Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Mención en Tecnologías de la Información

Automatización de la calificación de pruebas objetivas de tipo test en papel mediante móviles

Autor:

D. Sergio Escanciano Arribas

Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Mención en Tecnologías de la Información

Automatización de la calificación de pruebas objetivas de tipo test en papel mediante móviles

Autor:

D. Sergio Escanciano Arribas

Tutor:

D. César Llamas Bello

Resumen

El sistema desarrollado en este proyecto automatiza el reconocimiento y corrección de exámenes de tipo test mediante el uso de un dispositivo móvil habitual. Está compuesto por un software de escritorio para la generación y visualización de los tests, un servidor para almacenar, gestionar y corregir exámenes y finalmente una aplicación para dispositivos móviles que permite realizar el reconocimiento y corrección de los tests.

La principal característica diferenciadora de este trabajo es que permite realizar la corrección de manera rápida y sencilla, sin necesidad de un hardware específico y costoso. Como base de lo indicado se ha creado una plantilla de exámenes tipo test que es reconocida inequívocamente por el sistema y que otorga flexibilidad en su diseño para poder abarcar exámenes de diferentes características.

Además, proporciona tanto una modalidad de examen que se puede corregir directamente desde el dispositivo móvil al momento, como otra que permite enviar el test escaneado al servidor para que pueda ser corregido y analizado por el profesor utilizando la aplicación de escritorio.

Abstract

The system developed in this project automatize the recognition and correction of tests with the help of a common mobile device. This is composed of a software for the generation and visualization of tests, a server to store, manage and correct tests and, finally, an application for mobile devices that allows the user to recognize and correct the generated tests.

The main and unique feature of this project is that it allows to realize the correction in a quick and easy way, without the need of specific and expensive hardware. To support the system, a test template has been developed. It is unequivocally recognized by the software and gives enough flexibility in its design to allow the creation of different types of tests.

Also, it provides a type of exam that can be corrected directly from the mobile devices, and another type that allows to send the scanned test to the server in order to be corrected, and finally analyzed by the teacher using the desktop application.

Índice general

Índice de figuras

Índice de tablas

Presentación

Una de las principales motivaciones para la realización de este Trabajo Fin de Grado fue el poder diseñar una aplicación que, más allá de su valor académico e instructivo, pudiera presentar la base de una solución para una necesidad real.

Dentro del ambiente universitario es común el uso de exámenes de tipo test para comprobar los conocimientos del alumno sobre una materia determinada. El referirse específicamente a pruebas de tipo test no es arbitrario; éstas tienen unas restricciones que permiten definir un formato cerrado, que posibilita ejecutar un patrón de reconocimiento y a la vez mantener versatilidad en su diseño para abarcar un gran número de pruebas sobre distintos contenidos.

Normalmente estos tests son corregidos a mano por el profesor o utilizan un formato y medio físico específico que son reconocidos mediante máquinas ensambladas para dicho propósito. Por tanto, en general la corrección rápida y en masa de tests sólo es posible mediante el uso de hardware específico.

El trabajo recoge el diseño e implementación de un sistema que permite la corrección de exámenes de tipo test mediante el uso de una aplicación para dispositivos móviles *Android*[1]. Estos exámenes siguen una plantilla propia, sencilla para el profesor y el alumno, y que permite un reconocimiento automatizado, rápido y fiable sin la necesidad de adquirir un hardware especializado y normalmente costoso.

Este documento está esquematizado en tres capítulos introductorios, dos partes principales - estructuradas a su vez con varias secciones y sub-secciones -, y dos capítulos a modo de epílogo.

1. **Introducción**. Se describen elementos generales del trabajo, la tecnología utilizada, el formato de tests diseñado y la planificación utilizada.
   - 1.1 **Introducción y alcance del proyecto**. Indica las características principales del sistema.
   - 1.2 **Contexto tecnológico del problema**. Indica la tecnología utilizada para el desarrollo del proyecto y al escritura de este documento.
   - 1.3 **Realización de exámenes tipo test: planteamiento y recursos**. Recoge el diseño y desarrollo de las plantillas de examen utilizadas por el sistema
   - 1.4 **Planificación**. Indica el método planificación utilizado, así como detalles importantes del desarrollo del mismo.
2. **Parte I. Análisis del problema**. En esta primera parte se describen el contexto del problema y los requisitos, actores y casos de uso identificados durante el análisis. Además, incluye un diagrama de clases de análisis de alto nivel que ayuda en la identificación del problema.
   - 2.1 **Análisis y requisitos**. Detalla los requisitos funcionales y no funcionales identificados y los actores y casos de uso derivados.
   - 2.2 **Diagrama de clases de análisis**. Muestra el diagrama de clases de análisis de alto nivel.

