido con un *item*, en el caso de que sean valores hexadecimales que se refieren a colores es frecuente que los colores se definan en un fichero de recursos xml llamado *colors.xml* y que en el fichero *styles.xml* se usen referencias (son alias realmente) a los recursos de color definidos en *colors.xml*. Esto es así, entre otros, para ayudar a la definición de temas.

Lo análogo ocurre con el fichero *dimens.xml* a la hora de especificar valores de tamaños, anchos, altos, márgenes, relleno, etc., es decir, a la hora de definir valores que se usan para establecer las dimensiones, separaciones, ubicaciones, etc. de las vistas en las disposiciones de vistas o *layout*.

8.1.6.4 Adaptación de la vista a otros tipos de dispositivos

Como se ha comentado antes, gracias al uso de estilos y a que en los estilos se toman todos los valores correspondientes del fichero *dimens.xml*, con sólo crear versiones alternativas de este fichero se ha logrado adaptar toda la interfaz gráfica de HepApp a diferentes dispositivos que cuentan con tamaños y densidades de pantalla diferentes de forma que Android decide cuál de las versiones de este fichero debe utilizar en cada dispositivo.

Se han realizado pruebas en dispositivos con los tamaños y densidades de pantalla más comunes actualmente, se ha comprobado que la interfaz gráfica se muestra correctamente en todos los dispositivos probados y por ello se tiene la certeza de que también ocurrirá con la mayoría de la gran variedad de dispositivos Android disponibles en la actualidad. El desarrollo de HepApp se ha realizado de forma completa en un dispositivo de tipo Tablet Nexus 7 emulado. Tras completarse su desarrollo se crearon versiones alternativas del fichero *dimens.xml* y finalmente se realizaron con éxito las pruebas en varios dispositivos que se acaban de mencionar.

Para crear las versiones alternativas del fichero *dimens.xml* se ha utilizado el calificador de configuración *smallestWidth* (Android Developers, 2018). Se ha creado un directorio llamado *values-sw320dp* con la versión alternativa de *dimens.xml* y el directorio llamado *values-sw600dp* con la versión original de *dimens.xml*. Dado que es una muy mala idea y mala práctica no especificar una versión por defecto del fichero *dimens.xml* dado que Android podría no utilizar ninguna de las versiones de este fichero especificadas en directorios de recursos alternativos en el directorio llamado *values* también existe en este el mismo fichero *dimens.xml* que el del directorio *values-sw600dp*.

La idea de tener el mismo fichero *dimens.xml* en dos directorios diferentes chirría un poco, por ello en el momento de crear los directorios de recursos alternativos se intentó sin éxito evitar esto. En el momento del desarrollo se era consciente de la posibilidad que ofrece

Android de crear recursos de alias, pero en el caso de *dimens.xml* no se puede crear un alias apuntando al fichero, sino que sólo se puede crear un alias apuntando a cada uno de los recursos que se definen en este fichero y por lo tanto para este caso no es una solución porque está lejos de ser una solución escalable dada la cantidad de recursos que se definen en *dimens.xml* (implicaría crear un fichero xml por recurso definido).

El motivo del uso del calificador de configuración empleado es que es independientemente de la orientación del dispositivo en cada momento y sobre todo que permite diferenciar muy bien entre dispositivos Smartphone y dispositivos Tablet. Se usa *320dp* y *480dp* para dispositivos Smartphone, y *600dp* y *720dp* para dispositivos Tablet. Sólo se han definido versiones alternativas de *dimens.xml* para *320dp* y *600dp* por lo que en dispositivos que fueran más afines a *480dp* y *720dp,* aunque la visualización es la correcta, esta seguramente podría ser optimizable. Esto se lograría creando otras dos versiones alternativas de *dimens.xml* ubicadas en los directorios *values-sw480dp* y *values-sw720dp*, pero no se han implementado en HepApp. Podrían quedarse como una futura tarea menor pendiente.

Se puede decir entonces que, gracias a este mecanismo, HepApp cuenta con una versión móvil y una versión para Tablet integrada en una única aplicación.

8.1.6.5 Soporte de diferentes idiomas y mensajes de error

Ya se ha mencionado que gracias al fichero *strings.xml* y al sistema de recursos alternativos que ofrece Android se puede añadir fácilmente soporte para varios idiomas creando una versión alternativa de *strings.xml* por idioma al que se quiera dar soporte. Sí se debe mencionar un matiz importante, en un principio, en el fichero *strings.xml* se deben ubicar todas las cadenas de texto que puedan aparecer en la interfaz gráfica pero también las cadenas de texto susceptibles de poder ser traducidas que puedan acabar llegando al usuario final.

En el caso de HepApp en las capas de la lógica de negocio, lógica de conexión y modelo se utilizan mensajes de error que no van a llegar al usuario final y que están por ello escritos en el idioma nativo de quién ha desarrollado el sistema. Sin embargo, estos mensajes llevan un identificador numérico único. Un identificador numérico no necesita ser traducido porque es común para todos los idiomas. El identificador y los mensajes asociados en castellano tienen exclusivamente como fin la depuración, y cuando alguno de estos debe llegar al usuario el controlador los intercepta y cambia el mensaje de error por otro más amigable para el usuario que se toma de *strings.xml* y por ello se muestra correctamente traducido al idioma del usuario de la aplicación.

Esto en un principio puede reducir drásticamente la efectividad de poder encontrar un fallo en el sistema no detectado anteriormente a partir de la retroalimentación aportada por un usuario. Para solucionar esto, del mensaje de error original sí se podría dejar pasar el identificador numérico al mensaje final que se muestra al usuario para que cuando el problema llega al desarrollador este a partir de una tabla de correspondencias de identificadores de errores sepa de forma muy precisa el motivo y la parte del código en la que se origina el error.

Un detalle importante es que se debe utilizar un identificador diferente para dos situaciones con diferentes causas que puedan provocar un mismo mensaje de error lanzado por un mismo código.

Aunque este mecanismo para la gestión de los errores se encuentra implementado en HepApp, este está inactivo ya que se ha configurado al controlador para que enmascare completamente al usuario los mensajes de error originales con sus identificadores numéricos. Cuando el usuario notifica un error al desarrollador este indica el mensaje de error que la aplicación ha mostrado en la pantalla, pero también le indica qué orden de interacciones ha realizado hasta que se ha llegado la situación que ha generado el error. Con esta información, el desarrollador activando el sistema de *Log* (registro de errores para la depuración) puede reproducir el error y a partir de ello investigarlo y corregirlo. Por otro lado, en el caso de que un error fuera difícil de reproducir, o de cualquier otro problema que no permita al desarrollador dar con el error siempre se podría mandar al usuario concreto una versión de la aplicación con el mecanismo que permite hacer llegar al usuario del total de los mensajes originales de error sólo los identificadores, es decir, desactivar que el controlador enmascare también los identificadores numéricos. Esto busca que la aplicación pública para todo el mundo sea la que enmascara los mensajes de error por completo y que sólo como última instancia a muy pocos usuarios concretos se les mandara la aplicación que no enmascara los identificadores numéricos de los mensajes de error. Detrás de todo esto lo único que se busca es la privacidad del código y la seguridad del sistema basada en el principio de la ocultación de información al evitar proporcionar abiertamente detalles sobre la implementación concreta llevada a cabo. En el desarrollo de código *open source* esta práctica carecería de sentido, pero en el caso de HepApp se comprueba necesaria.

8.1.6.6 Uso de recursos drawable xml en lugar de imágenes

En la interfaz gráfica se han sustituido bastantes imágenes que tenía la interfaz gráfica de partida y referencia que se tenía de proyectos previos por recursos xml *drawable* definidos con código. Para ello, se han utilizado de forma combinada elementos como *shape*, *stroke*, *solid*, *corners*, *padding*, *selector* y *layer-list* que con los atributos y propiedades con valores correctos asignados lo han permitido. De esta forma se reduce notablemente el tamaño del fichero con el instalador de la aplicación (fichero *APK*) porque se ahorra necesitar bastantes imágenes junto a todas sus versiones alternativas frente a unos pocos ficheros xml de muy reducido tamaño. Por otro lado, esto también agiliza y optimiza el rendimiento y consumo de memoria de la interfaz gráfica.

8.2 Controladores

Es la capa que interactúa tanto con la capa de la vista como con la capa modelo. El controlador atiende las peticiones que cambian o modifican el modelo utilizando la interfaz proporcionada por el modelo. Durante la explicación de la capa vista se le ha mencionado en numerosas ocasiones de forma que se ha adelantado una buena aproximación de lo que es la

capa controlador. El controlador actúa frente a la vista como el intermediario entre esta y el resto del sistema software.

Conceptualmente hablando, el patrón *MVC* presenta el problema de que define muy bien la capa vista y la capa del modelo pero no la capa del controlador, pues muchas veces se interpreta como las funciones que el patrón define para esta capa sumando todo lo demás no definido en otras capas (actualmente se trata de algo bastante indefinido). En HepApp se han definido otras capas o subsistemas como la lógica de conexión y la lógica de negocio que se verían según el concepto del patrón *MVC* como partes (subcapas de igual nivel) de lo que se conoce como "controlador". Es entonces frecuente que se utilice el concepto de la capa controlador del patrón *MVC* combinado con los patrones GRASP patrón *Indirección* y patrón *Controlador* (Larman, 2001).

Al aplicar el patrón *Indirección*, se puede decir que el controlador es la única capa irremediablemente acoplada a la capa de la vista y al resto de las capas que conforman el sistema. El resto de las capas que conforman el sistema, idealmente, sólo se encuentran acopladas a la capa controlador y por ello se pueden considerar independientes porque en teoría, estas podrían trasladarse a otro proyecto diferente sin tener que realizar modificaciones ya que sólo sería necesario cambiar la capa controlador en dicho proyecto. Aunque desde el punto de vista del acoplamiento y dependencia entre capas la vista podría considerarse ciertamente independiente, dado que la interfaz gráfica suele ser bastante propia de cada sistema será más difícil que sea reutilizada completamente (lo que no quita para poder reutilizarse partes independientes de esta). Combinando el patrón *MVC* con el patrón *Indirección*, de forma breve, idealmente se logra que ninguna capa del sistema conozca que las demás existen y que, como excepción, la capa controlador sea la única que conoce que todas existen para encargarse de coordinarlas actuando como intermediario entre todas ellas.

Respecto al patrón *Controlador*, relacionando con lo anterior de la independencia de una capa que no es consciente de la existencia de nada más fuera de ella, este patrón sugiere que cuando se desee en un proyecto software usar o acceder a una capa o subsistema se debe utilizar entre esta y el resto del proyecto un elemento que actúe como intermediario de forma que es este elemento el que recibe las peticiones que el resto del proyecto desea realizar sobre la capa para adaptarlas y enviarlas a la capa. El elemento también mediará en el otro sentido si la capa debe devolver algún resultado. Este elemento intermediario es al que se llama controlador y la idea se puede trasladar a diferentes puntos. Si en lugar de entenderse entre capas se entiende entre sistemas completos independientes también se puede aplicar la idea para coordinarlos a ambos con el fin de llevar a cabo una tarea aún más compleja que la que desempeñan ambos sistemas por separado. Del mismo modo se puede aplicar a subcapas de igual nivel que conforman una capa de un sistema. Por ello, este patrón, lo que sugiere es la idea descrita y que esta se aplique varias veces en los diferentes puntos de un desarrollo software en el que se cree necesario, es decir, recomienda que se creen varios elementos controladores.

A continuación, se explica de forma más concreta cómo se han interpretado y aplicado estos patrones en HepApp. Se ha creado un objeto controlador global del sistema que es el que aplica los conceptos descritos de aplicar de forma combinada los patrones *MVC*, *Indirección* y *Controlador* ya descritos. Aunque el sistema completo se encuentra formado por cinco capas

bien definidas, para entender esta aplicación combinada de patrones, se va a explicar en el momento de aplicar cada patrón como se debe ver de forma lógica o abstraída al sistema.

Desde el punto de vista del patrón *MVC* se entiende el sistema completo como una capa vista, una capa modelo y una capa controlador siendo las capas lógica de negocio y lógica de conexión (al combo se le denominará capas de lógica) subcapas de igual nivel de la capa controlador.

Desde el punto de vista del patrón *Indirección*, se entiende el sistema completo como las cinco capas y es el controlador global entre las otras cuatro el que aplica este patrón actuando como intermediario y coordinando a todas ellas.

Desde el punto de vista del patrón *Controlador*, también se entiende el sistema como las cinco capas y se crea un controlador propio para cada capa de igual nivel debajo de la capa controlador constituida únicamente por el controlador global, es decir, un controlador propio para las capas modelo, lógica de negocio y lógica de conexión. Hecho esto, se aplica de nuevo el patrón para crear ese controlador intermediario entre la vista y las restantes capas ya con su controlador propio, es decir, se aplica de nuevo para crear lo que se ha llamado como controlador global o principal. Los otros controladores se llaman controladores de fachada, aunque se pueden entender como subcontroladores. En esta segunda vez que se aplica el patrón *Controlador* se entiende el sistema como si estuviera formado por la capa de vista y una capa constituida por las restantes (modelo + capas de lógica) entre las cuales se mete.

Viendo el sistema como las cinco capas bien definidas que lo constituyen, a nivel de implementación, cada una de estas capas, como ya se ha dicho, define su propio controlador (de fachada) y además este implementa una interfaz. Los controladores de cada capa (de fachada) se van a entender como parte de esta propia capa o como integrados en la capa. Esto es debido a la interfaz que, aunque sólo la implementa el objeto controlador, dado que todo lo externo independientemente del sentido pasa por este, se podría entender también como la interfaz de la capa, o sea, como si de un sistema completo independiente se tratara. Con esta implementación, el controlador global, guardará como atributos de clase referencias a los controladores de las otras capas por debajo de él. Realmente no guarda referencias de los objetos controladores, sino que lo que guardan son objetos del tipo de la interfaz que implementan estos controladores (instancias de los controladores tras realizar sobre estas *casting* al tipo de la interfaz). La gracia de esta implementación es que limita al controlador global a que este utilice, de todos los métodos que pueda tener implementados el controlador de una capa, sólo aquellos métodos que ha declarado en la interfaz.

Por otra parte, para el controlador global se ha empleado el patrón *Singleton* de forma que no se pueden crear instancias de su clase. Esto se debe a que sólo es la vista la que obtiene la referencia de este controlador para a través de él solicitar las cosas que necesita que no son de su incumbencia. Además, el controlador global o no tiene estado o si lo tiene se debe mantener entre las distintas actividades que obtengan en *onCreate()* la referencia a este. También, otro motivo es simplemente el de optimización del rendimiento pues es se considera pesada la tarea de estar constantemente instanciando objetos con ninguna (o casi ninguna) variación, lo cual se evita aplicando este patrón.

El controlador global, del mismo modo que lo hacen los otros subcontroladores, también podría declarar e implementar su propia interfaz de cara a la vista de forma que la vista

use una referencia del tipo de esa interfaz, pero es algo que en esta ocasión se ha decidido no llevar a cabo pues no se consideraba que aportara tanto beneficio como con los subcontroladores porque la capa controlador (compuesta en este caso sólo por el controlador global) sería la más difícil de reutilizar en otros proyectos dado que es la que está acoplada con las demás capas.

En alguna ocasión, aunque ya se haya dicho que no es algo conveniente, por motivos de simplicidad, se ha necesitado permitir pasar algún objeto de tipo *Context* al controlador global y puede que a través de este a otras capas inferiores. Por ello, el controlador global implementa un método *init()* que se explica más adelante.

Los controladores específicos de cada capa se explican en el subapartado en el que se habla de cada una. El controlador principal se explica a continuación, pero se parte de que se han leído de antemano los subapartados *Modelo*, *Lógica de negocio* y *Lógica de conexión* pues su explicación asume conocidos los conceptos explicados en estos.

El método *init()* del controlador principal genera una instancia de los controladores propios de las otras capas y guarda como atributos de clase las referencias a estas como referencias de los tipos de las interfaces que cada uno de ellos implementa. Se podría haber aplicado el patrón *Singleton* en los controladores específicos de cada capa dado que son constantemente llamados, aunque no se ha hecho ya que sólo se instancia un objeto de estos controladores una única vez en el sistema desde una clase que usa el patrón *Singleton* (controlador principal). Se puede ver que cumplen implícitamente el patrón *Singleton*. Se debe aclarar el detalle de que al guardar la referencia de un controlador en un atributo de clase del tipo de la interfaz que implementa ya se hace de forma implícita y automática la operación de *casting*.

El controlador principal también debe llevar a cabo la tarea de gestionar los errores que pueda recibir de los tipos propios de cada capa al llamar a través de sus controladores a métodos que estas ofrecen. Los errores también los puede recibir con ciertos valores de retorno de los métodos a los que llama (no siempre con excepciones). La gestión puede consistir en realizar alguna operación a mayores o puede consistir simplemente en empaquetar el error en una excepción de un tipo que pueda reconocer la capa vista para notificar el estado al usuario. A parte, también cuenta con el mecanismo que, en caso de que el error se vaya a pasar a la capa vista, debe decidir si oculta o no el identificador de error.

Por lo general, como ya se ha descrito en otros apartados, las solicitudes que impliquen a la lógica de negocio requerirán que antes se hayan hecho solicitudes a las capas modelo y/o lógica de negocio incluso en varias ocasiones. También, los resultados ofrecidos por la lógica de negocio serán los que se muestren en la capa vista previa adaptación si fuera necesario.

Recordado esto, sólo queda decir que este controlador implementa todos los métodos necesarios para solicitar a las otras capas todos los casos de uso que ofrecen y ya se han descrito en los respectivos subapartados que hablan de cada una. También, implementa los métodos

descritos en el subapartado que habla de la capa vista (porque es esta quién los llama y a la cual se los ofrece) que, llevando a cabo tareas normalmente sencillas de coordinación, redirección y mediación, genera nuevos casos de uso (de más alto nivel porque implican a varias capas).

Aunque se ha dicho que la capa controlador sólo está compuesta por el controlador principal para simplificar el escenario y contexto de las explicaciones antes dadas, realmente esta capa cuenta con otras dos clases de nombres *EjecutorObtenerVersionRecurso* y *EjecutorSincronizarBaseDatos*. Estas clases heredan de la clase *AsyncTask* y como su propio nombre indica implementan respectivamente un gestor o controlador de la ejecución de las tareas de la obtención de la versión del fichero asociado a un recurso remoto y la sincronización de la base de datos local con la base de datos remota. Están ubicadas en la capa controlador porque la gestión que llevan a cabo implica el uso de las posibilidades de varias capas. Dado que forman parte de la capa controlador, como ya se ha mencionado en el subapartado *Vista*, podrán y de hecho son utilizadas desde la capa vista con el método *execute()*.

Se ha decidido utilizar las *AsyncTask* porque el uso para el que están pensadas encaja perfectamente para llevar a cabo las tareas deseadas. Según la documentación oficial de Android, estas permiten realizar trabajo asincrónico en la interfaz de usuario (Android Developers, 2018). Realizan operaciones en segundo plano en un subproceso de trabajo y, luego, publican los resultados con el subproceso principal de la aplicación (*UI thread*), sin que haya que manejar los subprocesos o los controladores.

Los métodos *onPreExecute()*, *onProgressUpdate()* y *onPostExecute()* se ejecutan en el subproceso de la Interfaz gráfica (es el subproceso principal de la aplicación *UI Thread*), y el método de retrollamada *doInBackground()* se ejecuta en un grupo de subprocesos propio y gestionado automáticamente por la *AsyncTask*.

Como ya se ha dicho en los subapartados *Actividad principal* y *Visualizador genérico* cuando se hablaba del controlador como un ente (capa completa) que aún no se había definido, estas clases son utilizadas desde la capa vista donde se instancia un objeto (de alguno de los tipos que estas clases definen) sobre el que después se llama al método *execute()* pasando los argumentos necesarios. Estas clases son también aquellas que se mencionaron que desde el nivel de controlador accedían directamente a ejecutar métodos de la capa vista. Este es el mecanismo de retrollamada que tienen estas clases para aplicar directamente en la vista los resultados que generan las operaciones que llevan a cabo.

Introducidas estas clases, las tareas que llevan a cabo y lo que es una *AsyncTask*, se dedican dos subapartados para detallar cada una de ellas.

8.2.1 Ejecutor de la tarea obtención de la versión de un recurso remoto

Para poder ejecutar en su método *onPostExecute()* métodos de la actividad que la ha llamado, recibirá una referencia de la actividad como argumento en su constructor y la guardará como atributo de clase del tipo *WeakReference* particularizado al tipo *Activity*. Esto se ha implementado de esta forma para evitar el problema de las fugas de memoria (*leaks*). Por ello, sólo se puede recuperar la referencia a la actividad a partir del atributo de clase para guardarla en una variable local de los métodos en los que sea necesario utilizarla. Lo que se guarda como variable local es la referencia de tipo *Activity* (general) obtenida del atributo de clase a la que se aplica una operación de *casting* al tipo de actividad que se desea (en este caso a veces el tipo *VisualizadorRecursos* y a veces el tipo *VisualizadorImagenes*). También, en el constructor obtiene una referencia al controlador principal.