[1] `https://www.android.com/intl/es_es/`

3. **Parte II. Diseño de la solución propuesta**. En esta segunda parte se detalla la solución propuesta para el problema identificado en los capítulos anteriores. Incluye un capítulo con el planteamiento general del sistema y varios capítulos dedicados en exclusiva a cada una de las aplicaciones que lo componen.

    3.1 **Planteamiento general**. Muestra el diseño general del sistema propuesto a partir del análisis. Incluye diagramas genéricos y *UML* de secuencia, despliegue, etc.

    3.2 **Aplicación de generación y visualización de tests**. Describe en detalle la aplicación de generación y visualización de tests: su interfaz de usuario, la arquitectura y su implementación.

    3.3 **Aplicación de escaneado de tests**. Describe en detalle la aplicación de escaneado de tests: su interfaz de usuario, la arquitectura y su implementación. Además, y dado que es uno de los pilares del proyecto, se explica el método utilizado para reconocer los exámenes.

    3.4 **Aplicación servidor**. Describe en detalle la aplicación servidor: la arquitectura, su implementación y los componentes que la forman - base de datos y repositorio -.

4. **Epílogo**. En estos últimos capítulos se valora todo lo desarrollado.

    4.1 **Evolución del proyecto**. Indica la evolución que ha sufrido el trabajo desde su concepción inicial.

    4.2 **Conclusiones**. En este capítulo se realiza una reflexión y valoración sobre el trabajo realizado. Además, se indican las posibilidades de ampliación del mismo.

La memoria incluye también varios anexos que complementan la información mostrada.

1. **Anexo A. Manual de usuario de la aplicación de generación y visualización**. Incluye el manual de usuario de la aplicación de generación y visualización de tests.

2. **Anexo B. Manual de usuario de la aplicación servidor**. Incluye el manual de usuario de la aplicación servidor.

3. **Anexo C. Manual de usuario de la aplicación de escaneado de tests**. Incluye el manual de usuario de la aplicación de escaneado de tests.

4. **Anexo D. Información incluida en el soporte digital**. Indica el contenido del soporte digital entregado con toda la información del proyecto.

Capítulo 1

Introducción y alcance del proyecto

Este documento describe el proceso de diseño, desarrollo e implementación de un pequeño ecosistema destinado a la generación, corrección y análisis de pruebas objetivas de tipo test. Los exámenes son creados en una aplicación de escritorio, dando como resultado un fichero que será impreso para ser corregido mediante una aplicación para dispositivos móviles *Android*. Si el usuario desea que la prueba tenga un carácter oficial, sea corregida de manera transparente a los receptores del examen y sea almacenada para ser posteriormente revisada, entra en juego un servidor que realiza dichas tareas.

Se ha propuesto abarcar las principales fases de realización de un examen para permitir que el profesor pueda ejecutar una prueba de tipo test sobre sus alumnos sin la necesidad de otro tipo de software o hardware. Se pueden identificar las siguientes etapas:

1. **Diseño y generación del examen**: El profesor diseña el test con sus preguntas y respuestas - y diferentes versiones si lo considera necesario -, introduce los datos básicos del mismo - instrucciones, titulación, convocatoria, etc. -, y lo genera para ser impreso.
2. **Realización del examen por parte de los alumnos**: El profesor entrega el examen impreso a sus alumnos y éstos lo rellenan.
3. **Corrección del examen**: El profesor corrige los exámenes y analiza sus resultados.

Siguiendo lo indicado, las características del proyecto son:

- **Aplicación de escritorio para la generación y visualización de tests**. A partir de una plantilla básica permite definir un examen bajo las siguientes posibilidades:
  - **Especificación de información administrativa**:
    - **Número de plan**.
    - **Número de asignatura**.
    - **Número de grupo**.
    - **Convocatoria**.
    - **Fecha del examen**.
    - **Titulación**.
    - **Asignatura**.
    - Opción de añadir código *NIA - Número de identificación de alumno* - del alumno a un código *QR*[1] - *Quick Response Code* -, como se especificará más adelante.