La clase sobrescribe el método *doInBackground()* que en este caso va a recibir como argumento el identificador de tipo entero del recurso que se pasa a la clase al llamarla con el método *execute()*. Este método, que como se ha dicho ejecuta sus tareas en un subproceso (realmente un conjunto, pero se percibe como uno) propio autogestionado, mediante llamadas al controlador principal va a solicitar a la lógica de conexión que, como se ve en los subapartados *Comunicación con el servicio Web* y *Gestor de la comunicación que obtiene la versión de un recurso remoto*, obtenga la versión actual y la disponibilidad de actualización del recurso simple cuyo identificador ha obtenido como argumento. El método finalmente retorna como cadena de texto la versión del recurso simple obtenida.

Tal y como se advierte en la interfaz *InterfazControladorLogicaConexion* con comentarios en cada uno de los métodos "afectados", dichos métodos no deben llamarse desde el subproceso principal de la aplicación (*UI Thread*) porque podrían hacer funcionar mal a la aplicación completa y provocar que aparezcan diálogos de espera en los que el sistema operativo pregunta al usuario si desea forzar el cierre de la aplicación ya que está tardando mucho en responder. Estos métodos deben llamarse desde un subproceso (*thread*) a parte o algo equivalente (por ejemplo, *AsyncTask*). Esto también se indica en el controlador principal en todos los métodos de este que son redirecciones a los métodos afectados de la capa lógica de conexión, o en cualquier método que en función del valor de los argumentos que recibe puede llamar a uno de los métodos afectados de la lógica de conexión (se indica en el comentario para qué valor o valores de qué parámetro o parámetros no sería segura su ejecución desde el subproceso principal). Por lo general, todas las solicitudes que hace la vista al controlador principal se ejecutan en el subproceso principal de la aplicación salvo cuando se ejecutan aquellos marcados con comentarios. Los métodos afectados de la lógica de conexión son los dos métodos que llaman a los métodos (cada uno a un método) *obtenerTabla()* y *obtenerVersionRecurso()* que se explican en los subapartados *Gestor de la comunicación que obtiene la información para sincronizar la base de datos* y *Gestor de la comunicación que obtiene la versión de un recurso remoto*. Estos métodos no podrían llamarse desde el subproceso principal de la aplicación debido al temporizador que implementan ya que podría bloquear la aplicación por ese tiempo que Android consideraría excesivo como para mostrar los diálogos que indican que la aplicación no responde. Aclarado todo esto, estos métodos de la lógica de conexión que son llamados a través de los controladores (el principal y el propio de la capa) sólo se usan desde el método *doInBackground()* que implementan las *AsyncTask* de las otras dos clases que forman parte de la capa controlador. Igualmente, las *AsyncTask* siguen pudiendo utilizar en este método otros métodos del controlador principal de igual forma que lo hace el subproceso principal de la aplicación.

La clase sobrescribe el método *onPostExecute()* que va a recibir como argumento la cadena de texto que retorna el método *doInBackground()*. Este método, aparte de proporcionar a través del controlador principal la versión del recurso que se ha recibido a la lógica de negocio (previo paso por la capa modelo que en este caso va a devolverla sin realizar ninguna operación) para deducir si se dispone de una actualización, va a actualizar consecuentemente los controles de la barra inferior del visualizador e independientemente de esto, si el recurso ya estaba registrado como local, va a cargarlo automáticamente en el visualizador. Si el resultado obtenido es erróneo porque, por ejemplo, vale *null* porque el temporizador ha expirado, este método ejecutará el apropiado del visualizador para notificar el error al usuario. Este método, en función de un *flag* de reintentos automáticos que la actividad establece tras instanciar esta clase y antes de llamar al método *execute()*, llamará o no a un método del visualizador que acaba desencadenando que se cree y ejecute una nueva instancia de esta clase, es decir, que se vuelva a ejecutar de nuevo la *AsyncTask*. Eso sí, este método de la actividad tiene en cuenta el contador de reintentos del que ya se habló en los subapartados que hablan de la capa vista, que, a fin de cuentas, determina el número de reintentos que se permiten realizar. De esta forma, no se puede producir un bucle infinito de reintentos que imposibilite el uso de la aplicación. Se debe tener en cuenta que una *AsyncTask* es "de un solo uso" por lo que no se puede volver a ejecutar la misma instancia de una clase *AsyncTask*. Para volver a ejecutar la misma *AsyncTask* se debe instanciar de nuevo la clase que la define.

Por otro lado, en un uso offline de la aplicación el hecho de que se utilicen reintentos automáticos (que ya se sabe de antemano que van a fallar) no retrasará apenas la solicitud de cargar un recurso registrado como local en el visualizador porque la *AsyncTask* sin conexión a Internet finalizará su cometido muy rápidamente (devolviendo como respuesta un error) provocando que el temporizador se cancele sin apenas esperar.

Esta forma de haber implementado los reintentos puede considerarse no demasiado recomendable dado que, aunque sea mediante la actividad del visualizador, se puede entender que una *AsyncTask* genera y llama indirectamente a otra nueva *AsyncTask* que va a realizar la misma tarea que la que la llama. La implementación se ha realizado de esta forma por simplicidad, porque no causa inestabilidad en el uso de la aplicación y porque cuando la segunda *AsyncTask* ejecuta su método *doInBackground()* (que se ejecuta en un grupo nuevo de subprocesos) la que la ha llamado finaliza, por lo que en este caso siempre se tiene la garantía de que la *AsyncTask* que llama a una nueva va a finalizar antes de que esta nueva a la que llama finalice e incluso antes de que pueda lanzar a su vez otra nueva tercera *AsyncTask*.

Finalmente, decir que la clase cuenta con un método que permite cancelar la tarea que se está realizando llamando al método que cancela la tarea de la lógica de conexión que ya se ha descrito en el subapartado correspondiente y después llamando al método *cancel()* propio de la *AsyncTask* que provocará la cancelación de esta tan pronto como sea posible.

8.2.2 Ejecutor de la tarea sincronización de la base de datos

Se trata de la clase análoga a la descrita en el subapartado anterior que pretende llevar a cabo la tarea de la sincronización de la base de datos. Así pues, sólo se van a describir las particularidades que esta presenta dado que las partes análogas con la otra clase ya se han descrito en el subapartado anterior.

En esta ocasión, se implementa el método *onPreExecute()* para mostrar la capa oculta de la actividad principal que indica al usuario con una animación de progreso indefinido que se está realizando la sincronización de la base de datos local con la base de datos remota.

En esta ocasión el método *doInBackground()* solicita la tabla de la base de datos remota que contiene la versión actual de cada tabla y de la propia base de datos (con que cambie una tabla esta también cambia a la versión de la última tabla editada). A partir de los datos de esta tabla deduce qué otras tablas disponen de actualización para sólo pedir estas y que no sea necesario pedirlas todas. Así pues, obtiene del servicio Web todas las tablas que ha determinado y después genera con ellas todas las consultas SQL con las que va a formar la transacción que deberá ejecutar sobre la base de datos local. Finalmente ejecuta la transacción.

Como ya se ha dicho, todas estas tareas descritas las realiza el método *doInBackground()* a través de solicitudes al controlador principal quien después las lleva a cabo realizando las solicitudes necesarias a las otras capas del sistema.

Se debe mencionar que en este caso la tarea final de *doInBackground()* de ejecutar la transacción sobre la base de datos se considera una sección crítica por lo que antes de entrar en ella, se activa un mecanismo que mientras esté activo evite que se pueda ejecutar el método *cancel()* propio de la *AsyncTask*. El mecanismo se implementa mediante un *flag* booleano y el modificador *synchronized* de un par de métodos *getter* y *setter* definidos para ese *flag*. De nuevo, habiendo dispuesto de más tiempo para aprender el funcionamiento del paquete *java.util.concurrent* se podría haber realizado una mejor implementación de este mecanismo.

El método *onPostExecute()* que recibe el valor booleano que devuelve el método *doInBackground()* sólo oculta de nuevo la capa oculta que informa del progreso de la tarea de sincronización al usuario y muestra un mensaje al usuario indicando el éxito o el error resultante de la realización de la tarea de sincronización.

Por último, se debe indicar que el método que implementa la clase para cancelar la tarea que gestiona la *AsyncTask*, como antes, solicita a la lógica de conexión la cancelación de las tareas que tenga pendientes, oculta la capa oculta de la actividad principal y sólo llama al método *cancel()* si no se encuentra activado el mecanismo que protege la ejecución de la sección crítica del método *doInBackground()*, lo cual lo averigua utilizando el método *getter* con el modificador *synchronized* que proporciona el propio mecanismo. Se debe mencionar que este modificador de los métodos del mecanismo es necesario porque el mecanismo puede ser activado desde el método *doInBackground()* que se ejecuta en un subproceso a la vez que el usuario solicita (dando al botón atrás) la cancelación de la tarea que provocaría la ejecución del método de cancelación de la *AsyncTask* desde el subproceso principal de la aplicación, lo cual

podría suponer que se produjera un acceso concurrente desde dos subprocesos diferentes al mecanismo para activarlo y comprobar su estado simultáneamente. Al acabar la sección crítica del método *doInBackground()* se desactiva el mecanismo de protección de la ejecución.

8.3 Modelo

Es una de las capas que componen al sistema y su finalidad va a ser la de gestionar, proporcionar formato y almacenar la información que va a manejar el sistema. Proporciona, como mínimo, los casos de uso básicos sobre la información que almacena, esto es, leer, escribir, editar y eliminar (casos de uso *CRUD*). En un entorno de trabajo donde se realizan accesos en paralelo debe proporcionar atomicidad dando la garantía de que cualquier acceso sobre el modelo se va a realizar de forma segura. Esta capa recibe del controlador principal y a través de su propio controlador las peticiones que debe llevar a cabo. La capa implementa su propio tipo de excepciones y un *Enum* asociado con todas las posibles causas que pueden generar que se lance este tipo de excepción. Todos los errores que se puedan dar en esta capa se encapsulan en su tipo de excepción propio antes de permitir que lleguen al controlador principal. Por otro lado, esta capa se divide en dos subsistemas de igual nivel: El subsistema de la base de datos y el subsistema de entrada y salida básico.

8.3.1 Subsistema base datos

Es el subsistema que implementa la base de datos que va a ser una copia local sincronizada de parte de la base de datos que existe en la máquina servidora. También implementa las clases que proporcionan estructura a la información contenida en la base de datos para que esta pueda ser procesada posteriormente por la lógica de negocio, y clases que definen lógica propia de esta capa para realizar accesos sobre esta base de datos de la que se va a extraer la información.

8.3.1.1 Base de datos

Se va a utilizar el sistema gestor de bases de datos llamado *SQLite* pues Android cuenta con soporte de forma nativa para este sistema. Dado que en la máquina servidora se va a utilizar el sistema gestor de bases de datos *MySQL* (en el despliegue del servicio hosting o *MariaDB* en el entorno de pruebas) la base de datos en la aplicación presentará pequeñas variaciones.

Para implementar la base de datos se crea una clase de soporte auxiliar (*LocalSQLiteOpenHelper*) que va a heredar de la clase *SQLiteOpenHelper*. La clase declara en alguna constante el nombre de la base de datos y la versión actual. También tiene otros dos atributos de clase que son listas del tipo *String* para almacenar las consultas SQL que se usan para crear las tablas y las consultas SQL que introducen los datos iniciales con los que debe contar la base de datos.

En su constructor se cargan todas las cadenas con las consultas SQL mencionadas en las listas.

Se sobrescribe el método *onCreate()* que ejecuta todas las consultas SQL que se han cargado en las listas. De esta forman crean la base de datos y la agregan contenido inicial.

Se sobrescribe el método *onUpgrade()* que por ahora sólo elimina todas las tablas existentes y vuelve a iniciar la base de datos.

Se sobrescribe el método *onConfigure()* para llamar al método *setForeignKeyConstraintsEnabled()* de la clase *SQLiteDatabase* con el fin de habilitar las restricciones para las claves foráneas. De esta forma el método se llama cada vez que una conexión con la base de datos está siendo configurada.

Respecto a la estructura de la base de datos, aunque se explica en el subapartado *Patrón* y aún mejor en el subapartado del capítulo que habla sobre la máquina servidora *Estructura de la base de datos*, en la aplicación cuenta con ocho tablas de las cuales tres son relacionales porque sólo almacenan relaciones entre registros de las otras tablas.

8.3.1.2 Patrón DTO

El patrón *Data Transfer Object*, en la parte que interesa para este proyecto, se utiliza para proporcionar estructura a la información que contiene el modelo para que esta sea manejable por la lógica de negocio. Para ello, el patrón propone que para modelar la información se utilice una clase *Value Object* que almacene como atributos de clase la información y proporcione métodos *getters* y *setters* para poder interactuar con la información contenida en los atributos de clase. Se puede ver como una encapsulación de la información que a fin de cuentas es en lo que consiste proporcionarla estructura. Se debe crear una clase por cada tipo de información distinta a la que se debe dar estructura.

El uso de este patrón resuelve muy bien el problema existente para lograr pasar la información del modelo a la lógica de negocio de forma que esta pueda comprenderla y procesarla. Dada una tabla en una base de datos que almacena en sus registros un tipo de información, si esta tabla cuenta con numerosas columnas de diferentes tipos primitivos de datos como texto, números enteros, caracteres, cadenas de texto, etc… ¿Cómo se va a transferir esta información como argumentos a lo largo de los métodos? Si la tabla tuviera veinte columnas, sus registros se podrían transferir a lo largo de los métodos usando al menos tantos argumentos como columnas tiene la tabla. Cuando el número de columnas se eleva un poco esto se convierte en algo que no es escalable. Este es el problema que tan bien resuelve este patrón, porque para transferir toda la información de un registro de una tabla se utiliza un único argumento del tipo de la clase que este patrón sugiere definir y que proporciona estructura a la información.

Por contra, el uso de este patrón provoca cierto acoplamiento entre la capa modelo y la capa lógica de negocio, porque de nada serviría aplicar este patrón si luego la lógica de negocio por no conocer la clase (aquí está el acoplamiento) no entiende la información con estructura. Por suerte, este inevitable acoplamiento no es demasiado relevante y no tiene un impacto apreciable.

Así pues, se va a crear una clase por cada tabla de la base de datos que no es relacional.

La clase *RecursoVO* va a contener los recursos o recursos simples que están formados por un identificador, un identificador de sección a la que pertenecen y tres cadenas de texto para el título, descripción y dirección URL del recurso.

La clase *ModuloVO* va a contener los módulos que no son más que elementos clasificadores o colecciones de recursos y están formados por un identificador y dos cadenas de texto para el título y la descripción.

La clase *RecursoCompuestoVO* va a contener los recursos compuestos que son colecciones de hasta cuatro recursos simples y cuentan con los mismos campos que los módulos. La diferencia entre estos y los módulos es que estos agrupan recursos de características similares y los módulos agrupan recursos indiscriminadamente.

La clase *RecursoVersionVO* va a contener información adicional sobre aquellos recursos simples que se han registrado en el sistema como locales, es decir, se han descargado. Están formados por el identificador del recurso al que se refieren (en la base de datos es una clave foránea) y dos cadenas de texto para el nombre del fichero descargado asociado al recurso y la versión de dicho fichero.

La clase *ControlVersionesVO* va a contener información sobre el estado de sincronización de cada tabla con la máquina remota. Cuentan con un identificador de tabla y una cadena de texto para la versión actual de la tabla en la aplicación (base de datos local). También almacena el estado de sincronización de la base de datos completa, pues este cambia cada vez que una sola tabla cambia su estado.

8.3.1.3 Patrón DAO

El patrón *Data Acces Object* se utiliza para abstraer el acceso a los datos de la o las diversas fuentes de información desde las que se pueden obtener. El ejemplo típico que permite ilustrar la necesidad de este patrón es el de tener que cambiar un sistema ya implementado de un sistema gestor de bases de datos a otro. El uso de este patrón permite que se puedan obtener los datos de forma independiente de la fuente de información en la que se encuentran sea un fichero o una base de datos entre otros. Está por ello bastante relacionado con el patrón *DTO*.

En el caso concreto de este proyecto, al aplicar el patrón, se genera una interfaz por cada clase que se ha mencionado en el subapartado *Patrón DTO*. En cada una de estas interfaces de declaran los métodos que van a permitir interactuar con la información. Como se ha comentado antes, como mínimo se declaran los métodos para los cuatro casos de uso básicos CRUD de acceso sobre los datos. Para el proyecto actual se han declarado en cada interfaz métodos para otros casos de uso más avanzados o más complejos. Asociadas a estas interfaces, se debe crear mínimo una clase por interfaz que las implemente, es decir, clases que implementen como mínimo los métodos declarados en la interfaz que implementan. Para implementar estas cinco clases, se ha implementado antes una sexta que va a actuar como clase general para que usando de nuevo la herencia se evite tener que implementar de forma repetida métodos y comportamientos que serían comunes y necesitados en las cinco clases.

La clase *LogicaDAO* es la clase general y aporta métodos para realizar consultas *SELECT*, *INSERT*, *UPDATE* y *DELETE* sobre la base de datos. Estos métodos reciben como argumentos las consultas SQL como cadenas de texto y utilizan la clase *SQLiteDatabase*, que obtienen a partir de la clase *LocalSQLiteOpenHelper*, para ejecutar las consultas.

La clase *RecursoDAO* hereda de la clase *LogicaDAO* e implementa la interfaz *InterfazRecurso*. Es la clase relacionada con la clase *RecursoVO*, porque todos sus métodos retornan o reciben como argumento una instancia de esta clase. La clase cuenta con métodos más avanzados que permiten obtener un mapa de los recursos en función de los identificadores de la sección y/o de la subsección y/o del módulo.

La clase *ModuloDAO* hereda de la clase *LogicaDAO* e implementa la interfaz *InterfazModulo*. Es la clase relacionada con la clase *ModuloVO*, porque todos sus métodos retornan o reciben como argumento una instancia de esta clase. La clase cuenta con métodos más avanzados que permiten obtener un mapa de los módulos en función del identificador de la sección e insertar una entrada en la tabla relacional que contiene las asociaciones entre los recursos y los módulos. El método asociado a este último caso de uso se podría haber declarado en *InterfazRecurso* en lugar de en *InterfazModulo* pero no se ha hecho con la intención de descargar a la clase *RecursoDAO*.

La clase *RecursoCompuestoDAO* hereda de la clase *LogicaDAO* e implementa la interfaz *InterfazRecursoCompuesto*. Es la clase relacionada con la clase *RecursoCompuestoVO*, porque sus métodos retornan o reciben como argumento una instancia de esta clase. La clase cuenta con métodos más avanzados que permiten obtener un mapa de los recursos compuestos en función de los identificadores de la sección y/o de la subsección, y obtener una lista de los recursos simples asociados a un mismo recurso compuesto.

La clase *RecursoVersionDAO* hereda de la clase *LogicaDAO* e implementa la interfaz *InterfazRecursoVersion*. Es la clase relacionada con la clase *RecursoVersionVO*, porque todos sus métodos retornan o reciben como argumento una instancia de esta clase. La clase no cuenta con métodos más avanzados que los básicos.

La clase *ControlVersionesDAO* hereda de la clase *LogicaDAO* e implementa la interfaz *InterfazControlVersiones*. Es la clase relacionada con la clase *ControlVersionesVO*, porque todos sus métodos retornan o reciben como argumento una instancia de esta clase. La clase cuenta con métodos más avanzados que permiten obtener un mapa de las versiones de toda las tablas y la base de datos.

8.3.1.4 Clase de soporte auxiliar para la sincronización de la base de datos

Se trata de una clase que implementa la lógica necesaria para realizar una tarea muy concreta. Esta tarea concreta es la de llevar a cabo la parte que le corresponde al modelo en la tarea de la sincronización de la base de datos con la de la máquina remota. También actúa como un controlador específico dentro de la capa pues realiza acciones relacionadas con todas las partes de los subapartados anteriores que componen la capa modelo. Sin embargo, no se la llama controlador sino clase auxiliar porque, aparte de hacer más cosas que las que hace un controlador al implementar lógica, la tarea que controla es demasiado específica.