[1] `https://en.wikipedia.org/wiki/QR_code`

- **Dos modos de examen**:
  - **Oficial**. Se especificará el código *NIA* de los alumnos a los que va dirigido el examen y se creará una copia para cada uno de ellos. Las copias contendrá un código *QR* con la información de cada alumno. Mediante estos datos, la aplicación de reconocimiento será capaz de enviar los exámenes analizados a un servidor - definido por el profesor -, para su posterior corrección.
  - **Autocorrección**. Añade las respuestas del test al código QR del examen. En este modo, cualquier usuario con el programa de reconocimiento podrá escanear el test y recibir el resultado directamente desde la aplicación, sin tener que acceder a un servidor externo.
- **Ocho versiones de respuestas**: El profesor podrá especificar hasta ocho versiones de respuestas distintas para otorgar diferentes tests a sus alumnos.
- **Cincuenta preguntas por examen**: Se podrán definir hasta cincuenta preguntas por test.
- **Ocho respuestas por pregunta**: El profesor podrá especificar hasta ocho respuestas por pregunta.
- **Guardado del examen generado en formato *SVG*[2] - *Scalable Vector Graphics* -, y *PDF*[3] - *Portable Document Format* -**.

Así mismo, el programa permitirá ver los exámenes que hayan sido almacenados en la aplicación servidor. Esta característica incluye lo siguiente:

- **Recepción y filtrado exámenes del servidor**. El software recibe los datos almacenados en red y los muestra al profesor, permitiéndole filtrar los exámenes según su información administrativa.
- **Petición de corrección de exámenes**. Permite la corrección de uno o todos los tests aún no corregidos en el servidor.
- **Exportación de datos de corrección**. Exporta los datos de uno o todos los tests corregidos al formato estándar *CSV*[4] - *Comma-separated values* -, para poder ser analizados por programas externos.

- **Aplicación servidor para el almacenado y corrección de tests**. Servidor que almacena los datos de los exámenes en modo *Oficial* generados y escaneados, encargándose también de la corrección de los mismos.
- **Aplicación móvil para el escaneado de tests**. Permite a los usuarios escanear los tests realizados desde la aplicación de escritorio. Dependiendo del modo de examen, el programa mostrará el resultado del test -modo *Autocorrección*-, o enviará los datos a la aplicación servidor -modo *Oficial*-.

[2] `https://en.wikipedia.org/wiki/Scalable_Vector_Graphics`
[3] `https://en.wikipedia.org/wiki/Portable_Document_Format`
[4] `https://en.wikipedia.org/wiki/Comma-separated_values`

Capítulo 2

Contexto tecnológico del problema

Para la construcción del ecosistema necesario para que las tres aplicaciones convivan es necesaria la elección de una tecnología acorde, moderna, estable y que permita una intercomunicación sencilla y limpia. El proyecto se concibió con tres restricciones tecnológicas básicas:

- Debía realizarse para dispositivos *Android*, sin exigir características extras más allá de una cámara. Además se pretendía que la aplicación fuera funcional en la mayoría de dispositivos móviles actuales.
- Debía tener una o varias aplicaciones que formaran un servidor.
- Debía tener una aplicación web o de escritorio que permitiera el manejo de los tests de la manera más sencilla posible.

Dado que el lenguaje de programación principal en *Android* es *Java*[1] y que tiene una enorme compatibilidad multiplataforma gracias a su ejecución en máquina virtual -*Java VM*[2]-, se decidió implementar todos los programas utilizando este lenguaje como base, ya que se podía utilizar con facilidad para el desarrollo del servidor mediante *Java EE*[3] - *Java Enterprise Edition* -, y la aplicación de escritorio podría realizarse con un interfaz gráfico *Swing*[4].

Antes de detallar cada aplicación, cabe indicar que todo el desarrollo se ha llevado a cabo sobre el sistema operativo Microsoft Windows 10 en su versión Home y en un arquitectura x64.