Antes de continuar se debe matizar una diferencia entre la lógica de conexión y el modelo. Para explicarlo se plantea la duda de quién tendría que contar con la lógica para obtener la información y abstraer al sistema del acceso a la fuente de la misma si la fuente de información no es local e implica establecer una conexión a Internet como es por ejemplo el caso de un servicio Web que proporciona una *API REST*. La respuesta es que el modelo es siempre quien debe poner la lógica para dar estructura a la información que se obtiene de la forma que sea de una fuente de información, pero el modelo sólo debe contar con la lógica de acceso a las fuentes de información locales al sistema, es decir, de la lógica que abstrae al resto del sistema del acceso a fuentes de información locales o que no implican conexiones a través de Internet. Según este razonamiento, cuando se trata de una fuente de información externa como es el caso del servicio Web es la lógica de conexión la que debe encargarse de poner la lógica de acceso a dicha fuente a través de Internet para abstraer de esto al resto del sistema. Lo importante es que aunque se trate de una fuente de información externa y acceda a ella la lógica de conexión, sigue siendo tarea del modelo recibir y dar estructura a esta información la guarde después de forma local o no. Entonces la lógica de conexión, solicitará algo que alguien le ha pedido que solicite pero que no sabe qué es y después retornará la información que reciba también sin comprenderla a quién le pidió que la solicitara. El elemento que le pide a la lógica de conexión que solicite información será el controlador principal, quién puede que comprenda qué tiene que pedir a la lógica de conexión, pero no comprenderá lo que la lógica de conexión le retorne hasta que este se la pase a su vez al modelo y el modelo se la devuelva de nuevo estructurada. Se puede decir que este es el caso en HepApp, porque la aplicación accede a la base de datos remota de la máquina servidora a través de un servicio Web disponible con una *API REST*.

Aclarado lo anterior, dado que el cometido de esta clase es hacer parte de la tarea de sincronizar la base de datos local con la remota, primeramente, esta clase se encarga de recibir la información del servicio Web que obtiene la lógica de conexión (para esta clase proviene del controlador principal porque no sabe de la existencia de la capa lógica de conexión). Después, proporciona la estructura conocida para esta información con las clases del patrón *DTO* y finalmente con el objetivo de guardar esta información de forma local actúa de controlador con las clases del patrón *DAO* y otra clase (que se podría entender que también pertenece a las clases del patrón *DAO*) que implementa la lógica para ejecutar transacciones sobre la base de datos. Así pues, interactúa con las clases del patrón *DAO* para generar las consultas SQL que deberían ejecutarse para lograr actualizar correctamente todas las tablas de la base de datos cuyo contenido se vaya a sincronizar con el que se acaba de recibir mediante la lógica de conexión. Una vez que tiene todas estas consultas SQL, aunque no las comprenda porque esta clase no pertenece a las clases del patrón *DAO*, es cuando utiliza esa otra clase para formar y ejecutar una única transacción sobre la base de datos con todas esas consultas.

En el siguiente apartado se explica la clase que permite realizar transacciones sobre la base de datos y la razón por la que estas son necesarias para este proyecto.

8.3.1.5 Transacciones en la base de datos

En referencia a las bases de datos, una transacción es una operación que se realiza sobre la base de datos de forma atómica. Esta operación generalmente consiste en ejecutar varias consultas SQL en un determinado orden. La diferencia entre ejecutar directamente las consultas SQL y ejecutarlas como una transacción es que una transacción sólo puede tener dos escenarios como resultado de la operación. Si la transacción se ejecuta correctamente esta tiene el efecto sobre la base de datos que equivale a haber ejecutado con éxito todas las consultas en el orden indicado. Si la transacción falla, el efecto sobre la base de datos equivale a que no se hubiera ejecutado ninguna consulta, aunque alguna se haya ejecutado con éxito. Básicamente en caso de fallo de alguna de las consultas, aunque alguna ya se haya realizado correctamente, lo que se hace es revertir el efecto de todas las consultas ejecutada exitosamente hasta el momento para dejar todo como estaba. La idea se resume un poco en "o todas o ninguna". Por otro lado, la transacción también tiene otro motivo conveniente por el que usarse y este es que mientras se ejecuta una transacción se impide que se ejecuten consultas de otros sobre la base de datos para evitar que entre la ejecución de dos consultas de la transacción se cuele la ejecución de otra consulta ajena que pudiera modificar y bastante el resultado que se esperaba obtener tras la transacción porque varias de las consultas de estas han producido un efecto inesperado.

Por estos dos motivos, para el proyecto HepApp, el uso de las transacciones en las bases de datos, tanto la local de la aplicación Android como la remota de la máquina servidora, supone algo imprescindible.

La implementación de esta clase se parece bastante a la de la clase *LogicaDAO* pues recibe una lista de cadenas de texto siendo cada una una consulta SQL con la que se va a formar la transacción. Implementa los métodos para formar la transacción con la lista de consultas y ejecutarla sobre la base de datos.

En el caso de la aplicación Android, puesto que es su base de datos la que se sincroniza con la remota, se debe deshabilitar la revisión de claves foráneas en el momento de ejecutar la transacción porque no se desea que una consulta *DELETE* tenga un efecto de borrado en cascada como sí ocurre en la máquina servidora. Con *SQLite* se puede activar o desactivar la revisión de claves foráneas ejecutando una consulta SQL.

Sin embargo, con la implementación hecha no se necesita estar pendiente de ir activando y desactivando esto porque las consultas de la clase *LocalSQLiteOpenHelper* que crean las tablas de la base de datos local se han escrito de forma intencionada para que no exista ninguna relación entre las tablas. También se han escrito las consultas que hacen las clases del patrón *DAO* teniendo en cuenta esto, anidando consultas en una sola. Esto se ha hecho así porque en el proceso de la sincronización de la base de datos se crea una consulta para cada registro de cada tabla incluidas las relacionales que en la base de datos local sólo juegan dicho rol de forma conceptual. Todo lo opuesto ocurre en la base de datos remota donde sí se han creado las tablas de forma que estas estén relacionadas (algunas tienen claves foráneas no sólo de forma conceptual como en la aplicación Android), pero eso ya se ha explicado antes en el capítulo análogo a este dedicado al servicio Web desarrollado y desplegado en una máquina servidora.

8.3.2 Subsistema de entrada y salida

Es el subsistema que implementa la lógica necesaria para disponer de unos caso de uso básicos sobre el manejo de ficheros en el sistema de ficheros del dispositivo en el que se ejecuta la aplicación y el uso de la *API* de preferencias.

Cuenta con la clase *GestorBasicoFicheros* que implementa un método para eliminar un fichero a partir de una dirección URL proporcionada y un método que comprueba la existencia del directorio que recibe como argumento para intentar crearlo sólo si no existe. El primer método asume que la dirección URL es una dirección local que apunta a un fichero que existe en el dispositivo. Este método no realiza validaciones sobre la dirección URL (la reciba como *String* o como objeto de tipo URL) porque eso es algo que le corresponde a la lógica de negocio. Simplemente intenta hacer el borrado a partir de la dirección y retorna el estado de éxito o fracaso resultado de realizar la operación.

La clase *GestorBasicoImagenes* es un gestor más específico que *GestorBasicoFicheros* ya que se especializa en los ficheros y formatos relacionados con las imágenes. Cuenta con una constante para definir el directorio del sistema de ficheros compartido del dispositivo en el que se van a almacenar las capturas de pantalla. Implementa los métodos que permiten almacenar un *Bitmap* recibido como un fichero en formato imagen PNG en el directorio especificado por la constante. Utiliza el método *getExternalStorageDirectory()* para obtener el directorio del sistema de ficheros compartido primario, después con ayuda del *GestorBasicoFicheros* comprueba si ya existe dentro de este el directorio especificado en la constante para en caso contrario crearlo y finalmente guarda el *Bitmap* en el sistema de ficheros como un fichero de imagen. El nombre del fichero se genera tomando la hora del sistema en el momento que este método es llamado.

Dadas los escasos y sencillos casos de uso que implementan estas clases y que el uso que hace el sistema de estas no es para leer o escribir información que se muestre al usuario ni se utiliza para ser procesada en la lógica de negocio no tiene sentido utilizar el patrón *DAO* en esta ocasión. Además, las clases y métodos utilizados de la *API* Android para implementar estas clases ya proporcionan abstracción del tipo de sistema de ficheros del dispositivo.

Finalmente, este subsistema cuenta con la clase *GestorAPIpreferencias* que al estilo de las anteriores implementa métodos básicos en relación con la *API* de preferencias. Aunque su uso típico es junto a una actividad de preferencias generalmente para los ajustes globales de la aplicación, también pueden usarse para guardar valores primitivos de datos en forma de pares clave-valor. Ya se han comentado al respecto algunos aspectos en subapartados anteriores. Sin entrar mucho en detalle, Android se encarga de almacenar dichos pares clave-valor en un fichero y se pueden utilizar las preferencias con distintos grados de visibilidad en referencia a los permisos que Android establecerá para ese fichero. El modo que se suele usar es el privado que restringe el acceso a la propia aplicación. Se pueden utilizar las preferencias varias veces para distintos fines pues se creará cada vez un fichero distinto. El uso de las preferencias está abstraído de que por debajo se creen ficheros para almacenar los pares clave-valor.

En la aplicación se han utilizado las preferencias para almacenar los metadatos de la descarga en curso de un recurso, es decir, los datos que proporciona el *Gestor de Descargas* de Android que se almacenan temporalmente mientras se completa la primera parte de la descarga

y que se necesitan para llevar a cabo la segunda parte de la descarga. Volviendo a la clase que se estaba explicando, esta implementa métodos generales para realizar operaciones con las preferencias como el de recuperar un dato concreto, escribir varios datos y el de eliminar completamente unas preferencias. Para ello, estos métodos reciben un identificador de preferencias para realizar la operación que implementan sobre unas concretas, o sea, que sea seleccionable las preferencias sobre las que va a operar. En el caso de obtener un dato también se cuenta con un argumento para especificar la clave del dato que se quiere obtener de unas preferencias. En la aplicación se utilizarán para almacenar el identificador de descarga de la descarga en curso del fichero asociado a un recurso, para escribir los metadatos de la descarga y para eliminar las preferencias porque se van a utilizar de forma temporal.

8.3.3 Controlador de la capa modelo

Es el controlador (de fachada) específico de la capa modelo y su finalidad es la de recibir todas las peticiones que llegan desde fuera de la capa, coordinar los subsistemas que forman la capa para llevar a cabo la petición recibida y finalmente, si se genera un resultado para la petición retornarlo a quién ha hecho la solicitud. En este caso, necesita recibir un objeto de tipo *Context* con el contexto de la aplicación ya que este se necesitará para la base de datos pues la clase *SQLiteOpenHelper* lo hace necesario. Se recibe el contexto de la aplicación y no el de una actividad concreta porque el primero es válido mientras la aplicación se encuentra en ejecución y, por el contrario, el contexto de una actividad podría dejar de valer pues esta podría ser destruida a lo largo de la ejecución de la aplicación.

El controlador cuyos métodos se encargarán de llamar en el orden apropiado a varios métodos de los diferentes subsistemas de la capa con los argumentos correctos se encarga también de que cualquier error que se origine en los métodos a los que llama, es decir, que cualquier excepción que puedan lanzar los métodos a los que llama se encapsule o dé origen a una excepción del tipo propio que define la capa con el mensaje de error adecuado y el identificador numérico de error asociado a ese error. Esto ya se había comentado anteriormente, pero se debe volver a aclarar que la llamada desde distintos métodos del controlador a un mismo método que genera el mismo error en caso de producirse el error debe dar lugar a una excepción del tipo definido en la capa con el mismo mensaje de error, pero con distintos identificadores numéricos.

Este controlador implementa la interfaz *InterfazControladorModelo* y por ello debe como mínimo implementar los métodos declarados en esta.

El controlador implementa métodos para obtener cualquier objeto o lista de objetos de los tipos definidos por las clases del patrón *DTO* a partir de los métodos de las clases del patrón *DAO*. Se encarga de generar las listas a partir de los mapas que estas clases proporcionan puesto que recibe solicitudes que esperan que se les devuelva una lista en lugar de un mapa e internamente los subsistemas de la capa a la que pertenecen trabajan con mapas. Esto es algo que caree de importancia porque en este caso son intercambiables las estructuras de lista y

mapa. Estos métodos del controlador reciben como argumentos los valores concretos de los identificadores (ya descritos) que después pasa a las clases del patrón *DAO* para obtener de la base de datos un objeto o lista concreta, es decir, filtrar de entre todos, los objetos concretos que se desean obtener. Por ejemplo, un recurso simple concreto y no otro, o la lista de los recursos de una sección y módulo concretos.

El controlador cuenta con métodos privados que hacen uso de *Java Generics* para mejorar la implementación y estructura de los métodos públicos declarados en la interfaz.

El controlador también cuenta con otros métodos para atender las solicitudes de registro de un recurso cuya primera parte de la descarga se ha completado, borrar este registro junto al fichero que se indicaba en dicho registro, guardar datos en unas preferencias, obtener el identificador de descarga que se haya guardado antes en unas preferencias, eliminar unas preferencias, guardar el *Bitmap* que se recibe en el sistema de ficheros, y proporcionar estructura a información en crudo (*raw*) que se recibe para luego ejecutar una transacción sobre la base de datos.

A fin de cuentas, cuenta con métodos para atender todas las solicitudes exteriores que tienen como fin utilizar uno de los casos de uso que ofrece al resto del sistema esta capa.

La misión del controlador a veces puede ser tan simple como redireccionar una solicitud a la parte correcta de la capa y retornar, si lo hubiera, el resultado que se genere. Con lo de que se encarga de coordinar se refiere a, por ejemplo, en el método que elimina el registro de un recurso como local, este implica que no se debe intentar borrar el fichero del recurso si no se ha borrado con éxito antes de la base de datos el registro. Como tal, el controlador no está aportando ninguna funcionalidad, simplemente está utilizando las que distintas partes de la capa ya ofrecen, en el orden y momento adecuados. Por otro lado, en este ejemplo concreto, otro pequeño detalle que se debe mencionar es que solicitar el borrado de un fichero implica que se vaya a intentar borrar no que se garantice que se logre. Por ello, el método que ejecuta el intento de borrado puede de alguna forma (retorno de un valor concreto y/o uso de las excepciones) indicar el resultado de la operación, especialmente en caso de error. En este caso sí es tarea del controlador gestionar ese error y decidir si realizar alguna acción adicional, si omitirlo o si notificarlo a quien le hizo la solicitud original. En el caso concreto de esta solicitud, que acaba desencadenando en el intento de borrado, en caso de error la gestión del error que realiza el controlador es omitir el error. Esto se ha hecho así puesto que no importaría que no se borre el recurso porque no impediría en el futuro la descarga de un fichero con el mismo nombre ya que las descargas en esta aplicación sobrescriben. Esto es desconocido en la capa modelo la cual toma este como comportamiento predeterminado sólo para estas solicitudes sin saber la razón que hay detrás, por lo que sigue sin saber cosas de otras capas que no necesita saber ni comprender (y por ello no supone un problema).

8.4 Lógica de negocio

Es una de las capas que componen al sistema y su finalidad va a ser la de realizar operaciones sobre la información estructurada que recibe, es decir, procesar la información para

ofrecer nuevos casos de uso y funcionalidades al resto del sistema. La información que recibe se la proporciona el controlador principal y esta puede provenir directamente de la vista (datos introducidos por el usuario), del modelo (datos almacenados en bases de datos y ficheros entre otros) o de ambas. La información que recibe o son datos de tipo primitivo o es información estructurada cuya estructura conoce. La capa, con esta información, realiza varias operaciones para procesarla y generar un resultado. Ese resultado es otra nueva información generada a partir de la recibida (también se considera como nueva la información original tras ser filtrada o seleccionada). El resultado lo proporciona a quién le hizo la solicitud de procesado (controlador principal) y proporcionó junto a ella la información a procesar. La nueva información, el controlador principal podrá, entre otros, optar por mandarla al modelo para un cambio de estructura de información y/o para que la almacene, mandarla a la vista para mostrársela al cliente o ambas. También puede hacer combinaciones de ello como mandarla al modelo para cambiar la estructura pero que no la almacene y al tenerla de vuelta de nuevo con la nueva estructura mandarla a la vista. Aún mejor, podría mandarla al modelo para que la quite la estructura y así pueda después mandarla a la lógica de conexión quién podrá transmitirla a través de Internet a otro sistema.

Respecto a la estructura de la información, es importante mencionar un par de detalles. El primero es que la lógica de negocio siempre generará información con estructura como resultado de procesar otra información también con estructura. Sí puede ocurrir que la nueva información resultante adquiera otra estructura como resultado del procesado. El segundo es que no se debe confundir que la información adquiera otra estructura como resultado de un procesado con que la información adquiera otra estructura como resultado de una conversión entre formatos no equivalentes y con que la información adquiera otra estructura como resultado de una conversión entre formatos equivalentes. Los dos primeros casos son tarea de la lógica de negocio y lo segundo es tarea del modelo. Se diferencian bien los casos cuando se piensa en qué puede y sólo conoce cada capa. Los dos primeros casos se pueden ejemplificar bien con ejemplos de criptografía. En el primero, la lógica recibe un recurso y aplicando cifrado simétrico algunos de sus campos originales han pasado a ser un sólo campo cifrado y, además, también añade nuevos campos que no existían en la estructura original. En el segundo caso, la lógica de negocio recibe un certificado y otra información a parte, y con ello genera un nuevo certificado de otro tipo más avanzado. El tercer caso se puede explicar con la existencia de una estructura de información y otra que extiende a la primera estructura. De esta forma el modelo podría recibir del controlador información con la estructura extendida para que el modelo devuelva la información con la otra estructura reducida. Este último caso suele ser más raro que aparezca, pero permite acabar de explicar el concepto que se pretendía comentar.

Esta capa recibe del controlador principal y a través de su propio controlador las peticiones que debe llevar a cabo. La capa implementa su propio tipo de excepciones y un *Enum* asociado con todas las posibles causas que pueden generar que se lance este tipo de excepción. Todos los errores que se puedan dar en esta capa se encapsulan en su tipo de excepción propio antes de permitir que lleguen al controlador principal.

8.4.1 Lógica de recursos simples

Se trata de un apartado de la lógica de negocio formado por dos clases. No constituye una subcapa o subsistema independiente, pero por organización se encuentra en un paquete (directorio en los proyectos *Java*) propio.

La clase *Recurso* define un constructor que recibe como argumento un objeto de tipo *RecursoVO* que guarda en un atributo de clase. La clase implementa métodos *getter* y *setter* que redireccionan a los *getter* y *setter* de la clase *RecursoVO*. La diferencia es que la clase *RecursoVO* cuenta con estos métodos para todos sus atributos de clase y la clase *Recurso* sólo para aquellos que se quiera permitir (puede implementarlos todos, pero sólo algunos no tendrán visibilidad privada). Por ejemplo, puede permitir a otra clase que obtenga el identificador del recurso, pero no que lo establezca.

El constructor procesa la dirección URL del recurso llevando a cabo operaciones de validación. Si la validación falla el constructor lanzará una excepción y no se llegará a crear la instancia. Aunque con alta probabilidad se sabe de antemano que la dirección URL que se valida es correcta porque ya se ha validado antes en este u otro sistema, no se debe por ello dejar de volver a comprobar la validez de la dirección. Estrictamente, la clase no tiene por qué saber que se ha validado antes en otra parte y por ello no debe fiarse de que sea válida. Para procesar la dirección URL en forma de validación la clase cuenta con un método de acceso estático llamado *validarURL()*.

La clase cuenta con otros métodos, como *obtenerDetallesRecurso()* que permite obtener un resumen del recurso (lista *String* con el título y la descripción). La clase cuenta con un método que permite establecer un objeto del tipo *RecursoVersionVO* en un atributo de clase. El atributo de clase sólo tendrá valor en aquellos recursos registrados como locales en el sistema. Así pues, la clase cuenta con dos métodos para obtener la dirección URL, uno para la dirección remota del recurso que toma del objeto de tipo *RecursoVO* y otro para la dirección local del recurso que genera a partir del nombre del fichero asociado al recurso que toma del objeto del tipo *RecursoVersionVO*. Como antes, la clase implementa algunos métodos *getter* y *setter* relacionados con la clase *RecursoVersionVO*.

Finalmente, tiene un método que recibe como argumento la versión con la que cuenta el mismo recurso en el sistema remoto para compararla con la que hay en el objeto de tipo *RecursoVersionVO* y retornar con un resultado booleano la existencia o no de una actualización del recurso. Este dato se guarda en un atributo de clase y cuenta con su *getter* y *setter*. El argumento que recibe el método, llega a la capa desde el controlador principal, el cual, a su vez, lo obtiene del siguiente modo: El controlador principal solicita a la lógica de conexión que obtenga cierto tipo de información remota, después pasa al modelo lo que le retorna de la lógica de conexión para que proporcione estructura a la información recibida y finalmente el controlador principal pasa a la lógica de negocio lo que le retorna el modelo. Este es en general el procedimiento que se sigue para obtener información de un sistema remoto (servicio Web con *API REST*) que acaba llegando a la lógica de negocio. Es más, siguiendo, la nueva información resultante del procesado de la lógica de negocio podría acabar a través del controlador principal en cualquiera de las otras capas del sistema, en un sistema remoto, o en una combinación y/o recorrido por varios de estos. Son múltiples las posibilidades y dependerá en cada caso de lo que se necesite hacer.

Esta clase es un ejemplo de que, lo que en el modelo se gestiona y almacena como dos estructuras distintas de información, a nivel lógico, o sea, en términos de clases software a partir de elementos del dominio, puede corresponder a una sola clase.

La otra clase de nombre *ContenedorRecursos* es un contenedor o almacén, es decir, una colección de objetos del tipo *Recurso*. Guarda en un atributo de clase una lista de instancias de objetos de este tipo e implementa los métodos con comportamiento que se refieren al conjunto o colección más que al recurso individual. Un recurso individual no tiene constancia de la existencia de otros recursos por lo que no puede implementar comportamiento que implique conocer la existencia y detalles de varios recursos individuales.

Su constructor recibe como argumento una lista de objetos del tipo *RecursoVO* y a partir de ella construye la lista de objetos del tipo Recurso que almacena en el atributo de clase.

Cuenta con un método *obtenerListaRecursos()* que genera un mapa en el que la clave es el identificador del recurso y el valor es la lista de cadenas de texto que se recibe al llamar al método *obtenerDetallesRecurso()* sobre el recurso.

Otros métodos permiten obtener el identificador del recurso con el menor identificador, una lista con los identificadores de los recursos, una lista con los títulos de los recursos y un objeto de tipo *Recurso* a partir de su identificador.

8.4.2 Lógica de módulos

Se trata de un apartado de la lógica de negocio formado por dos clases. No constituye una subcapa o subsistema independiente, pero por organización se encuentra en un paquete (directorio en los proyectos *Java*) propio.

Las clases son *Modulo* y *ContenedorModulos*, y en la línea de las clases del subapartado anterior, realizan la implementación análoga para sus constructores y sus métodos *obtenerDetallesModulo()* y *obtenerListaModulos()* de respectivas clases.

8.4.3 Lógica de recursos compuestos

Se trata de un apartado de la lógica de negocio formado por dos clases. No constituye una subcapa o subsistema independiente, pero por organización se encuentra en un paquete (directorio en los proyectos *Java*) propio. Estas clases siguen el modo de implementación análogo al explicado en el subapartado *Lógica de recursos simples*.

El constructor de la clase *RecursoCompuesto* recibe como argumento un objeto de tipo *ContenedorRecursos* con los recursos simples que lo componen. Se guarda esta instancia en un atributo de clase.

Además, cuenta con un método para obtener el recurso simple y su identificador del recurso simple representativo del recurso compuesto. Para saber cuál es el recurso simple representativo de un recurso compuesto se ha determinado que será el de identificador más

bajo de todos. En un futuro, como mejora del dúo de sistemas (aplicación Android y servicio Web), se podría soportar que se almacenara cuál es el recurso simple representativo lo que permitiría que en vez de estar determinado por el de menor identificador fuera algo configurable. Otros métodos son redirecciones al objeto de tipo *ContenedorRecursos* para permitir obtener listados de los identificadores y títulos de los recursos simples.

En la clase *ContenedorRecursosCompuestos* no hay nada nuevo salvo el también segundo argumento en el constructor que es una lista de objetos del tipo *ContenedorRecursos*.

8.4.4 Lógica de elementos remotos

Se trata de un apartado de la lógica de negocio formado por dos clases. No constituye una subcapa o subsistema independiente, pero por organización se encuentra en un paquete (directorio en los proyectos *Java*) propio. Estas clases siguen el modo de implementación análogo al explicado en el subapartado *Lógica de recursos simples*.

La clase *TablaRemota*, en su constructor, a partir del identificador del objeto de tipo *ControlVersionesVO* que recibe como argumento obtiene el tipo de tabla del *Enum TablasBaseDatos* y lo guarda como atributo de clase. La clase cuenta con el método *getter* para este atributo.

El *Enum* antes mencionado representa a las diferentes tablas que existen en la base de datos local de la aplicación. Cuenta con dos atributos de clase que son dos cadenas de texto para almacenar el nombre de la tabla y el nombre del tipo de operación que debe utilizar la lógica de conexión por el método *HTTP POST* para solicitar y obtener del servicio Web dicha tabla. Aunque no es una tabla, el *Enum* contiene una entrada para referirse a la base de datos por lo ya explicado a final del subapartado *Patrón DTO* sobre el estado de sincronización. Como se dijo en el subapartado *Clases Enum*, este *Enum* se encuentra en un paquete (directorio) en la raíz del directorio que contiene el código *Java* del proyecto.

La clase *ContenedorTablasRemotas* implementa el método *obtenerVersionTabla()* para obtener la versión de una tabla según el argumento del tipo del *Enum* anterior. El método *compararVersionTabla()* realiza las mismas tareas que el método anterior pero adicionalmente compara la versión que recibe como argumento con la que toma de la tabla seleccionada con el otro argumento de tipo el *Enum*.

8.4.5 Lógica de calculadoras

Se trata de un subsistema de la capa lógica negocio que implementa toda la lógica y los algoritmos para realizar todos los cálculos posibles a partir de los datos médicos introducidos desde la vista por el usuario. Cuenta con su propio controlador y presenta una interfaz que implementa el controlador. Este subsistema también cuenta con su propio tipo de excepción.

8.4.5.1 Rango de un parámetro de entrada

La clase *RangoParametroEntrada* permite modelar un rango para un parámetro de entrada. Cuando se habló en la capa vista de los campos de texto y controles mediante los cuales se introducían datos médicos se aclaró qué parte de su validación le correspondía a la vista y cuál a la lógica de negocio.

Los rangos tienen un doble objetivo, por un lado, sirven para implementar la validación de los parámetros de entrada por parte de la lógica de negocio y por otro lado permiten mapear los valores introducidos con otros determinados establecidos.

Para especificar un rango, se utiliza la notación de intervalos donde los símbolos *(*, *[*, *)* y *]* se corresponden respectivamente con los valores enteros 2, 4, 3 y 5. Los rangos se especifican con dos de estos símbolos, dos valores numéricos (enteros o decimales) y un valor de salida. Ese valor de salida, del mismo modo que si de un cuantificador se tratara es el valor que tomaría el parámetro de entrada si el valor introducido por el usuario para dicho parámetro estuviera contenido en el intervalo especificado en el rango. Por ejemplo, si el usuario introduce el valor *1,3* para un parámetro para el que uno de los rangos que lo definen es *[0.9, 2] -> 1* se habrá comprobado que el valor introducido es correcto (ha pasado la validación porque el valor introducido está contenido en alguno de los intervalos de los rangos que definen a dicho parámetro) y para realizar los cálculos se utilizará el valor *1* para este parámetro. Si se hubiera introducido el valor *1.7* también se utilizaría el valor *1* para realizar los cálculos. Si se hubiera introducido el valor *2.4* si no existiera otro rango el valor introducido no sería válido y se abortaría la tarea de cálculo correspondiente.

Se ha reservado el valor entero *-1* como comodín de forma que permite establecer intervalos con un sólo extremo cuando se utiliza el valor *-1* para sustituir al valor de uno de los extremos. También permite establecer que el valor de salida sea el mismo que el valor introducido.

Desde el punto de vista de la implementación, un rango se configura utilizando valores numéricos. Dos para los símbolos del intervalo, dos para los extremos del intervalo y uno para el valor de salida. Así pues, con los valores numéricos *4*, *1.1*, *-1*, *3* y *-1* se está especificando el rango *[1.1, +∞) -> valorIntroducido*. Si se introduce el valor *3.7* el valor de salida será *3.7*, y si se introduce el valor numérico *9.1* el valor de salida será *9.1*. Ambos valores pasarán satisfactoriamente la validación.

La clase entonces cuenta con un constructor que recibe como argumento un array de tipo *double* con los cinco valores numéricos que definen al rango. También cuenta con el método *validarValorDelParametro()* que recibe como argumento el valor introducido para el parámetro y retorna el resultado de la validación con un valor booleano. Su último método permite obtener el valor de salida que configura al rango, es decir, el valor de salida que el rango define o el valor *-1*. Cuando se usa el valor *-1* para configurar el rango, este método no retorna el valor introducido en el parámetro de entrada, sino que devuelve ese valor *-1*.

8.4.5.2 Contenedor de rangos de un parámetro de entrada

La clase *ContenedorRangosEntradaParametros* sigue el modo de implementación análogo al explicado en el subapartado *Lógica de recursos simples*, es decir, es un contenedor o almacén, o colección de objetos del tipo *RangoParametroEntrada*. Guarda en un atributo de clase una lista de instancias de objetos de este tipo e implementa los métodos con comportamiento que se refieren al conjunto o colección más que al rango individual.

El constructor recibe una matriz de valores numéricos *double* para construir a partir de los cinco valores numéricos de cada fila una instancia de la clase *RangoParametroEntrada* que se almacena después en la lista del atributo de clase. A nivel conceptual se describe como una matriz, pero dado que se conoce que la longitud de las filas es fija y es cinco, en la implementación el constructor recibe un array en lugar de una matriz donde cada fila se ha concatenado a la anterior. El constructor cuenta con la lógica para extraer (*split*) cada fila del array.

La clase implementa un método *validarParametro()* que recibe el valor introducido para un parámetro y llama al método *validarValorDelParametro()* de cada rango que contiene en la lista del atributo de clase. Cuando el valor del parámetro que se ha recibido se encuentra contenido en uno de los rangos (*match*) se para de comprobar los restantes, se almacena cuál es el rango que ha resultado la operación exitosa y se da el valor del parámetro introducido por válido. En caso de que el valor introducido no se encuentre contenido en ningún rango se lanzará una excepción.

El otro método que implementa retorna el valor de salida que configura al rango y que se obtiene con el método correspondiente de la clase *RangoParametroEntrada*.

8.4.5.3 Parámetro de entrada

La clase *ParametroCalculadora* modela un parámetro médico de la calculadora y se apoya en la clase *ContenedorRangosEntradaParametros* y en el *Enum* de nombre *UnidadesCalculadora*.

El *Enum* define todas las unidades que soporta el sistema para la calculadora tanto del sistema internacional como no. El *Enum* cuenta con tres atributos de clase, un identificador de la unidad, un *flag* booleano que indica si es una unidad internacional y un valor numérico *double* con el factor que habría que aplicar a un valor expresado en esa unidad para convertirlo a la unidad asociada. La unidad asociada de una unidad internacional es su equivalente unidad no internacional y la unidad asociada de una unidad no internacional es su equivalente unidad internacional.

Como ya se ha introdujo en apartados anteriores, se puede dar la situación de que el mismo valor en dos parámetros médicos de entrada diferentes con la misma unidad internacional al convertirse a unidades no internacionales pueden corresponderse con dos valores distintos, aunque tengan la misma unidad no internacional. Para solventar este problema, se definen las unidades que presenten esta situación particularizadas al parámetro médico de entrada tantas veces como sea necesario. Así pues, se define la unidad internacional *umol/L* varias veces, por ejemplo, *UMOL_L_BILIRUBINA* y *UMOL_L_CREATININA*, con factores de conversión distintos y lo mismo ocurre con su unidad asociada no internacional *mg/dL* que se define varias veces también con factores de conversión distintos. Para implementar este mecanismo, el valor del identificador que se asigna a cada entrada del *Enum* se encuentra suficientemente separado para cada unidad. También se puede dar la situación en la que una unidad internacional se corresponda con dos unidades internacionales diferentes. Es también el caso de la unidad internacional *umol/L* con la unidad no internacional *mmol/L* para el parámetro médico sodio. Para solucionar estas situaciones, aplicando el mismo razonamiento se acaba configurando una tercera entrada en el *Enum* de nombre *UMOL_L_SODIO* que tendrá asociada la entrada *MMOL_L_SODIO*. Es importante ver, que en el *Enum* sólo se encuentran definidas todas las unidades que puede soportar el sistema con todas sus variaciones particularizadas definidas, pero no tiene las relaciones entre todas las unidades que define. En el siguiente subapartado se explica el motivo de esto. Se debe mencionar el detalle de que se ha reservado el identificador *-1* para la unidad definida en el *Enum* de nombre *SIN_UNIDAD* que va a contar con el *flag* de unidad internacional con valor *true* y un factor de conversión de valor *1*. Se trata de la única unidad que no tiene otra asociada (o su unidad asociada es ella misma).

El *Enum* implementa un método para obtener la unidad a partir de su identificador y los métodos *convertirUnidadEquivalenteInternacional()* y *convertirUnidadEquivalenteNoInternacional()* que reciben como argumento el valor numérico que van a retornar convertido o no según la unidad desde la que se llamen. El primero no realiza ninguna conversión si la unidad desde la que se llama es una unidad internacional y en caso contrario realiza la conversión devolviendo el valor multiplicado por el factor de conversión que tiene la unidad que ha llamado al método. La implementación del segundo es análoga a la del primero.

Volviendo a la clase *ParametroCalculadora*, su constructor recibe como argumentos el valor numérico introducido para el parámetro médico de entrada, la matriz de configuración de rangos para ese parámetro, el array con el par de identificadores de la unidad internacional y la unidad no intencional asociadas al parámetro, un *flag* booleano que indica si el valor recibido del parámetro médico de entrada está en unidades internacionales y un *flag* booleano que indica que se debe forzar la conversión a unidades no internacionales. El segundo argumento sería la matriz conceptual implementada como array de la que se ha hablado en el subapartado *Contenedor de rangos de un parámetro de entrada*.

La lógica de la calculadora trabaja por defecto con unidades internacionales, de forma, que convierte a unidades internacionales todos los valores de los parámetros médicos de entrada. Para realizar la conversión, utiliza el *Enum* explicado antes. Se debe recordar que se aplica la conversión sobre todos los valores, aunque ya se encuentren en unidades internacionales. Por si, como ocurre en HepApp, algún algoritmo de la calculadora tiene definidos los rangos que usa en unidades no internacionales, sólo se debe activar el *flag* del

último argumento del constructor de esta clase para que para ese parámetro en concreto se fuerce la conversión, pero a unidades no internacionales en vez de a unidades internacionales.

La clase cuenta con un método *validarParametro()* que es una redirección al de mismo nombre de la clase *ContenedorRangosEntradaParametros* y un método *obtenerValorSalida()* que retorna el valor de salida mapeado que se da por supuesto que está en las unidades correctas o si el método de la clase *ContenedorRangosEntradaParametros* al que llama retorna *-1* en lugar de un valor de salida mapeado, este método retorna el valor del parámetro médico de entrada que se recibió como primer argumento y se almacenó en un atributo de clase tras realizar la conversión de unidades establecida por los *flag* que también recibe como argumentos el constructor de dicha clase. Es importante destacar que se guarda como atributo de clase el valor convertido del parámetro médico de entrada que podrá o no coincidir con el valor original que se recibió como argumento en el constructor. Como sólo este es el que se guarda, sólo es ese el que el método *obtenerValorSalida()* podría retornar si no hay configurado un valor de salida mapeado.

8.4.5.4 Configuración de parámetros de entrada

Se crea una clase de constantes que pretende centralizar toda la configuración de la lógica de calculadoras. En esta clase es en la que se especifican las matrices (sólo a nivel conceptual porque están implementadas como array) con la configuración de los rangos de un parámetro médico de entrada, en definitiva, configurar los parámetros. Es por ello, que por cada parámetro se especifica una matriz. Para un mismo parámetro se pueden crear varias matrices de configuración particularizadas al algoritmo de la calculadora al que se refieren. Esto se debe a que a veces el valor de un mismo parámetro se utiliza para calcular varios algoritmos que pueden necesitar distintos rangos de ese mismo parámetro. De hecho, esta es una situación habitual en HepApp y algunos parámetros cuentan con cuatro o más matrices particularizadas. Lo importante no es que haya varias matrices por cada parámetro, sino que sólo se use una y la adecuada en cada algoritmo distinto.

Se debe mencionar, que ya en la vista, en el mapa en el que se guardan los valores acumulativos a lo largo de las pantallas con formulario se realiza un mapeo de los controles *Button* a números enteros. Por ello, por ejemplo, un control con el valor *None* se almacena como un entero en la estructura que almacena que se ha seleccionado ese control de todos los del grupo al que pertenece. La lógica también recibe y utiliza valores numéricos, por ello se configuran con intervalos parámetros que en la realidad no toman valores numéricos. Que se usaran los mismos valores en la vista y en la lógica de negocio se podría entender como un pequeño acoplamiento entre ambas, pero, a fin de cuentas, tanto en la vista como en la lógica de negocio se están dando las relaciones ya deducidas lo cual evita a cada capa que deba deducirlas. Ninguna capa podría deducir estas relaciones sin conocer detalles sobre la otra capa, pero sí puede utilizar las relaciones que ya le vienen dadas sin necesidad de tener que entenderlas. La vista tiene las relaciones entre los controles y los valores a los que debe mapearlos y la lógica de negocio tiene las relaciones entre los valores que recibe y los parámetros a los que debe mapearlos. La situación se mejora gracias al uso de constantes o

Enum puestos en común en lugar de valores enteros. En cualquier caso, para romper el acoplamiento que pudiera existir entre las dos capas existe precisamente el controlador principal quien debe realizar la tarea de adaptar los valores de una capa a otra en cualquiera de los dos sentidos que se necesitara. Una de las finalidades del controlador principal como se ha explicado es el de lograr independencia entre capas a cambio de un gran acoplamiento de cada capa con él. Así pues, la solución óptima es que el controlador principal medie entre ambas capas y que se utilicen constantes o *Enum* en ambas para facilitar la tarea de mediación. Un detalle importante es que, aunque las capas utilicen los mismos valores, esto no evita que el controlador tenga que seguir haciendo la tarea de mediar entre ambas. Sólo se han utilizado los mismos valores por una cuestión de comodidad para el desarrollador, aunque se vuelve a recalcar que se deberían haber empleado otros elementos software en lugar de valores enteros.

Volviendo a la clase de constantes, se declaran arrays de tipo entero de longitud dos para configurar para cada parámetro las relaciones entre unidades internacionales y unidades no internacionales a partir de los identificadores de las unidades definidas en el *Enum* de nombre *UnidadesCalculadora* que se ha explicado en el subapartado anterior. El motivo para que estos array no se hayan declarado en el propio *Enum* es el de centralizar toda la configuración de la lógica de calculadora en una sola clase de constantes y que se independizara al *Enum* de los parámetros porque estas relaciones están vinculadas a cada parámetro médico de entrada.

8.4.5.5 Algoritmo de calculadora genérico

La aplicación HepApp implementa un buen número de algoritmos médicos que obtienen resultados a partir de los datos que el usuario introduce en los formularios de las pantallas de las calculadoras. Dado el gran número de algoritmos que se deben implementar en la lógica de negocio se decide utilizar una vez más las posibilidades de la herencia para separar las partes que puedan tener en común todos estos algoritmos en una clase más general de la que luego hereden las clases específicas que implementan los algoritmos. En este caso, el punto más fuerte de esta clase abstracta de algoritmo de calculadora es el de poder generalizar la gestión de los parámetros de entrada a través de todas las clases vistas hasta ahora en varios de los subapartados anteriores.

Los algoritmos de la calculadora van a seguir la estructura general de contar con uno o varios constructores y un método para realizar el cálculo del algoritmo. Esto no quita para que algún algoritmo implemente métodos propios que necesite utilizar.

La clase general recibe como argumento en su constructor un mapa con los valores de los parámetros médicos de entrada que almacena como atributo de clase. El mapa tiene por clave una cadena de texto y por valor un valor numérico *double*. En el subapartado anterior se ha mencionado la importancia de la tarea del controlador de mediar entre dos capas independientes y lo óptimo que es facilitarle esta tarea mediante el uso de constantes o *Enum*.

En este caso, la interfaz que ofrece el subsistema lógica de calculadoras que implementa el controlador propio de este subsistema declara las constantes de tipo cadena de texto con las

que va a reconocer a qué parámetro de entrada se refiere cada valor que recibe este subsistema mediante el mapa que se acaba de explicar. Por su parte, ya se vio que la capa vista también hace lo propio con su interfaz de constantes descrita en el subapartado *Actividad calculadora general*. No se ha considerado relevante que la interfaz con las constantes en la lógica de negocio sea la del subsistema en vez de la interfaz del controlador de la capa lógica de negocio. Se ha implementado por simplicidad que el controlador principal acceda directamente a las constantes de esta interfaz, aunque estrictamente hablando están correctamente ubicadas porque deberían volver a estar presentes (repetidas o cambiadas) en la interfaz del controlador de la capa (que es la única interfaz que el controlador principal debería ver) y actuar este controlador de mediador entre el exterior de la capa y el subsistema de la capa. Lo análogo ocurre en la vista.