2.1. Aplicación de generación y visualización de tests

Esta es la primera aplicación que utilizará el usuario del sistema y, como su nombre indica, sirve para generar los exámenes y visualizar los resultados de los tests del modo *Oficial*. Está escrita en *Java 8*, con *NetBeans*[5] como entorno de desarrollo y utilizando las siguientes características y bibliotecas externas:

- **Swing**. Es una biblioteca de *Java*, parte de las *Java Foundation Classes*[6] de *Oracle*[7], que se utilizan para el diseño de interfaces gráficas. Está ampliamente documentada, abstrae las peculiaridades de cada plataforma y tiene un gran soporte en *NetBeans* gracias al editor gráfico que proporciona.

[1] `https://www.java.com/es/`
[2] `https://en.wikipedia.org/wiki/Java_virtual_machine`
[3] `http://www.oracle.com/technetwork/java/javaee/overview/index.html`
[4] `https://en.wikipedia.org/wiki/Swing_(Java)`
[5] `https://netbeans.org/`
[6] `https://en.wikipedia.org/wiki/Java_Foundation_Classes`
[7] `https://www.oracle.com/es/index.html`

- **ZXing**[8]. Es una biblioteca *Open Source* de procesado de códigos de barras de varios formatos. Es la más utilizada en *Java* y en *Android*, y es usada tanto en esta aplicación como en la de dispositivos móviles. En este caso se encarga de la generación del código *QR*.
- **Apache Batik**[9]. Es un conjunto de bilbliotecas de la fundación *Apache*[10] centrada principalmente en la generación, manipulación y visualización de gráficos vectoriales *SVG*. Se utiliza para la generación de los exámenes -ya que la plantilla está realizada en formato *SVG*-, y su conversión a *PDF*.
- **JSONUtil 1.5**[11]. Es una pequeña biblioteca para la generación y análisis de ficheros *JSON*[12] - *JavaScript Object Notation* -. Es sencilla y consume poco, por lo que es utilizada tanto aquí como en el servidor, principalmente para la comunicación de información mediante los servicios *REST*[13] - *Representational State Transfer* -, diseñados.

El uso de la versión 8 de Java, más allá del hecho de utilizar la iteración más moderna, proporciona varias nuevas propiedades que han sido utilizadas en el código. Entre ellas se pueden destacar las funciones lambda o los nuevos métodos más sencillos de lectura y escritura de ficheros.

En cuanto a los programas utilizados, la aplicación se ha realizado íntegramente con el entorno de desarrollo *NetBeans* en su versión 8.0.2. Sobre la plantilla del examen se hablará con más detalle en secciones posteriores, pero cabe destacar que se diseñó con el programa *Inkscape*[14] en su versión 0.9.1.

2.2. Aplicación servidor

El servidor está compuesto de tres componentes distintos que forman la aplicación completa. Al igual que la aplicación de escritorio, el proyecto Java se ha creado utilizando *NetBeans*.

2.2.1. Servidor principal GlassFish con Java

Éste es el servidor principal, ya que es el que recibe las peticiones del exterior y las gestiona utilizando los servicios de bases de datos y repositorio de información que se indicará más adelante.

Se trata de un servicio web de tipo *REST* que ofrece varios métodos de tipo *POST* para gestionar la generación, visualización y corrección de exámenes. Utiliza *GlassFish*[15] - un servidor de aplicaciones *Java EE* - para ejecutar la aplicación y, al igual que la aplicación de escritorio, se ha desarrollado utilizando *Java 8*. Exactamente el programa funciona sobre un servidor *GlassFish* en su versión 4.1, ya que versiones anteriores o posteriores tienen cambios en ciertos módulos que producen errores en la ejecución del software.

Cabe destacar que para la corrección de exámenes se utiliza la última versión *Java* de la librería *OpenCV*[16] *3.1*. Esta versión es un *binding* de la implementación original en *C++*, utilizando la interfaz *JNI* - un *framework* que permite la llamada de código nativo *C* y *C++* desde *Java* - como base; por ello, requiere el enlazado de una biblioteca de código nativo -en este caso, una *DLL* para *Windows* con arquitectura *x64*-. Esto hace perder en parte

[8] https://github.com/zxing/zxing
[9] https://xmlgraphics.apache.org/batik/
[10] http://www.apache.org/
[11] https://github.com/billdavidson/JSONUtil/
[12] http://www.json.org/
[13] https://en.wikipedia.org/wiki/Representational_state_transfer
[14] https://inkscape.org/es/
[15] https://glassfish.java.net/
[16] http://opencv.org/

la facilidad multiplataforma directa que da Java, pero al ser OpenCV una librería muy popular, es posible compilar el código para un gran número de sistema operativos -entre ellos, los principales sistemas *Windows*, *Linux* y *Mac OS*-.