Como ya ocurría en el subapartado anterior, se han utilizado por comodidad del desarrollador las mismas constantes en la vista y en la lógica de negocio, aunque esto no evite que se tenga que implementar igualmente que el controlador principal lleve a cabo la tarea de mediar entre ambas capas independientes adaptando las constantes que declara una capa a las que declara la otra. Se puede decir que es lo mismo que ocurre muchas veces en los métodos constructores donde los argumentos que recibe se asignan a atributos de clase de igual nombre y se utiliza *this* para diferenciar a qué se refiere el nombre común en cada caso.

La clase algoritmo de calculadora general declara el método abstracto *calcularAlgoritmo()* que recibe como argumento un array con las claves del mapa de los valores de los parámetros de entrada que también se utilizarán para construir el mapa que tendrá las instancias de objetos de tipo *ParametroCalculadora*.

Siguiendo las ideas de implementación utilizadas en la vista, el constructor de una subclase declarará un array y dos matrices de forma local. La intención es que estos elementos sean propios o especificados para cada algoritmo. El array se forma con las constantes de la interfaz de este subsistema, la matriz de tipo *double* tiene en cada fila un array (matriz conceptual) de la clase que centraliza la configuración de los parámetros y la otra matriz de tipo entero tiene en cada fila un array de la clase que centraliza la configuración de las unidades (y los parámetros). Cada elemento del array y cada fila de las matrices deben estar en un orden coherente según al parámetro al que se refieren. Así pues, en el algoritmo *Child Pugh Score (CPS)* y entendiendo las matrices como un array de constantes que son array en vez de elementos individuales, para el parámetro bilirrubina el array tendrá como su primer elemento *InterfazControladorCalculadoras.KEY_ENTRADA_DATO_BILIRUBINA*, la matriz *double* como primera fila *ConfiguracionParametros.configuracionBilirubinCPS* y la matriz entera como primera fila *ConfiguracionParametros.unidadesBilirubin* donde *InterfazControladorCalculadoras* es la interfaz que implementa el controlador de este subsistema y que define las constantes, y *ConfiguracionParametros* es la clase del subapartado anterior que centraliza la configuración de la lógica de calculadoras. Esto se repite por todos los parámetros médicos de entrada que necesita ese algoritmo en concreto. El constructor de una subclase, entre otros argumentos, recibe el mapa de antes que luego pasa al constructor de la superclase.

La clase algoritmo de calculadora general implementa el método *obtenerMapaParametrosEntrada()* que será llamado en el constructor de la subclase tras declarar el array y matrices descritas antes. Este método recibe como argumentos precisamente

el array y las matrices junto al *flag* que indica si los valores recibidos están en unidades internacionales que recibe como argumento el constructor. Con estos argumentos (diferentes en cada algoritmo) y el mapa con los valores de los parámetros médicos de entrada que ya tiene genera el mapa que tendrá las instancias de objetos de tipo *ParametroCalculadora*. Al generarlo puede detectar si falta el valor de algún parámetro médico de entrada necesario para calcular el algoritmo desde el que se ha llamado en este método. Si falta algún valor, notifica el error con la excepción propia de este subsistema, lo cual para la ejecución y se propaga a otras clases hasta aquella que implemente la gestión del error. Generalmente la gestión del error llegará hasta el controlador de calculadoras quien determinará que en vez de notificarlo a quien le ha solicitado calcular el algoritmo (controlador principal) simplemente establecerá el resultado del algoritmo como "sin resultado". La clase implementa el método *obtenerMapaParametrosEntradaForzarUnidadesNoInternacionales()* que realiza las mismas tareas que el método *obtenerMapaParametrosEntrada()* sólo que estableciendo el *flag* que fuerza la conversión a unidades no internacionales (es un alias del otro método cuya diferencia es que establece un *flag* adicional para todos los parámetros que va a procesar). Cada algoritmo llamará en el constructor a uno u a otro según si su lógica trabaja con unidades internacionales (la mayoría) o trabaja con unidades no internacionales (la minoría). El mapa que generan estos métodos se guarda como atributo de clase.

Por otro lado, la clase general implementa un método *validarParametrosEntradaAlgoritmo()* que deberá ser llamado en la subclase al principio del método *calcularAlgoritmo()* y que recibe como argumento el mapa con las instancias de objetos de tipo *ParametroCalculadora*. El método simplemente ejecutará el método *validarParametro()* sobre cada instancia que tiene el mapa que recibe como argumento. Si la validación de alguno de los parámetros falla, como ocurre con los métodos anteriores se lanza una excepción del tipo propio de este subsistema que probablemente llegará hasta el controlador de calculadoras para que en este caso llegue al controlador principal quien lo encapsula como ya se ha comentado para mostrar en la vista un mensaje de error que le indique el problema al usuario. En esta ocasión se trata de un error que provoca una alternativa prevista en el flujo de ejecución del sistema, es decir, es un error esperado que se puede indicar claramente sin ambigüedades al usuario. El método *calcularAlgoritmo()* al final deberá establecer el valor de una cadena de texto y/o un valor numérico con el resultado del cálculo correcto del algoritmo. La clase general cuenta con un método *getter* para cada uno estos dos atributos de clase.

8.4.5.6 Algoritmos de la calculadora

Se crea una clase por algoritmo que soporta la aplicación. Todos ellos van a heredar de la clase de algoritmo general que se ha descrito en el subapartado anterior. En este, se proporcionan bastantes detalles sobre cómo va a ser la implementación de estas clases. En el capítulo *Algoritmos de hepatología* se explica la teoría y la lógica de los algoritmos desde un punto de vista médico y conceptual que después se traslada como implementación a las clases del subsistema lógica de calculadoras.

Respecto a los métodos constructores, en cada clase recibirán a parte del mapa de valores tantos argumentos como valores de los ajustes necesiten para realizar el cálculo. En el caso concreto del criterio que se selecciona en los ajustes, como ocurre en la clase que implementa el algoritmo *Alberta* este cuenta con un constructor por cada uno de los cuatro posibles criterios, de forma que cada constructor recibe como argumento la instancia del tipo o clase que implementa el criterio seleccionado. Además, como algunos algoritmos pueden necesitar no sólo valores de los parámetros de entrada para realizar su cálculo, sino que pueden necesitar el resultado de otro algoritmo, también pueden definir constructores que reciben como argumentos instancias del tipo del algoritmo cuyo resultado necesitan como si se tratara del valor de un parámetro de entrada. Esto sólo es una forma de estructurar y optimizar la implementación, pues las instancias de otros algoritmos que reciben como argumentos en los constructores previamente han sido creadas a partir del mismo mapa de valores de parámetros médicos que siempre se sigue recibiendo en cada constructor. Por ello, cada algoritmo, por lo general, siempre cuenta también con un constructor que sólo recibe el mapa de valores de los parámetros médicos de entrada y los valores de los ajustes que necesita porque a partir del mapa se podría calcular de nuevo el otro algoritmo que necesita. Por ejemplo, es el caso de los algoritmos *CPS* y *CLIP*, donde el algoritmo *CLIP* necesita el resultado del algoritmo *CPS*. Suponiendo que se está en la calculadora completa, el usuario espera que el sistema le muestre el resultado de ambos algoritmos, pero la lógica de calculadora puede optar por calcular el algoritmo *CPS* una vez para mostrar el resultado y volverlo a calcular de nuevo dentro de la clase del algoritmo *CLIP* porque este lo necesita o calcularlo una vez y cuando vaya a calcular el algoritmo *CLIP* a este se le pase utilizando el constructor adecuado la instancia que ya se tiene del algoritmo *CPS*. Un caso semejante ocurre con los algoritmos *MELD*, donde el algoritmo *MELDNa* utiliza el resultado de *MELD* y el algoritmo *5vMELD* utiliza el resultado de ambos.

Las instancias de otras clases de algoritmos o algoritmos de criterios que una clase necesite se guardan como atributos de clase. A parte de que se definan varios métodos constructores según las instancias de otras clases que se necesiten también se puede disponer de forma adicional de métodos *getter* y *setter* para estos atributos de clase. Respecto al método *calcularAlgoritmo()*, este puede ser llamado o no directamente desde el constructor al final. Si por ejemplo se usa el constructor que cuenta con argumentos para recibir todas las instancias de otras clases, este podría llamar directamente al método *calcularAlgoritmo()* porque a diferencia de lo que ocurre si se utiliza un constructor que no cuenta con estos argumentos, tras instanciar la clase no se van a utilizar ninguno de sus métodos *setter* antes de llamar al método *calcularAlgoritmo()*. Si el constructor que no recibe instancias de otra clase llamara directamente al método *calcularAlgoritmo()*, siempre tocaría volver a calcular los otros algoritmos que se necesitan en la propia clase.

Como ya se mencionó, estas clases como subclases de la general deben implementar el método *calcularAlgoritmo()* para que aporte la lógica de cálculo del algoritmo al que se refiere la clase. Esta lógica se debe escribir entre la llamada al método de validación de parámetros en la superclase y el establecimiento de los resultados en los atributos de clase. El valor de cada parámetro médico de entrada que se necesite en el código de este método se obtiene ejecutando el método *obtenerValorSalida()* sobre la instancia correspondiente del mapa con instancias del tipo *ParametroCalculadora*. Para saber qué instancia tomar del mapa para el valor de cada parámetro se puede utilizar el array de claves del mapa que recibe el método

calcularAlgoritmo() como argumento o directamente las constantes de la interfaz *InterfazControladorCalculadoras*.

A parte de todo esto, cada clase que implementa un algoritmo puede implementar métodos propios, como ocurre en el caso del algoritmo *Alberta* donde en vez de un único resultado como tal, este genera adicionalmente hasta dos sugerencias de tratamientos y un esquema. A parte, según el resultado genera una información detallada extra. En definitiva, genera varios resultados y por ello, aparte de establecerlos como atributos de clase en su método *calcularAlgoritmo()*, debe definir los métodos *getter* correspondientes. De nuevo, como algunos de los resultados que genera el algoritmo son imágenes o textos que deberá visualizar la vista a partir de recursos *drawable* (capa oculta más información y el esquema *Alberta*) o recursos cadena de texto en *strings.xml* se vuelve a utilizar el mecanismo de clases *Enum* junto a los controladores funcionando como mediadores entre ambas capas.

8.4.5.7 Controlador de calculadoras

Es el controlador propio del subsistema e implementa la interfaz *InterfazControladorCalculadoras* de la cual ya se ha hablado.

El controlador implementa un método que permite obtener el resultado de un algoritmo concreto. El resultado lo devuelve como una cadena de texto. Lo importante de este método es que a lo que se refiere como resultado es el resultado numérico (o la parte numérica del resultado) que proporciona el algoritmo puesto que este no es diferente en distintos idiomas. Para algoritmos que generan más de un resultado o no se trata de resultados numéricos (o no dependientes del idioma), como es el caso del algoritmo *Alberta*, como ya se ha mencionado, como el algoritmo cuenta con métodos *getter* propios para obtener esos resultados, el controlador también debe contar con métodos análogos que podrán ser una simple redirección a los del algoritmo o algo más si se necesita.

El método que permite obtener el resultado de un sólo algoritmo recibe como argumento un *Enum* a partir del cual, junto a una estructura *switch*, determina cuál de todos los algoritmos es el que se le ha pedido. Para evitar tener que utilizar tantas variables locales, cada una del tipo de un algoritmo diferente, cuando se crea la instancia de la clase que implementa el algoritmo pedido se realiza la operación *casting* a la clase general de algoritmo de forma que la instancia que se almacena de forma local en el método es de este tipo. Esto hace que sólo sea necesario una variable local y no tantas como diferentes algoritmos existan en el sistema. Esta operación de *casting* es posible gracias a la relación de herencia que existe entre la clase general de algoritmos de calculadora y todas las clases específicas que implementan los algoritmos heredando de la general. También, se debe decir que esto es posible en este método pues el resultado que retorna el método es el que se obtiene a partir del método *getter* ya explicado que implementa la clase general de algoritmos de calculadora. Obviamente, este método es incompatible con el algoritmo *Alberta* o sólo podría proporcionar uno de sus resultados numéricos si los tuviera.

Otro de los métodos del controlador permite obtener un mapa con los resultados (numéricos) de todos los algoritmos que sean posibles calcular con los valores de los parámetros médicos de entrada recibidos (puede que el usuario no haya especificado todos los de los formularios de las pantallas de la calculadora). Este método, para ahorrar tareas de cálculo, siempre intentará realizar primero el cálculo de aquellos algoritmos que no necesitan del resultado de otros para luego pasar a calcular los algoritmos que necesitan del resultado de otros algoritmos. Así pues, este método nunca realizará el cálculo del algoritmo *CLIP* antes de haber realizado el del algoritmo *CPS*. Es más, si el cálculo de *CPS* fallara porque falta un valor de un parámetro de entrada ni siquiera intentará calcular *CLIP*. Como ya se ha comentado, para hacer posible esta ejecución optimizada es clave el uso del constructor correspondiente en cada caso de todos los que implementa una clase específica de algoritmo de calculadora. De nuevo, como ocurría en el método anterior, para almacenar localmente algunas de las instancias de algoritmos que se calculan y guardan para reutilizarse en otros algoritmos se realiza la operación de *casting*. A diferencia del método anterior, se debe volver a hacer de nuevo la operación de *casting* para obtener de vuelta el tipo específico en el momento de usar el constructor de un algoritmo, pues los algoritmos reciben en sus constructores instancias de las clases específicas de otros algoritmos y no instancias del tipo general. En cualquier caso, esto tampoco es algo crítico pues la segunda operación de *casting* se podría realizar en la propia clase específica.

El resultado de este método es un mapa que devuelve los resultados de los algoritmos de igual forma que hacía el otro método (cadenas de texto) sólo que utiliza como clave el *Enum* que el otro método recibe como argumento. Este *Enum* es el que se ha mencionado al final del subapartado anterior y lo utiliza el controlador principal en su tarea de mediar entre capas.

Será también tarea del controlador recibir las excepciones del tipo propio que las clases de los algoritmos pueden lanzar, para decidir si las retransmite o si gestiona el error. Cuando la excepción indica la falta del valor de un parámetro médico de entrada decide gestionar el error estableciendo que el resultado numérico del o los algoritmos pedidos es el carácter "-". Cuando el error es que alguno de los valores introducidos no pasa la validación, es decir, está fuera de rango porque ningún rango definido para ese parámetro en un algoritmo concreto contiene el valor introducido, se propaga o retransmite la excepción con lo que esto implica que es abortar la ejecución. Lo bueno en este caso del uso de las excepciones como mecanismo de error es, como se ha dicho, que abortan la ejecución, y por ello, en el método que calcula todos los algoritmos posibles esto implica que el que se produzca un error de este tipo aborta que se intenten calcular los algoritmos que aún estén sin calcular.

Se debe recordar, que los valores introducidos en la vista entre todos los formularios, ya sean valores médicos o ajustes, dado que en la vista se almacenan ambos tipos de valores en mapas separados, a la hora de tener que mandarlos, por comodidad la vista decide concatenar (*append*) ambos mapas y mandarlos de esta forma. Por su parte, la lógica de calculadora también espera recibir todos los valores en un único mapa. El que una capa decida hacerlo no depende de que lo haga la otra, sino que por separado es cada una la que decide que así lo quiere. Se trata entonces de una coincidencia y por ello la tarea de mediación al respecto que deberá hacer el controlador principal será hacer nada (aunque la vista los podría haber mandado separados y la tarea de mediación hubiera consistido simplemente en juntarlos). Esto es algo

típico, ya no por comodidad porque sea mejor usar un único argumento que tener que utilizar varios, sino porque por limitaciones en implementación, a la hora de retornar los resultados de un método, estos sólo pueden retornar un único elemento de información estructurada. A parte de esto, se puede hacer una analogía con el uso de compresores y descompresores a la hora de mandar varios ficheros a través de Internet, pues ya no sólo es buscar la reducción sin pérdidas del tamaño, sino el que se manden todos empaquetados en un único fichero (que también se puede denominar correctamente archivo porque es una colección de ficheros).

Finalmente, respecto a los valores de ajustes recibidos, el controlador cuenta con métodos privados no declarados en la interfaz (no se pueden utilizar desde fuera) que extraen del mapa que se recibe los valores de los ajustes. Los métodos de la interfaz que realizan el cálculo de uno o todos los algoritmos llaman a estos métodos privados si un algoritmo necesita ciertos valores de los ajustes porque así lo indican con los argumentos que declaran en sus métodos constructores. El mapa que se pasa a los constructores de las clases que implementan los algoritmos es el mapa resultante tras que el controlador quite el mapa concatenado con los valores de los ajustes.

8.4.6 Controlador de la capa lógica de negocio

Es el controlador (de fachada) específico de la capa lógica de negocio. Recordando algunas de las cosas ya dichas acerca de los controladores en otras partes de este documento, su finalidad es la de recibir todas las peticiones que llegan desde fuera de la capa que tienen como fin utilizar uno de los casos de uso que esta ofrece al resto del sistema, coordinar las clases y los subsistemas que forman la capa llamando en el orden apropiado a varios de sus métodos con los argumentos correctos para llevar a cabo la petición recibida y, finalmente, si se genera un resultado para la petición retornarlo a quién ha hecho la solicitud.

También se encarga de que cualquier error que se origine en los métodos a los que llama se encapsule en excepción del tipo propio que define la capa con el mensaje de error adecuado y el identificador numérico de error asociado a ese error.

Este controlador implementa la interfaz *InterfazControladorModelo* y por ello debe como mínimo implementar los métodos declarados en esta. Además, dispone de métodos privados que hacen uso de *Java Generics* para mejorar la implementación y estructura de los métodos públicos declarados en la interfaz.

Para desempeñar su misión a veces lleva a cabo tareas tan simples como redireccionar una solicitud a la parte correcta de la capa y retornar, si lo hubiera, el resultado que se genere. En esta capa es lo que ocurre con el subsistema de lógica de calculadoras pues sólo debe hacer redirecciones a los métodos de la interfaz de este subsistema. Esta es la forma en la que el controlador proporciona al exterior todos los casos de uso y funcionalidades descritas en los subapartados anteriores que habla de la capa lógica de negocio.

Por motivos sobre todo de optimización, se ha declarado un atributo de clase para almacenar un recurso simple (clase *Recurso*) y un recurso compuesto (clase *RecursoCompuesto*)

como si de una memoria caché se tratara. De esta forma, varias solicitudes consecutivas que sean referidas al mismo recurso podrán ejecutarse con un mayor rendimiento. Junto a estos atributos de clase hay algunos métodos privados para gestionar esta funcionalidad que ofrecen como memoria caché.

El controlador cuenta con un método *obtenerListaUnificadaModulosSecciones()* que recibe como argumento una lista de identificadores de secciones y una lista de mapas. La clave de los mapas son los identificadores de las secciones y el valor asociado a la clave es una lista de módulos. Entonces se puede ver el segundo argumento como una lista de listas de módulos donde cada lista es la lista de módulos de una misma sección. El orden de la lista que se recibe en ambos argumentos se supone coherente. A partir de estos argumentos y realizando peticiones a otras clases de la capa pretende generar un mapa en el que las claves van a ser identificadores de módulos y el valor asociado a esa clave es una lista. El mapa tendrá una entrada por cada módulo distinto existente entre todas las secciones de los argumentos recibidos. La lista asociada a cada una de estas claves va a contener la siguiente información: Como primer elemento el título del módulo al que se refiere la clave del mapa (posición *0* de la lista). Después, aprovechando que las secciones comienzan a numerarse en *1*, guardará en esta lista en la posición indicada por el identificador de cada sección un *flag* (tipo cadena de texto) que indica si dicho módulo existe en esa sección. Se trata de un método genérico, pero que siempre se va a utilizar sólo para las secciones *Capítulos* y *Podcast* porque para cargar los elementos que deben aparecer en la barra lateral de navegación sólo interesa saber para cada módulo diferente existente en el sistema cuales tienen asociados recursos en cuales de esas dos secciones (si es sólo en una y cual o si es en ambas).

El controlador implementa otro método que recibiendo como argumentos lo que devuelve el anterior, una lista de nombres de las secciones cuyos identificadores se han utilizado en el método anterior (estando cada nombre en la posición del identificador asociado a la sección a la que se refiere) y una lista del nombre de las secciones cuyos identificadores no se han utilizado en el método anterior, genera y retorna una lista. La lista que retorna es la lista de cadenas de texto que la vista mostrará tal cual en la barra lateral de navegación.

Con estos dos métodos generales de la lógica de negocio, el controlador principal puede pedir a esta capa que procese de una forma muy concreta una información que le proporciona. Como resultado se obtiene una nueva información resultado del procesado y que el controlador principal retornará a la vista. Así pues, la vista le pide al controlador que este le proporcione el contenido que debe cargar en la barra lateral de navegación y el controlador principal obtiene información del modelo que luego pasa a la lógica de negocio para que la procese de la forma explicada y la devuelva al controlador principal. El controlador principal devuelve esta información a la vista para que la muestre. La vista entonces pide algo y el controlador principal se la proporciona sin necesitar conocer para nada cómo este a su vez la ha obtenido. Desde el punto de vista de la lógica de negocio, el controlador principal le pide procesar de una forma muy concreta una información que le proporciona y le devuelve el resultado. La lógica de negocio debe encargarse de este tipo de cosas, pero, por ello, en ningún momento llega a saber para qué es el procesado que realiza. De nuevo, el controlador principal debe mediar para adaptar la información que genera la lógica de negocio a la información que le ha pedido la vista. En este caso concreto, esta tarea de mediación consiste en no hacer nada porque la información que retorna la lógica de negocio, que no es consciente de la existencia de la capa vista, puede entenderla la capa vista, que tampoco es consciente de la existencia de la capa lógica de negocio, sin necesidad de ninguna adaptación.

8.5 Lógica de conexión

Es una de las capas que componen al sistema y su finalidad va a ser la de realizar tareas relacionadas con las comunicaciones por Internet para proporcionar al sistema funcionalidades como el acceso a otros sistemas completos independientes que van a actuar como fuentes de información externas o interactuar con el *Gestor de Descargas* de Android para la descarga de ficheros. Como ya se ha mencionado en la capa modelo, la lógica de conexión no proporciona estructura a la información que recibe de otros sistemas externos. Esta capa recibe del controlador principal y a través de su propio controlador las peticiones que debe llevar a cabo. La capa implementa su propio tipo de excepciones y un *Enum* asociado con todas las posibles causas que pueden generar que se lance este tipo de excepción. Todos los errores que se puedan dar en esta capa se encapsulan en su tipo de excepción propio antes de permitir que lleguen al controlador principal. Por otro lado, esta capa se divide en dos subsistemas de igual nivel: El subsistema de comunicación con la *API REST* de un servicio Web y el subsistema de comunicación con el *Gestor de Descargas* de Android.

8.5.1 Comunicación con el servicio Web

Es el subsistema que implementa la lógica necesaria para poder acceder a través de Internet con del protocolo *HTTP* a un servicio Web proporcionado por un sistema externo completo e independiente alojado en una máquina servidora remota que contiene una base de datos con todos los contenidos de HepApp. Así pues, este subsistema va a permitir obtener de la base de datos remota todos los datos que necesita la aplicación para funcionar. El subsistema sabrá qué información debe pedir y cómo debe pedirla, aunque no sea capaz de comprenderla. La información a la que va a acceder es la que se necesita para mantener sincronizada la base de datos local de la aplicación, así como obtener las versiones de los ficheros asociados a los recursos remotos. El subsistema se va a apoyar en la biblioteca *Volley* de *Google*.

8.5.1.1 La cola de peticiones

Se define la clase *ColaPeticionesServidor* que implementa la interfaz *VolleyCallback* y que es la que va a permitir hacer uso de la biblioteca *Volley*. La clase ha utilizando el patrón *Singleton* porque es lo correctamente recomendado por la documentación oficial de la biblioteca *Volley* ya que la aplicación a lo largo de su ejecución es bastante posible que solicite numerosas veces la versión de diferentes ficheros remotos y el uso de este patrón permite optimizar estas tareas (Android Developers, 2018).

La clase define unas constantes privadas para definir etiquetas que se van a vincular a distintos tipos de peticiones de forma que más adelante se pueda referir a partir de esta etiqueta al conjunto de todas las peticiones de un tipo concreto pendientes de resolver para, por ejemplo, solicitar su cancelación. También se define como una constante la dirección URL del servicio

Web al que se va a acceder a través de su *API REST* (por ejemplo, http://hepapp.es/api/read.php).

La clase cuenta con un método que usa *Java Generics* para añadir las peticiones ya generadas a la clase *RequestQueue* de *Volley* que es la que realmente actúa como cola de peticiones y se va a encargar de llevar a cabo la comunicación a través de Internet de forma transparente. También se implementan dos métodos que solicitan cancelar grupos completos de peticiones pendientes de un tipo a partir de las constantes declaradas en la clase que actúan como etiquetas de tipos de peticiones.

La clase implementa un método llamado *realizarPeticionCadena()* que permitirá generar la petición del tipo *StringRequest* de *Volley* para poder obtener información almacenada en un objeto tipo *String* desde una fuente externa. Además, implementa el método *realizarPeticionMapaObjetos()* que permitirá generar la petición del tipo *StringRequest* de *Volley* para poder obtener información almacenada en un objeto de tipo *String* desde una fuente externa. En ambas la respuesta es obtenida como un objeto *String* que contendrá información con una estructura conocida por la lógica de conexión. Esta estructura es *JSON* y es ampliamente utilizada para consumir un servicio Web que ofrece una *API REST*. Se ha dicho que la lógica de conexión no puede proporcionar estructura a la información, pero eso no implica que no pueda conocer estructuras de datos propias de las comunicaciones para abstraer al sistema del acceso a una fuente de información externa. La lógica de conexión conoce entonces la estructura *JSON* porque en esta se encuentra encapsulada la información que pretende conseguir para el controlador principal que es quien se la ha pedido. Debe conocer la estructura *JSON* para poder desencapsular la información que ha obtenido, pero esto no implica que deba comprender la información desencapsulada cuya estructura desconoce y proporcionará de vuelta a ciegas al controlador principal. Esta información será la que el controlador principal dará al modelo para que la proporcione estructura, pero todo esto la capa lógica de conexión no lo sabe.

Los dos métodos cuentan con dos argumentos con el modificador *final*, el primer argumento es una clase anónima de la que se habla más adelante y el segundo argumento es un mapa que contendrá parámetros extra que se añaden a otros ya existentes para generar la petición *HTTP* al servicio Web.

Ambos métodos generan la petición de tipo *StringRequest* indicando que se debe utilizar el método *HTTP POST* y proporcionando dos clases anónimas que implementarán respectivamente el método de retrollamada *onResponse()* y el método de retrollamada *onErrorResponse()*. La implementación de estos dos métodos de retrollamada (van a actuar como *listeners*) en cada uno de los métodos que genera la petición para la cola de peticiones tendrá una implementación propia y distinta. En cada caso, los métodos *onResponse()* y *onErrorResponse()* al ser métodos de retrollamada serán llamados automáticamente por *Volley* cuando se haya llevado a cabo la petición generada con éxito o error respectivamente.

Las clases anónimas son frecuentemente utilizadas para la implementación de métodos de retrollamada ya que permiten que los métodos que estas implementan puedan tener acceso

a las variables locales y atributos de clase que están a su alcance en la clase en la que se escribe su código a pesar de que sus métodos son lanzados desde otra clase que no tiene una instancia ni acceso a la clase con el código de la clase anónima.

Así pues, la razón de la implementación de dos métodos que generan peticiones del mismo tipo es que cada uno genera peticiones para solicitar distintos tipos de información. El método *onResponse()* de la clase anónima que se define en el método *realizarPeticionCadena()*, ya recibe como argumento un objeto de tipo *String* con la información recibida en caso de éxito de la fuente de información externa. En este caso el método comprueba que la información que recibe del argumento tiene estructura *JSON* que conoce y por ello procede a desencapsular la información. Como resultado de esta tarea obtiene de nuevo una cadena de texto con la información que le ha solicitado el controlador principal y que la lógica de conexión no comprende. Realmente, como es un subsistema, este sólo ve al controlador de la capa lógica de conexión que es quien devuelve la información al controlador principal. El método *onResponse()* de la clase anónima del método *realizarPeticionMapaObjetos()* realiza las operaciones análogas con las que genera un mapa de objetos de estructura desconocida. Los objetos los recibe como una lista de cadenas de texto cada uno.

El modificador *final* de los argumentos de estos métodos que generan peticiones permite que los argumentos puedan ser accesibles para las clases anónimas que se definen en dichos métodos. Respecto a la clase anónima que en cada método define el método *onErrorResponse()* estos llevan a cabo la misma acción en ambos métodos.

El primer argumento que reciben estos métodos se trata también de una clase anónima. En cada método la clase anónima es diferente porque dichas clases anónimas implementan distintos métodos según lo que declara para cada una la interfaz a la que están asociadas. Así pues, la interfaz *VolleyCallback* que define que la clase *ColaPeticionesServidor* debe implementar los métodos *realizarPeticionCadena()* y *realizarPeticionMapaObjetos()*, define otras dos interfaces de nombre *ServidorStringCallback* y *ServidorMapaObjetosCallback*. Estas son respectivamente las interfaces asociadas a las clases anónimas que los métodos de generación de peticiones reciben como primer argumento. Las clases anónimas que estos reciben como primer argumento son las que van a permitir establecer otro mecanismo de retrollamada hacia las clases en las que se definen dichas clases anónimas. La interfaz *ServidorStringCallback* declara un método con firma *public void onStringReceived(String respuesta)* y la interfaz *ServidorMapaObjetosCallback* declara un método con firma *public void onDataReceived(Map<String, List<String>> mapaElementos)*. Ambas interfaces declaran un método con firma *void onError()*.

Los métodos generadores de peticiones van a llamar en su método *onResponse()* al método *onStringReceived()* o *onDataReceived()* (pasándoles la información recibida desencapsulada) según la clase anónima que reciban como primer argumento. Cuando se produce un error en *onResponse()* al desencapsular la información o se acciona el método *onErrorResponse()* en lugar del método *onResponse()* se va a llamar al método *onError()* de la clase anónima que en cada caso se recibe como primer argumento.

Finalmente, el mapa que los métodos generadores de peticiones reciben como segundo argumento servirá para especificar algunos parámetros (otros se añaden en estos métodos o los añade posteriormente *Volley*) para la operación que se va a pedir al servicio Web y que viajarán en la cabecera *HTTP* dado que se usa el método *HTTP POST*.

8.5.1.2 Gestor de la comunicación que obtiene la información para sincronizar la base de datos

La clase *SyncDBhandler* se va a encargar de gestionar el uso que se hace de la cola de peticiones explicada en el subapartado anterior y por lo tanto de forma indirecta de la biblioteca *Volley* para el uso que requiere de esta. El objetivo de esta clase es entonces el de solicitar una tabla completa de la base de datos remota. Implementa el método *obtenerTabla()* que recibe como argumento el nombre de la operación que debe mandar como parámetro extra al servicio Web para obtener una de las tablas de la base de datos remota. Sabe entonces qué está pidiendo al servicio Web pero una vez que lo recibe de este sigue sin saber interpretar la información.

El método define una clase anónima que implementa la interfaz *ServidorMapaObjetosCallback* y por ello implementa el método *onDataReceived()* y el método *onError()*. En esta clase anónima el primer método llama al método *establecerTablaRecibida()* pasándole el argumento que recibe y el segundo llama al método *establecerFlagCancelacion()* con el valor booleano *true*.

Después de definir la clase anónima llama al método *realizarPeticionMapaObjetos()* pasando la clase anónima que acaba de definir y el mapa de parámetros al que ha añadido el parámetro que va a indicar al servicio Web el nombre de la operación a realizar, a continuación establece un temporizador de hasta un máximo de *6,5* segundos y finalmente retorna el valor que obtiene llamando al método *obtenerTablaRecibida()*.

Los métodos de visibilidad privada *establecerTablaRecibida()*, *obtenerTablaRecibida()*, *establecerFlagCancelacion()* y *obtenerFlagCancelacion()* son métodos *getter* y *setter* de atributos de clase que tienen en su firma el modificador *synchronized* para indicar que la tarea que realiza ese método debe ejecutarse como si se encontrara en un bloque *synchronized*. Sin entrar demasiado en detalle, este modificador a efectos prácticos permite que el acceso a una misma variable o atributo se realice de forma concurrente, es decir, permite el sincronismo entre varios *thread* al acceder a la variable o atributo de forma que los accesos se realizan de forma atómica como ocurría con las transacciones en las bases de datos. Esto previene, por ejemplo, que, si un *thread* está escribiendo un valor, otro *thread* pueda realizar un acceso de lectura a la mitad de la tarea de escritura que resultaría en una lectura errónea. ¿Y por qué es necesario este modificador? Porque el uso de la clase *SyncDBhandler* se hace desde un *thread* y la ejecución de la petición que se genera se realiza en otro. Esto justifica también la necesidad de los mecanismos de retrollamada pues *Volley* realiza sus operaciones en un *thread* propio que activará el método *onResponse()* (o el método *onErrorResponse()*) cuando tenga una respuesta a la petición, que a su vez ejecutará el método *onDataReceived()* (o el método *onError()*), que a su vez ejecutará métodos de la clase *SyncDBhandler* desde un *thread* distinto al *thread* que

también utilizó la clase para generar la petición y que se ha quedado esperando a la respuesta del servicio Web. Si el *thread* que activa el mecanismo de retrollamada ejecuta métodos de la clase *SyncDBhandler* al mismo tiempo que el *thread* que primeramente usa la misma clase para generar la petición al servicio Web también lo hace (porque está esperando) podría darse una situación de acceso simultáneo no deseado como el descrito y que se soluciona con el modificador *synchronized*.

El temporizador de espera se ha implementado con un bucle que realiza en cada iteración una espera de cien milisegundos hasta un máximo de *65* iteraciones. En cada iteración, antes de realizar la espera se llama a los métodos *obtenerTablaRecibida()* y *obtenerFlagCancelacion()* para comprobar si el atributo en el que el otro *thread* debe cargar la respuesta recibida del servicio Web cuando la haya vale distinto de *null* o el *flag* de cancelación del temporizador (que es también un atributo de clase) vale *true* para cancelar consecuentemente el temporizador. El temporizador entonces se puede cancelar porque se ha producido un problema con la petición realizada (o esta no se ha podido llevar a cabo o la respuesta recibida no es la esperada), porque ya se ha recibido exitosamente la respuesta a la petición y carece de sentido seguir esperando o porque el usuario ha solicitado explícitamente que se debe abortar (al cancelar el intento de sincronización del contenido de la aplicación que se produce automáticamente al iniciar la aplicación). Hubiera sido más conveniente realizar la implementación del temporizador utilizando las clases del paquete *java.util.concurrent*. Sin embargo, se ha implementado de esta forma por simplicidad y por la falta de tiempo para aprender y experimentar todo lo necesario sobre las clases de dicho paquete.

El tiempo de espera que se establece para el temporizador es clave para evitar errores constantes forzados porque siempre expire el temporizador por hacer una espera demasiada pequeña pero también para no permitir una espera ilimitada, o mejor aún, no desperdiciar a penas el recurso tiempo de CPU y optimizar el rendimiento si el tiempo de espera se encuentra bien ajustado. Elegir el valor de espera del temporizador es un compromiso entre el fuerce de errores innecesarios y rendimiento. Si el error de la tarea provoca un nuevo reintento entonces sólo se produce una pérdida de rendimiento si finalmente llega a realizarse la operación que siempre es más grave que el que nunca se realice la operación porque no se produzcan reintentos de la operación o ya se hayan realizado demasiados.

Tanto cuando el temporizador expira como cuando se cancela se llama al método *obtenerTablaRecibida()* y el valor que este retorne será el que retorne el método *obtenerTabla()*. Este valor podrá ser *null* si se ha producido un error o si ha expirado el temporizador antes de haberse obtenido una respuesta correcta del servicio Web, o podrá ser la respuesta correctamente recibida del servicio Web.

Finalmente, la clase implementa un método *cancelarPeticion()* que solicitará primero a la cola de peticiones que aborte todas las peticiones de un tipo y después llamará al método *establecerFlagCancelacion()* pasando el valor booleano *true*. Este método, por ejemplo, es el que se llamará cuando el usuario solicite abortar la petición al servicio Web explícitamente.

8.5.1.3 Gestor de la comunicación que obtiene la versión de un recurso remoto

La clase *GetVersionHandler* es la clase de implementación análoga a la clase *SyncDBhandler* y que tiene por finalidad solicitar la versión del fichero de un recurso remoto.

Para ello, implementa un método *obtenerVersionRecurso()* que recibe como argumento el identificador del recurso remoto asociado al fichero del que se quiere obtener la versión actual. Este método, aparte de establecer el parámetro del nombre de la operación que debe realizar el servicio Web también establece el identificador del recurso remoto como parámetro con el que generar la petición. La clase anónima que define implementa la interfaz *ServidorStringCallback* y por ello los métodos que esta declara. El método *onError()* presenta el mismo comportamiento que en la clase *SyncDBhandler*. Para generar la petición, en este caso, llama al método *realizarPeticionCadena()*. El valor del temporizador de espera en este caso es de siete segundos también en iteraciones de cien milisegundos.

El motivo de haber establecido este valor de espera para el temporizador, que supone un aumento significativo dado el tamaño y la simplicidad de la respuesta que se espera recibir frente a la de la clase *SyncDBhandler*, es que se considera bastante más importante o crítica la obtención de esta respuesta para el funcionamiento correcto de la aplicación entre otros porque en el uso de la aplicación puede que se necesite hacer con frecuencia peticiones de este tipo dado que se intenta, y en caso de fallo hasta varias veces, la petición de la versión de un recurso cada vez que se carga un recurso en un visualizador. Por ello, se pretende dar un valor al temporizador que priorice que la solicitud de obtención de la versión de un recurso no falle por no haber esperado lo suficiente en cada intento ante el tiempo de espera de más en el que la CPU permanece en estado ocioso, pues un valor inapropiado podría forzar varios reintentos que acabaran lastrando aún más el rendimiento del sistema completo y proporcionara al usuario una sensación de pesadez y malfuncionamiento que acabara afectando a la satisfacción y la percepción que este tiene de la aplicación.

8.5.2 Comunicación con el gestor de descargas de Android

Es el subsistema que implementa la lógica necesaria para poder interactuar con el *Gestor de Descargas* de Android para solicitarle la descarga de ficheros. Dado que el *Gestor de Descargas* de Android está implementado en el dispositivo móvil como una aplicación del sistema, y que como tal se ejecuta en su propio subproceso principal, se necesita mecanismo que permita conocer a HepApp que la descarga ha finalizado para continuar realizando las acciones correspondientes. En esta situación, por tratarse de una aplicación independiente no se puede implementar un mecanismo de retrollamada similar a los ya vistos. Se utilizará entonces un *BroadcastReceiver* junto a un *IntentFilter* pues el *Gestor de Descargas* notificará que la descarga se ha completado lanzando un *Intent* de características concretas que HepApp tomará y reconocerá.

8.5.2.1 Solicitud de descarga a partir de una dirección URL

La clase *DescargarFicheroDesdeURL* es la que implementa toda la lógica necesaria para realizar peticiones de descarga al *Gestor de Descargas* de Android.

Esta define varias constantes, por un lado, unas constantes para el protocolo y directorio que se usarán para generar la dirección URL y la ruta de destino que se va a proporcionar al *Gestor de Descargas* para que sepa en qué directorio del sistema de ficheros debe guardar al fichero que se descargue. Por otro lado, las constantes que van a definir el identificador que se va a utilizar para la *API* de preferencias junto a las claves de los valores que se van a almacenar.

Sólo la constante con el identificador usado en la API de preferencias deberá ser visible desde fuera del subsistema. Se ve más adelante, pero se utilizará un mapa con clave y valor de tipo *String* que contendrá los valores que deben guardarse sobre la descarga.

Cuenta también con un método de acceso estático que permite comprobar (retornando un valor booleano) si el protocolo de una dirección URL que recibe como argumento (como objeto de tipo *URL*) es el protocolo que tiene definido en la constante y que se refiere a un fichero local.

El constructor recibe cinco argumentos, un objeto de tipo *Context* con el contexto de la aplicación que necesitará para obtener una referencia al *Gestor de Descargas* de Android, el identificador del recurso a descargar, un objeto de tipo *URL* con la dirección de la ubicación actual del fichero a descargar que se encontrará en la máquina servidora remota, una cadena de texto con el nombre de la actividad visualizador que ha originado la acción de descargar un recurso y una cadena de texto con el texto que se indicará al *Gestor de Descargas* que deberá mostrar en la notificación de descarga que él muestre.

El constructor, con ayuda del método *getExternalStorageDirectory()* de la clase *Environment* genera a partir de los argumentos que recibe y almacena como atributos de clase la ruta en la que se va a guardar el fichero que se descargue y el nombre de dicho fichero.

Se implementan tres métodos *getter* para obtener el valor de estos dos atributos de clase y para obtener la ruta de destino completa formada a partir de los dos atributos de clase.

La clase se ha implementado de esta forma para que tras una petición del controlador principal para hacer los preparativos de una solicitud de descarga al *Gestor de Descargas* de Android (constructor de la clase), el controlador principal pueda obtener tanto la ruta del directorio de descarga como la ruta de destino completa (incluye el nombre del fichero a descargar) para solicitar al modelo que compruebe la existencia del directorio (y que lo cree si no existe), así como para comprobar si ya existe en ese directorio un fichero con el mismo nombre para que lo elimine en caso de existir, antes de ordenar que se mande al *Gestor de Descargas* la petición de descarga del fichero deseado.