2.2.2. Servidor de base de datos

Para almacenar los datos funcionales de los tests era necesario emplear un sistema de base de datos. Primeramente se decidió que se utilizaría una base de datos relacional bajo el lenguaje de bases de datos *SQL*[17] - *Structured Query Language* -, ya que ambos sistemas tienen una gran aceptación y un largo recorrido, además de un soporte directo en *Java*. Tras este punto, surgió una duda a la hora de formar la arquitectura del servidor. Por una parte, el separar el servidor de base de datos en un programa y proceso aparte permitía desacoplar la base de datos del resto del sistema, dando cierta libertad al usuario para desplegarlo como deseara -manteniendo el esquema de tablas dado requerido por el sistema-, pero provocaba dos problemas principales:

- **Conexión del servidor principal con el de base de datos**. Una vez compilado el servidor principal, éste debe saber la ruta de acceso al servidor de base de datos. Esto se puede conseguir escribiendo una ruta fija dentro del programa o indicándole que acceda a ella en algún tipo de archivo de configuración externo. Ambas alternativas producen un resultado inadecuado para un proyecto de estas características, ya que dejar una ruta fija quita flexibilidad al programa y el tener que indicar un fichero de configuración provoca utilizar ficheros y programas externos al propio servidor.

- **Creación de las tablas necesarias**. El sistema requiere un esquema específico de tablas, por lo que además de instalar el servidor se debe ejecutar un *script* para generarlas con el formato adecuado.

Además de lo indicado, dificultaba la distribución y el despliegue del sistema, ya que cada vez que se instalara en un nuevo sistema se debían realizar varias instalaciones y configuraciones. Para solucionarlo, se implementó una interfaz común para el acceso a la base de datos, que utiliza a su vez el interfaz *JDBC*[18] - *Java Database Connectivity* -. De esta forma, se dan dos posibilidades a la hora de compilar y distribuir el sistema:

- **Servidor de base de datos externo**. Requiere la instalación y configuración de un servidor externo. Se decidió usar *MariaDB*[19], ya que es un sistema derivado de *MySQL* -creado por los mismos desarrolladores originales-, de código abierto y una gran aceptación, siendo utilizado, entre otros, por *Facebook* y *Wikipedia*. Además, la gran mayoría de órdenes y parámetros son exactamente iguales que en *MySQL*, por lo que su uso requiere un corto periodo de aprendizaje si se conoce el otro sistema. Para su acceso desde la aplicación Java se utiliza el cliente oficial *MariaDb Java Client*.

- **Servidor de base de datos embebido**. Viene incluído dentro del propio servidor principal, y cuando es utilizado por primera vez inicializa automáticamente la base de datos; además, ésta queda almacenada en un fichero individual dentro del directorio donde se desplegó el servidor para poder facilitar su traslado y la recuperación de datos. En esta alternativa se utilizó la biblioteca *SQLite*[20], con su implementación *SQLite JDBC*[21], ya que proveía el mismo interfaz que el caso del servidor externo y otorgaba todas las características necesarias.

[17] `https://en.wikipedia.org/wiki/SQL`
[18] `https://en.wikipedia.org/wiki/Java_Database_Connectivity`
[19] `https://mariadb.org/`
[20] `https://www.sqlite.org/`
[21] `https://bitbucket.org/xerial/sqlite-jdbc`

2.2.3. Repositorio de imágenes Apache Jackrabbit

Junto al guardado de información funcional y administrativa de los exámenes hacía falta un servicio donde poder guardar las fotos de los tests. Tanto las utilizadas como resguardo oficial de la corrección como la propia imagen utilizada.

Esto podía realizarse en la propia base de datos, pero no es la manera más adecuada de hacerlo. Además, de cara a practicar el despliegue de un sistema web más complejo, el añadido de un sistema de repositorio aportaba un interés extra. Un repositorio de contenido es un sistema para el almacenado de datos de manera jerárquica, permitiendo además la realización de diferentes tareas.