El controlador principal, sólo si las comprobaciones del modelo son exitosas, solicitará la descarga del fichero. Para ello, la clase implementa el método *descargarRecurso()*. Este método obtiene una referencia al *Gestor de Descargas* con el método *getSystemService()* y

después instancia un objeto del tipo *DownloadManager.Request()* para generar la solicitud de descarga. Posteriormente configura la petición con la dirección URI de la ubicación actual del fichero a descargar, el título de la notificación que se recibió en el constructor, la restricción de no realizar la descarga en modo *roaming* y la especificación de la ruta y el nombre con que se almacenará el fichero descargado.

Después utiliza el método *enqueue()* al que pasa la petición que se acaba de generar y configurar para solicitar al *Gestor de Descargas* que encole la petición de descarga.

Finalmente, con el identificador de descarga que obtiene al llamar al método *enqueue()*, genera el mapa con los metadatos de la descarga utilizando por claves las constantes definidas como constantes en la clase. Estos metadatos de la descarga son el identificador de la descarga, el identificador del recurso, el nombre de la clase de la actividad del visualizador desde el que el usuario ha solicitado la descarga y la dirección URL local de destino en la que se va a guardar el fichero. El método *descargarRecurso()* retorna este mapa para que el controlador principal se lo pase a la capa modelo quién lo almacenará de forma persistente a través de la API preferencias utilizando el identificador definido en una constante que antes obtuvo a partir del método *getter* correspondiente.

La clase cuenta con un método *cancelarDescargaRecursoEnCurso()* para solicitar al *Gestor de Descargas* de Android que cancele la petición de descarga con identificador el que recibe como argumento.

8.5.2.2 Gestión del evento descarga completada

Para la gestión del evento de descarga completa o errónea que notifica el *Gestor de Descargas* de Android, primero se debe detectar este. Como ya se ha dicho, este evento lo recibe la aplicación con un *BroadcastReceiver* pues el *Gestor de Descargas* de Android manda un *Intent* al procesar la petición de descarga que previamente se le ha hecho.

El *BroadcastReceiver* se define en el controlador principal mediante una clase anónima que implementará el método *onReceive()* que tiene como argumentos un objeto del tipo *Context* y un objeto de tipo *Intent*. La clase anónima que define el *BroadcastReceiver* se ha implementado en el controlador principal porque al recibirse el evento de descarga completada por parte del *Gestor de Descargas* se necesita pedir a la capa modelo que recupere el mapa de metadatos de la descarga antes de pedir a la capa lógica de conexión que gestione el evento recibido de descarga completada. Así pues, el método *onReceive()* se ejecuta a nivel de la capa controlador (controlador principal) y tras solicitar a la capa modelo el mapa de metadatos de la descarga llama (a través del controlador de la capa lógica de conexión) al método *comprobarEventoDescargaCompelta()* de la clase *GestorEventoDescargaRecurso* del subsistema en cuestión. Finalmente, según el valor booleano que retorna el método anterior, solicita al modelo el borrado de los metadatos de la descarga y después llama al método *notificarVisualizadorDescargaCompelta()* de la clase *GestorEventoDescargaRecurso* o da por finalizada su ejecución. El identificador utilizado para la API preferencias se la vuelve a pedir (las veces que sean necesarias) al controlador de la lógica de conexión quien la toma de la constante

pública que define la clase *DescargarFicheroDesdeURL* como ya se ha dicho (como se comentó hablando de otra capa, aunque el controlador principal podría acceder directamente a la constante lo suyo es que llame a un método del controlador de la capa lógica de conexión el cual se la obtenga).

El método *comprobarEventoDescargaCompelta()* recibe como argumentos el *Intent* que se recibe del método *onReceive()* y el mapa de los metadatos de la descarga. El método retorna un valor booleano no para indicar si el *Gestor de Descargas* de Android ha notificado que la petición de descarga se ha llevada a cabo con éxito o se ha producido un error, sino para indicar si el evento de descarga completada detectado se refiere a la descarga que se pidió o no. La detección del evento se realiza mediante un *BroadcastReceiver* que dejará pasar todos los *Intent* que lance el *Gestor de Descargas*, incluso aunque se refieran a peticiones de descargas que otras aplicaciones han hecho al *Gestor de Descargas* de Android. Por ello la importancia del identificador de descarga que se obtuvo al llamar al método *enqueue()* del *Gestor de Descargas*, porque comparando el identificador de descarga que se obtiene en el *Intent* y el que se tiene guardado en el mapa de metadatos de la descarga se puede identificar si el evento de descarga completada detectado se refiere o no a la petición de descarga que se hizo y cuya respuesta se estaba esperando. Es importante destacar que si el evento de descarga completa recibido no se refiere a la descarga que se solicitó este método debe retornar el valor booleano *false* para que el controlador principal no solicite al modelo que borre los metadatos de la descarga ni llame al visualizador que originó la petición de descarga porque después de este evento de descarga completa podría producirse otro que esta vez sí se refiera a la descarga que se solicitó.

El método *notificarVisualizadorDescargaCompelta()* sólo será llamado tras la solicitud del controlador principal a la capa modelo para que elimine los metadatos de la descarga, porque al igual que esta solicitud este método sólo puede ejecutarse cuando se comprueba que el evento detectado sí se refiere a la descarga que se solicitó. Recibe como argumentos el objeto de tipo *Context* que se recibe del método *onReceive()* y el mapa de metadatos de la descarga. No se debe confundir que se solicite a la capa modelo el borrado de los metadatos con que estos se encuentren de antes cargados en una variable de mapa local del método *onReceive()*. Con estos argumentos, el método podrá generar y lanzar un *Intent* (con los *flag* adecuados definidos como constantes en la clase *Intent*) destinado a la actividad del visualizador que originó la descarga del recurso. Este método sí comprueba si el evento de descarga completa detectado indica si la descarga se ha completado correctamente o se ha producido un error porque en función de esto será distinto el *Intent* que genera y lanza para el visualizador (cuya clase obtiene mediante *Java Reflection* y *Java Generics* a partir del nombre de la clase del visualizador que recibe en el mapa de metadatos). Si la descarga se ha completado correctamente, se adjuntará en el *Intent* el identificador del recurso que se ha descargado y la dirección URL local con la que se va a registrar el recurso como local en el sistema (ambos datos se toman del mapa de metadatos). Si se ha producido un error con la descarga, el *Intent* sólo llevará adjunto un *flag* booleano que indicará al visualizador que se ha producido un problema con la descarga de forma que este notifique la situación al usuario. De nuevo, como ya se ha comentado varias veces, si fuera necesario, el controlador principal será el encargado de mediar entre la capa vista y la capa lógica de conexión para adaptar y poner en común las constantes que se usan como claves de los datos adjuntos en el *Intent*. Obviamente, el valor de las constantes en ambas capas debe ser el mismo, pero con que el controlador principal media se refiere a que, si fuera necesario, indica a cada capa qué constante de las que tiene debe utilizar en cada caso como clave para obtener

un dato. También podría ser que el visualizador a la hora de obtener cada valor del *Intent* tendría que pedir al controlador la clave que debe utilizar para obtenerlo.

El visualizador obtendrá este *Intent* a través de su método *onNewIntent()* para proceder a la segunda parte de la descarga (el registro) como ya se explicó en el subapartado *Visualizador genérico*. El visualizador sólo recibe este *Intent* con el método indicado si ya está creada su actividad porque, en caso de no estar creada su actividad, debido a los *flag* establecidos en el *Intent* no se creará una nueva instancia de la actividad que implementa el visualizador y, por consecuencia de esto, no se realizará la segunda parte de la descarga que se dará por fallida por omisión del *Intent*.

La clase *GestorEventoDescargaRecurso* toma las constantes que definen las claves del mapa de metadatos de la clase *DescargarFicheroDesdeURL* pues perteneciendo ambas clases al mismo subsistema se tiene acceso a las constantes.

La clase *GestorEventoDescargaRecurso* carece de estado y por ello sus métodos son de acceso estático. Estos se podrían haber definido directamente en la clase *DescargarFicheroDesdeURL*, pero por una cuestión de organización y separación conceptual se han implementado en otra clase. Por ello, la clase no implementa un constructor porque no está pensada para que pueda ser instanciada.

Como se ha comentado, el *BroadcastReceiver* hace uso de un *IntentFilter* para que de todos los *Intent* que va a recibir el *BroadcastReceiver* (todos los que manden el conjunto de todas las aplicaciones instaladas en el dispositivo) se haga un filtrado y sólo se dejen pasar aquellos que cuentan con unas características concretas. El filtrado se especifica con la clase *IntentFilter* a la que se especifica la constante *ACTION_DOWNLOAD_COMPLETE* de la clase *DownloadManager* para indicar que sólo deje pasar los *Intent* lanzados por el *Gestor de Descargas* de Android que se refieren a la notificación de descarga completa. La asociación del *IntentFilter* con el *BroadcastReceiver* se realiza también en el controlador principal en el momento del registro del *BroadcastReceiver* en la aplicación, porque es el controlador principal el que tiene al *BroadcastReceiver* por haber implementado la clase anónima que lo define.

El controlador de la lógica de conexión implementa el mecanismo para rechazar nuevas solicitudes de descarga del fichero asociado a un mismo recurso, al menos mientras haya una descarga en curso del mismo recurso. En teoría este mecanismo no debería tener que intervenir nunca pues la vista ya implementa un mecanismo análogo (este se desbloquea al pasar a cargar en el visualizador otro recurso diferente), pero esto no quita que la capa lógica de conexión tenga que implementar otro mecanismo análogo porque no se debe fiar a ciegas de las solicitudes que recibe y no es consciente de la existencia de la capa vista (y por ello de su mecanismo). Se sigue permitiendo la gestión del evento de descarga completada del fichero asociado a dos recursos distintos porque los metadatos de cada descarga se pueden almacenar usando la API de preferencias con identificadores distintos. Para ello, la clase

DescargarFicheroDesdeURL define una segunda constante de visibilidad pública con otro identificador para utilizar con la API de preferencias sólo cuando el identificador definido en la otra constante ya está en uso. El controlador de la lógica de conexión cuenta con la lógica para gestionar esta tarea y almacena la relación temporal (hasta que se resuelva su descarga) entre estas constantes y el recurso que se está descargando. Se ha limitado de forma intencionada a que sólo se puedan tener pendientes de resolución dos descargas de diferentes recursos de forma simultánea. Todas las peticiones de nuevas descargas serán rechazadas si se ha alcanzado esta limitación.

8.5.3 Controlador de la capa lógica de conexión

Es el controlador (de fachada) específico de la capa lógica de conexión. Recordando algunas de las cosas ya dichas acerca de los controladores en otras partes de este documento, su finalidad es la de recibir todas las peticiones que llegan desde fuera de la capa que tienen como fin utilizar uno de los casos de uso que esta ofrece al resto del sistema, coordinar las clases y los subsistemas que forman la capa llamando en el orden apropiado a varios de sus métodos con los argumentos correctos para llevar a cabo la petición recibida y, finalmente, si se genera un resultado para la petición retornarlo a quién ha hecho la solicitud.

También se encarga de que cualquier error que se origine en los métodos a los que llama se encapsule en excepción del tipo propio que define la capa con el mensaje de error adecuado y el identificador numérico de error asociado a ese error.

Este controlador implementa la interfaz *InterfazControladorLogicaConexion* y por ello debe como mínimo implementar los métodos declarados en esta. Además, dispone de métodos privados que hacen uso de *Java Generics* para mejorar la implementación y estructura de los métodos públicos declarados en la interfaz.

Para desempeñar su misión a veces lleva a cabo tareas tan simples como redireccionar una solicitud a la parte correcta de la capa y retornar, si lo hubiera, el resultado que se genere. En esta capa es lo que ocurre con los dos subsistemas de los que se compone. El controlador recibe en algunos de sus métodos un objeto de tipo *Context* con el contexto de la aplicación que no es susceptible de quedar invalidado si una actividad se destruye. Cuenta con métodos para solicitar y cancelar la descarga de un fichero, la obtención de la versión de un recurso remoto y la obtención de una tabla de la base de datos remota para permitir la sincronización con la base de datos local.

La lógica que pone este controlador ya se ha explicado en varios de los subapartados anteriores. Un detalle a mayores que se puede aclarar, es que la cancelación de la descarga de un fichero la realiza el controlador principal primero solicitando a través de la lógica de conexión la cancelación al *Gestor de Descargas* y después solicitando al modelo el borrado de los metadatos de la descarga guardados. De esta forma, si es demasiado tarde para el *Gestor de Descargas* de Android, aunque la aplicación detecte el evento de descarga completada no dispondrá de los metadatos de la descarga necesarios para su gestión y este se omitirá.

8.6 Otros

Las imágenes no configurables desde el panel Web como las imágenes de las capas de más información y los diferentes esquemas de *Alberta* de la sección calculadoras que se incluyen junto a la aplicación como ficheros de recursos (directorio *res* del proyecto de *Android Studio*) se han optimizado logrando una reducción entre el 75% y el 92% del tamaño de cada imagen sin afectar a las dimensiones ni la calidad de las mismas. Es decir, esta optimización, se puede considerar como algo transparente al usuario dado que este no percibiría que se ha aplicado. Esto ha permitido reducir considerablemente el tamaño final del fichero *APK* con la aplicación construida y lista para instalar en un dispositivo Android compatible.

Capítulo 9: Conclusiones y líneas futuras

En este capítulo primeramente se presenta un estudio económico en el que se valoran los costes implicados en caso de haber encargado la realización de este proyecto a una empresa. A continuación, se mencionan aspectos y posibles mejoras que podrían hacerse sobre el trabajo efectuado en caso de continuarse el proyecto. Finalmente se comentan las conclusiones que se extraen de haber realizado el trabajo, así como la revisión de todos los objetivos planteados al principio del documento.

9.1 Estudio económico

Realizar un presupuesto económico que pretenda recoger todos los aspectos variables de los que puede depender el coste del desarrollo de un sistema es algo que no se puede realizar de forma sencilla. Esto se debe a que aplicaciones aparentemente sencillas pueden suponer un gran tiempo de desarrollo y lo mismo sucede al contrario. En definitiva, es algo que depende mucho del tipo de aplicación que se necesite desarrollar y también de cuál es el estado del proyecto en el momento de solicitar su desarrollo. No es lo mismo encargar el desarrollo de una aplicación contando con un diseño completo de la interfaz gráfica y el flujo de funcionamiento del sistema, proporcionar una aplicación Web de referencia para la que se quiere crear una aplicación o tener sólo una vaga idea de qué se quiere que se desarrolle. Por otro lado, en el mundo de la tecnología todo es lo suficientemente cambiante como para afectar de forma relevante a la estimación del coste.

A continuación, se plantea en la tabla Tabla 3 la estimación del coste económico para la solución desarrollada en este documento.

Estimación del coste del desarrollo			
Concepto	**Cantidad**	**Precio**	**Total**
Sueldo programador junior	430h	11€/h	4730€
Licencias software	0	0€	0€
Servicio hosting	1	5€/mes	5€
Equipamiento	Equipo escritorio 650€ Dispositivo móvil 170€	Coste de 3 meses	102,5€
Cuota de autónomos	3	278,87€/mes	836,61€
Suministros	Internet y corriente	78€/mes	234€
Alquiler espacio de trabajo	-	-	-
		Total estimación	5908,11€

Tabla 3. Estimación del coste del desarrollo del proyecto HepApp.

Entrando algo más en detalle con los datos de la tabla, se cree que el tiempo estimado del desarrollo de la solución planteada es entorno a las 430 horas, lo cual se corresponde con casi tres meses contando cuarenta horas semanales y cuatro semanas por mes. Para el subtotal se ha considerado el sueldo promedio de un desarrollador junior (primer trabajo o menos de un año de experiencia) en España. En cuanto a las licencias del uso de software para el desarrollo no hay coste ya que todas las tecnologías utilizadas para el desarrollo son gratuitas. Esto podría

haber encarecido bastante más el coste final de haberse utilizado otras tecnologías. En esta ocasión se requiere de un servicio hosting para desplegar la aplicación Web una vez desarrollada. El resto son los gastos típicos asociados al equipamiento necesario, los suministros del equipamiento y la cuota de autónomos que corresponde. Respecto al espacio de trabajo, no se ha considerado ningún coste adicional pues podría llevarse a cabo el trabajo desde casa.

Por otro lado, se deben tener en cuenta varias consideraciones sobre el desarrollo de la solución no plasmadas en la tabla anterior. El coste de desarrollo de una aplicación se incrementa notablemente si esta debe contar con algunas variables consideradas ciertamente complejas o muy técnicas como son el acceso a datos a través de Internet para actualizar la información de la aplicación, la complejidad de que los contenidos de la aplicación sean dinámicos en vez de fijos o la integración con otros sistemas. El desarrollo de la aplicación móvil es un desarrollo nativo porque cuenta con quizás la suficiente complejidad como para que no pudiera ser desarrollada de forma híbrida. Esto también implica que en caso de quererse contar con la aplicación móvil para otra plataforma se tendría que desarrollar de nuevo.

Además, ya se partía de tener todo el diseño gráfico de la aplicación móvil hecho y, lo que es más, tener perfectamente pensado todo su flujo de funcionamiento. De no contar con ello, el tener que recurrir a un nuevo perfil profesional (diseñador gráfico) podría encarecer el coste. Asimismo, no es lo mismo encargar el desarrollo de un proyecto cuando sólo es una vaga idea que cuando se trata de algo totalmente planificado. De hecho, se ha partido de contar con implementaciones anteriores para la aplicación móvil. Por otro lado, también se debe tener en cuenta que la solución planteada ha consistido en el desarrollo de dos sistemas que deben estar integrados para funcionar en conjunto como se espera. A parte de que esto se puede considerar como el desarrollo de dos proyectos por separado (dos presupuestos) y la posterior integración, se debe tener en cuenta que implica que el proyecto sea atendido con alta dedicación por un perfil profesional más cualificado y con experiencia más específica dado que se trata de un proyecto más avanzado, complejo y técnico (al menos así es visto por las empresas de cara al coste). Se podría decir que todo esto está incluido de alguna manera en las horas de trabajo estimadas, pero no es un reflejo fiel de la realidad ya que todos estos aspectos en el mundo laboral se consideran como plusvalías que suben cada una el coste de forma apreciable.

9.2 Conclusiones

Para lograr cumplir con éxito los objetivos del capítulo *Introducción y contexto* se han seguido los siguientes pasos.

Primeramente, se han estudiado y analizado las diferentes tecnologías disponibles para el desarrollo del sistema solicitado prestando atención a las características y posibilidades ofrecidas por cada una. Tras ello, se opta por utilizar las tecnologías Web para una parte del sistema completo y la plataforma Android para el desarrollo nativo de una aplicación móvil que completa al sistema planteado. Esta decisión se apoya además en otras motivaciones distintas a las estrictamente técnicas como la cuota de mercado o de uso y la popularidad de las tecnologías utilizadas, así como las preferencias y gusto propio.

Tras ello, se propone y valida un diseño lógico que recoge todas las funcionalidades que se espera que estén presentes en el sistema de forma correctamente estructurada y flexible, ya que debe soportar el dinamismo implícito en el requerimiento de que los contenidos sean configurables (y por ello variables) de forma externa.

Una vez que se cuenta con un diseño adecuado este se ha llevado a la práctica mediante iteraciones que consisten en etapas de implementación y pruebas, originando al sistema completo resultante.

Como último paso se crea un manual de uso de todo el sistema tanto para los usuarios de la aplicación como para el usuario del panel de control Web que se va a encargar de configurar de forma remota desde este el contenido de la aplicación móvil. Es aquí cuando se valida el sistema desarrollado al comprobar desde una perspectiva más amplia y general el funcionamiento de todos los casos de uso y funcionalidades del sistema cuyo uso se documenta. La sencillez y simplicidad del manual da una idea de la usabilidad y del fácil uso del producto final desarrollado.

El desarrollo de este trabajo me ha permitido por un lado culminar algunos aspectos de toda la formación adquirida durante la carrera a un siguiente paso, al mismo tiempo que ha permitido aplicar de forma conjunta conocimientos que se habían visto de forma separada para poderlos llevar a la práctica desarrollando un único sistema compuesto e integrado.

Con este proyecto he podido extender, profundizar y utilizar conjuntamente los conocimientos adquiridos en asignaturas como Programación, Ingeniería de Sistemas Software, Arquitecturas de aplicaciones distribuidas, Tecnologías para aplicaciones Web, Desarrollo de Aplicaciones para Dispositivos Móviles y Sistemas de tiempo real. Así pues, debido a la completitud del trabajo, se ha logrado adquirir cierta destreza en el uso avanzado del lenguaje de programación *Java* y el manejo de bases de datos estructuradas, el diseño y la arquitectura de sistemas, el paradigma de la orientación a objetos y la orientación a eventos, las consideraciones sobre la ejecución en paralelo de tareas y su posterior sincronización, el uso de redes para la comunicación a través de Internet y la integración de sistemas de diferente naturaleza técnica, entre otros.

Respecto a las metodologías y destrezas de aprendizaje autónomo, estas han progresado por la necesidad de la constante consulta de información en documentación y manuales oficiales de las tecnologías utilizadas, así como en la observación de problemas planteados con sus soluciones por otros desarrolladores más experimentados y con más recorrido. Relacionado con esto, tanto para el desarrollo de los sistemas como de la redacción de esta memoria se ha puesto en práctica la capacidad de distinguir entre la información válida y fiable, y la que no lo es.

Además, se pueden valorar algunas competencias como la independencia, la capacidad de organización, o la planificación y gestión del tiempo implicadas al enfrentar la realización de este trabajo respetando los plazos establecidos. Relacionado con esto, se ha aprendido a utilizar herramientas para la mejora de productividad como es el caso de la toma de anotaciones y esquematización de las ideas para uno mismo, o el uso de un repositorio privado (en *GitHub*) para el control de versiones con su registro de cambios asociado.

Se debe destacar de este trabajo la dificultad de tener en cuenta todos los aspectos de un sistema destinado a un uso real en lugar de quedarse dentro del ámbito académico. Esto exige un funcionamiento mucho más estable y óptimo, y características adicionales que se pueden englobar dentro del concepto *calidad de software*. Por ello, el sistema desarrollado se ha probado en un entorno de pruebas utilizando varios dispositivos emulados y posteriormente se ha desplegado sobre un entorno de producción haciendo uso de un servicio profesional de hosting y el uso de la aplicación desde dispositivos físicos.

En cierto modo, se debe decir que se ha aprendido ligeramente sobre las temáticas involucradas en este proyecto como son la medicina (hepatología) y la educación (formación online y metodología educativa *Flipped Classroom*) a pesar de ser ajenas a los estudios realizados necesarios para la construcción de la solución tecnológica de este trabajo.

Personalmente también se aprecian otras competencias implicadas en este trabajo como la capacidad para redactar esta memoria, la capacidad de comunicar, describir y comprender conocimientos, procedimientos, resultados e ideas expresadas, valorar el impacto de las soluciones desarrolladas, las habilidades para la solución de problemas lógicos, o la capacidad de adaptación a los cambios (flexibilidad), entre otras.

Algunos retos que se han enfrentado durante el desarrollo de la solución han sido el uso de los mecanismos de concurrencia, la creación de servicios Web con una *API REST*, el consumo de la API desde la aplicación móvil, la integración entre los dos sistemas, la implementación estructurada de los algoritmos de las calculadoras y el uso de algunas bibliotecas o módulos software externos como *Volley* o el *Gestor de descargas* de Android. Como aspecto positivo, decir que no se ha encontrado con ningún problema de implementación relevante que no se haya podido solucionar o haya impedido el correcto funcionamiento del sistema resultante.

Por último, se agradece el apoyo de los especialistas y la información rigurosa aportada sobre el tema del que trata el trabajo que han hecho posible la realización del mismo.

También debo destacar la satisfacción y realización personal que hay detrás de este trabajo por las intenciones altruistas del proyecto y el saber que podría resultar de verdadera ayuda a doctores, alumnos universitarios en formación y otros profesionales de la salud que, con el uso de las herramientas y funcionalidades que HepApp ofrece se pueda, en definitiva, simplificar y mejorar la calidad de vida de los pacientes que padecen enfermedades relacionadas con la hepatología.

9.3 Líneas futuras

Durante el desarrollo del proyecto se han detectado posibles mejoras o nuevas funcionalidades que se podrían añadir. Las siguientes mejoras, se han separado según si son temas más técnicos o no.

- En el visualizador de las imágenes de la sección Figuras, en el caso de la interacción que permite dibujar sobre las imágenes se podría añadir la

funcionalidad de permitir también añadir cuadros de texto para realizar anotaciones textuales.

- A modo de una nueva sección también gestionada desde un panel de control Web o integrando otra aplicación Web, se podría mostrar desde un sólo lugar un nuevo tipo de contenidos del tipo noticias y artículos especializados y seleccionados que permitan a uno mantenerse actualizado en el panorama médico relacionado con la hepatología y sus avances. Además, dar a conocer el trabajo de otros investigadores, mostrar datos estadísticos resultantes de estudios y aspectos divulgativos sobre el tema con intención de concienciación. También puede servir para dar a conocer eventos como conferencias, charlas, reuniones, seminarios, jornadas, etc.
- Se puede integrar una aplicación Web que pretende actuar como una comunidad y especie de foro de dudas que sea como lo que *stackoverflow* es al desarrollo y programación, es decir, una comunidad de especialistas en la que otros usuarios pueden encontrar soluciones a problemas y dudas médicas consultadas. Esto debería realizarse de forma controlada de forma que se trate de una comunidad de calidad fundamentada en la reputación basada en un sistema de puntuación y catalogación de las respuestas como válidas y relevantes. De esta forma es posible que los usuarios adquieran incluso hasta cierta capacidad de moderación debido a su reputación adquirida en sus respuestas a otros usuarios. De forma análoga podría mezclarse esta idea con alguna similar a la planteada por la plataforma *Edmodo* aunque de nuevo especializado al tema médico de la hepatología.
- Otra aplicación Web que se puede integrar es la del registro de pacientes para almacenar datos y resultados que después puedan ayudar al paciente y al procesado de los datos para la generación de estudios y la extracción de estadísticas. En este caso, al tener que manejar datos privados de los pacientes se tendrían que considerar las implicaciones y restricciones en materia de seguridad pues se trataría de una funcionalidad no abierta para todo el mundo. También podría utilizarse para ayudar a los pacientes con el seguimiento del tratamiento utilizando elementos muy visuales como gráficas, calendarios, recordatorios utilizando el sistema de notificaciones, etc.

- Implementar la posibilidad de abrir y visualizar los recursos PDF sin necesidad de descargar de forma previa el fichero completo sino descargando sobre la marcha sólo las páginas que se van deseando visualizar.
- Se pueden mejorar los mensajes de error para que proporcionen más información relevante, y de forma más amigable y específica, sobre cualquier problema que aparezca en el uso del sistema.
- Se pueden llevar a cabo aún pequeñas pero numerosas mejoras de rendimiento, estabilidad, consumo de memoria, datos y energía. También la corrección de varios errores menores de la interfaz gráfica para lograr que la implementación hecha sea más fiel a la idea original del proyecto. En resumen, llevar a cabo optimizaciones y correcciones de pequeños errores.
- Se puede hacer uso de criptografía para proteger la comunicación entre la máquina servidora y la aplicación estableciendo el cifrado de las comunicaciones y mecanismos de autenticación del cliente que accede e integridad de las

respuestas obtenidas de la máquina servidora. En definitiva, hacer seguro al canal de comunicación empleado extremo a extremo. También podría establecer la base para un futuro sistema de monetización de la aplicación no en busca de la rentabilidad del proyecto sino el de cubrir sus gastos de manutención de forma parcial o total (por ejemplo, si se necesitara ofrecer soporte técnico al incluir funcionalidades más avanzadas como algunas de las descritas antes).

- Algunas otras mejoras técnicas sugeridas en el capítulo *Descripción técnica y sobre el desarrollo de HepApp*. Algunas de estas son el uso de fragmentos para la implementación de las pantallas en la calculadora general (abstracta), la creación de recursos alternativos *dimens.xml* para los directorios *values-sw480dp* y *values-sw720dp*, la aplicación del paquete *java.util.concurrent* tanto para el mecanismo de la sección crítica de la *AsyncTask* como para el temporizador de la espera de la obtención de la respuesta de la petición al servicio Web con *Volley*, permitir que sea configurable el recurso representativo de un recurso compuesto, o el uso de *Enum* en lugar de constantes en interfaces de enteros o *String* para facilitar la mediación entre capas independientes por parte del controlador principal, entre otras.
- Se puede proporcionar al panel Web las funcionalidades para que de forma automática meta el relleno transparente a las imágenes que se suben a la máquina servidora para ajustar las imágenes al ratio alto-ancho recomendado sin deformarlas. Además, que optimice estas reduciendo al máximo su tamaño, como se ha comentado que se ha hecho en el subapartado *Otros*.

Referencias

Alberta Health Services (2017). Hepatocellular Carcinoma. url: https://www.albertahealthservices.ca/assets/info/hp/cancer/if-hp-cancer-guide-gi007-hepatocellular-carcinoma.pdf. Last visited: August, 2018

Android Developers. (2018). url: https://developer.android.com/. Last visited: August, 2018

Android Open Source Project. (2018). url: https://source.android.com/. Last visited: August, 2018

Android PdfViewer. (2018). url: https://github.com/barteksc/AndroidPdfViewer. Last visited: August, 2018

Apache Cordova. (2018). url: https://cordova.apache.org/. Last visited: August, 2018

Apple Developer. (2018). url: https://developer.apple.com/. Last visited: August, 2018

Bishop, J.L. & Verleger, M.A. (2013). The Flipped Classroom: A Survey of the Research. Paper presented at the Proceedings of the 120th ASEE Annual Conference & Exposition, Atlanta (USA). url: http://www.asee.org/file_server/papers/attachment/file/0003/3259/6219.pdf

Bootstrap. (2018). The most popular HTML, CSS, and JS library in the world. url: https://getbootstrap.com/. Last visited: August, 2018

Cabrero Almenara, J. (2006). Bases pedagógicas del *e-learning*. Revista de Universidad y Sociedad del Conocimiento, 3. Issn:1698-580X. url: https://dialnet.unirioja.es/servlet/articulo?codigo=1970689. Last visited: August, 2018

Charland, A. & LeRoux, B. (2011). Mobile Application Development: Web vs. Native. ACM Queue, 9(4). doi: 10.1145/1966989.1968203

Cheng, C. H., Lee, C. F., Wu, T. H., Chan, K. M., Chou, H. S., Wu, T. J., … Chen, M. F. (2011). Evaluation of the new AJCC staging system for resectable hepatocellular carcinoma. World Journal of Surgical Oncology, 9, 114. http://doi.org/10.1186/1477-7819-9-114

Chevret, S., Trinchet, J.-C., Mathieu, D., Rached, A. A., Beaugrand, M. & Chastang, C. (1999). A new prognostic classification for predicting survival in patients with hepatocellular carcinoma: Groupe d'Etude et de Traitement du Carcinome Hépatocellulaire. *J Hepatol, 31*:133–141. doi: 10.1016/S0168-8278(99)80173-1

Child C.G. & Turcotte J.G. (1964). Surgery and portal hypertension. In: Child CG, editor. The liver and portal hypertension: 50-64. Philadelphia: Saunders.

Color Picker. (2018). url: https://materialdoc.com/components/pickers/#color-picker. Last visited: August, 2018

Costello, K. & Hippold, S.C. (2018). Gartner Says Worldwide Sales of Smartphones Returned to Growth in First Quarter of 2018. url: https://www.gartner.com/newsroom/id/3876865. Last visited: August, 2018

De la Mata, M. & Barrera, P. (2003). Gestionar la lista de espera de trasplante hepático. Hepatología Continuada, 2(5): 236-239. url: http://aeeh.es/wp-content/uploads/2012/05/v2n5a143pdf001.pdf

Delía, L.N., Galdámez, N., Thomas, P., & Pesado, P.M. (2013). Un Análisis Experimental de Tipo de Aplicaciones para Dispositivos Móviles. XVIII Congreso Argentino de Ciencias de la Computación (CACIC), pp. 766-776 ; url: https://digital.cic.gba.gob.ar/handle/11746/2091

European Association for the Study of the Liver, EASL (2016). Guías de práctica Clínica EASL: Trasplante hepático. Journal of Hepatology, 64: 433–485. url: http://www.easl.eu/medias/cpg/liver-transplantation/Spanish-report.pdf.

González-Pinto, I., Miyar, A., García Bernardo, C., Rodríguez, M., Barneo, L., Vázquez, L., González Diéguez, L. & Martínez Rodríguez, E. (2007). Criterios mínimos de trasplante hepático: Child, MELD, excepciones. Med Clin, 8(2): 38-41. url: http://www.elsevier.es/es-revista-medicina-clinica-2-pdf-X0025775307105744-S300.

Herrero Santos, J.I. (2011). Trasplante hepático y tumores. Criterios de trasplante hepático en el hepatocarcinoma, ¿pueden ampliarse?. Gastroenterología y Hepatología Continuada, 10(3): 99-102. url: http://aeeh.es/wp-content/uploads/2011/12/v10n3a675pdf001.pdf.

jQuery. (2018). url: https://jquery.com/. Last visited: August, 2018

Kamath, P. S., Wiesner, R. H., Malinchoc, M. , Kremers, W. , Therneau, T. M., Kosberg, C. L., D'Amico, G. , Dickson, E. R. and Kim, W. R. (2001). A model to predict survival in patients with end-stage liver disease. Hepatology, 33: 464-470. doi:10.1053/jhep.2001.22172

Kinoshita, A., Onoda, H., Fushiya, N., Koike, K., Nishino, H., & Tajiri, H. (2015). Staging systems for hepatocellular carcinoma: Current status and future perspectives. World Journal of Hepatology, 7(3), 406–424. http://doi.org/10.4254/wjh.v7.i3.406

Larman, C. (2001). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Second Edition. Prentice Hall.

Lei, H., Chau, G., Lui, W., Tsay, S., King, K., Loong, Ch. & Wu, Ch. (2006). Prognostic value and clinical relevance of the 6th Edition 2002 American Joint Committee on Cancer staging system in patients with resectable hepatocellular carcinoma. Journal of the American College of Surgeons*, 203*, 426-435. doi:10.1002/hep.25680

Leung, T. W., Tang, A. M., Zee, B. , Lau, W. Y., Lai, P. B., Leung, K. L., Lau, J. T., Yu, S. C. and Johnson, P. J. (2002), Construction of the Chinese University Prognostic Index for hepatocellular carcinoma and comparison with the TNM staging system, the Okuda

staging system, and the Cancer of the Liver Italian Program staging system. *Cancer, 94(6)*: 1760-1769. doi:10.1002/cncr.10384

Limaye, R., Bayandorian, H. & Kamil, S. (2018). Model-View-Controller. url: https://patterns.eecs.berkeley.edu/?page_id=42. Last visited: August, 2018

Linares, A., Rodríguez, M., & Rodrigo, L. (2004). Algoritmo diagnóstico y terapéutico del carcinoma hepatocelular. Oncología (Barcelona), 27(4), 95-101. url: http://scielo.isciii.es/scielo.php?script=sci_arttext&pid=S0378-48352004000400019

Llovet, J.M., Brú, C., & Bruix, J. (1999). Prognosis of hepatocellular carcinoma: the BCLC staging classification. *Sem Liv Dis. 19*(3), 329-338. doi:10.1055/s-2007-1007122

Luján-Mora, S. (2002). Programación de aplicaciones web: historia, principios básicos y clientes web. San Vicente (Alicante): Editorial Club Universitario.

Luján-Mora, S. (2001). Programación en Internet: clientes web. San Vicente (Alicante): Editorial Club Universitario.

MariaDB.org. (2018). url: https://mariadb.org/. Last visited: August, 2018

Martinez-Miera, G., Esquivel-Torres, S., Medina Granados, J.P., Luna-Castillo, M., Castillo-Chiquete, R., Calzada-Grijalva, J.F. & Gonzalez-Velazquez, F. (2014). Presentation, staging, and outcome of patients with hepatocellular carcinoma at a center in Veracruz, Mexico. Revista de Gastroenterología de México, 79(3):171–179. url: http://www.revistagastroenterologiamexico.org/es-presentation-staging-outcome-patients-with-articulo-S2255534X14000656

Matthews, J. C., Pagani, F. D., Haft, J. W., Koelling, T. M., Naftel, D. C., & Aaronson, K. D. (2010). Meld Score Predicts Lvad Operative Transfusion Requirements, Morbidity, and Mortality. Circulation, 121(2), 214–220. http://doi.org/10.1161/CIRCULATIONAHA.108.838656

Mazzaferro, V., Llovet , J., Miceli, R., Bhoori, S., Schiavo, M., Mariani, L. et al (2009). Predicting survival after liver transplantation in patients with hepatocellular carcinoma beyond the Milan criteria: a retrospective, exploratory analysis. *Lancet Oncol, 10*(1), 35-43. doi:10.1016/S1470-2045(08)70284-5

Mazzaferro V, Regalia E, Doci R, Andreola S, Pulvirenti A, Bozzetti F, et al. (1996). Liver transplantation for the treatment of small hepatocellular car- cinomas in patients with cirrhosis. N Engl J Med, 334: 693-699.

Microsoft Xamarin. (2018). Desarrollo de aplicaciones de Xamarin con Visual Studio. url: https://visualstudio.microsoft.com/es/xamarin/. Last visited: August, 2018

Myers R.P., Shaheen A.A.M., Faris P., Aspinall A.I. & Burak K.W. (2013). Revision of MELD to Include Serum Albumin Improves Prediction of Mortality on the Liver Transplant Waiting List. PLOS ONE, 8(1): e51926. https://doi.org/10.1371/journal.pone.0051926

Okuda, K. , Ohtsuki, T. , Obata, H. , Tomimatsu, M. , Okazaki, N. , Hasegawa, H. , Nakajima, Y. and Ohnishi, K. (1985), Natural history of hepatocellular carcinoma and prognosis in relation to treatment study of 850 patients. Cancer, 56: 918-928. doi:10.1002/1097-0142(19850815)56:4<918::AID-CNCR2820560437>3.0.CO;2-E

PhoneGap. (2018). url: https://www.phonegap.com/. Last visited: August, 2018

PHP. (2018). Manual de PHP. url: https://secure.php.net/manual/es/. Last visited: August, 2018

Pin Vieito N., Guerrero Montañés A. & Delgado Blanco M. (2014). Hepatocarcinoma: estado actual. Galicia Clin, 75 (4): 171-181. url: https://dialnet.unirioja.es/descarga/articulo/4906471.pdf

Pons, F., Varela, M., & Llovet, J. M. (2005). Staging systems in hepatocellular carcinoma. HPB : The Official Journal of the International Hepato Pancreato Biliary Association, 7(1), 35–41. http://doi.org/10.1080/13651820410024058

Sin, D., Lawson, E. & Kannoorpatti, K. (2012). Mobile web apps – the non-programmer's alternative to native applications. 5th International Conference on Human System Interactions (HSI), Perth, WA, pp. 8-15. doi: 10.1109/hsi.2012.11

Sorensen J.B., Klee M., Palshof T. & Hansen H.H. (1993). Performance status assessment in cancer patients. An inter-observer variability study. Br J Cancer, 67:773-5.

StatCounter Global Stats. (2018). Browser, OS, Search Engine including Mobile Usage Share. url: http://gs.statcounter.com/. Last visited: August, 2018

Statista. (2018). The portal for statistics. url: https://www.statista.com/. Last visited: August, 2018

The Cancer of the Liver Italian Program (CLIP) Investigators (1998). A new prognostic system for hepatocellular carcinoma: A retrospective study of 435 patients. Hepatology, 28: 751-755. doi:10.1002/hep.510280322

The Cancer of the Liver Italian Program (CLIP) Investigators (2000), Prospective validation of the CLIP score: A new prognostic system for patients with cirrhosis and hepatocellular carcinoma. Hepatology, 31: 840-845. doi:10.1053/he.2000.5628

Toso, C. , Meeberg, G. , Hernandez-Alejandro, R. , Dufour, J. , Marotta, P. , Majno, P. and Kneteman, N. M. (2015). Total tumor volume and alpha-fetoprotein for selection of transplant candidates with hepatocellular carcinoma: A prospective validation. Hepatology, 62: 158-165. doi:10.1002/hep.27787

Wiesner, R., Edwards, E., Freeman, R., Harper, A., Kim, R., Kamath, P., Kremers, W., Lake, J., Howard, T., Merion, R.M., Wolfe, R.A. & Krom, R. (2003). Model for end-stage liver disease (MELD) and allocation of donor livers. *Gastroenterology, 124(1)*: 91-6.

Xataka Android. (2018). url: https://www.xatakandroid.com/. Last visited: August, 2018

Yao, F. Y., Ferrell, L. , Bass, N. M., Bacchetti, P. , Ascher, N. L. & Roberts, J. P. (2002). Liver transplantation for hepatocellular carcinoma: Comparison of the proposed UCSF criteria with the Milan criteria and the Pittsburgh modified TNM criteria. *Liver Transpl, 8*: 765-774. doi:10.1053/jlts.2002.34892

Yao, F. Y., Xiao, L. , Bass, N. M., Kerlan, R. , Ascher, N. L. and Roberts, J. P. (2007), Liver Transplantation for Hepatocellular Carcinoma: Validation of the UCSF-Expanded Criteria Based on Preoperative Imaging. American Journal of Transplantation, 7: 2587-2596. doi:10.1111/j.1600-6143.2007.01965.x