Entre varias opciones, se decidió utilizar *Apache Jackrabbit*[22], que es un repositorio de contenido de la fundación *Apache* que contiene una implementación completa del *API* JCR - *Content Repository for Java Technology* -, indicado en las especificaciones *JSR 170* y *JSR 283*. Al igual que en el caso del servidor de base de datos, surgió la cuestión de utilizar un repositorio externo o uno embebido. Aquí la solución fue mucho más sencilla, ya que *Jackrabbit* incluye un servidor independiente en formato *JAR* que puede ser utilizado desde la línea de comandos y como biblioteca Java del proyecto del servidor, creando entonces un repositorio *Jackrabbit* completo dentro de la ruta donde se desplegó el servidor principal.

Siguiendo lo explicado anteriormente, la integración completa en Java permite una interconexión entre aplicaciones sencilla y coherente.

2.3. Aplicación de escaneado de tests

La última sub-sección de este apartado es la aplicación móvil para el escaneado de tests. A pesar de no ser la herramienta principal para el proyecto - ya que lo primero a realizar es el diseño y generación de los exámenes -, constituye la aplicación más interesante, ya que incluye el uso de código nativo en *Android* y la primera y principal implementación del algoritmo de reconocimiento de exámenes - que posteriormente fue portado a código Java para ser utilizado por el *binding Java* de *OpenCV* en el servidor web -.

Está escrita principalmente en *Java* -en este caso sin utilizar ninguna característica particular de la versión 8-, y está diseñada para dispositivos con sistema operativo *Android 5.0 Lollipop - API 21* - o superior. Utiliza ciertas características introducidas a partir de dicha versión de *Android*, como los botones de estilo *Material Design*. A pesar del uso de una versión moderna del sistema operativo, para garantizar una mayor compatibilidad se ha utilizado el *API* de la cámara *android.hardware.Camera* en contra del más actual *android.hardware.Camera2*.

Para el algoritmo de reconocimiento de tests se ha utilizado la biblioteca *OpenCV* en su versión 3.1. De manera similar a lo indicado en el servidor, ésta posee una implementación *Java* para *Android*, pero para conseguir un mayor rendimiento - algo crítico en computación de imagen en tiempo real en dispositivos móviles -, y tener un mayor control sobre el código, se optó por utilizar la versión en código nativo *C++*. Para su ello fue necesario la configuración y uso del *Android NDK*[23] *- Native Development Kit -*.

Inicialmente esta aplicación se empezó a desarrollar utilizando el nuevo entorno de desarrollo de *Android*, *Android Studio*[24]; éste está desarrollado por *Google* y es el propuesto como entorno oficial y principal, ya que, entre otras cosas, incluye varias facilidades y nuevas características que no tienen otras alternativas. No obstante, durante la realización del proyecto dicho entorno se encontraba en sus primeras versiones estables, y su soporte del *NDK* no era ni es adecuado - todavía está centrado en el desarrollo de aplicaciones puramente *Java* -. Por todo ello, el entorno finalmente utilizado fue Eclipse, que a día de hoy sigue siendo el programa de desarrollo principal

[22] `http://jackrabbit.apache.org/jcr/index.html`
[23] `https://developer.android.com/ndk/index.html?hl=es`
[24] `https://developer.android.com/studio/index.html?hl=es`

utilizado para *Android* y el que ofrece una mejor integración con el *NDK*, ya que también es un *IDE* para *C* y *C++*.

2.4. Escritura de la memoria

Este documento ha sido escrito utilizando el sistema LaTeX. Esta decisión se tomó por el interés de utilizar uno de los sistemas estándar para la escritura de documentos técnicos y científicos, y para poder conseguir un acabado visual adecuado manteniendo un documento lo más manejable posible.

Al igual que el resto del proyecto, se ha realizado bajo Windows, por lo que se ha utilizado la implementación *MiKTeX*[25], que ofrece una alternativa actualizada, con un completo sistema de gestión de dependencias y fácil de instalar y configurar. Como complemento a esto se ha empleado el programa *TeXstudio*[26], un entorno de escritura que facilita el uso de funciones y la compilación a *PDF* de LaTeX.

Para la realización de los diagramas *UML*[27] *-Unified Modeling Language-*, se ha utilizado el software *Astah* en su versión *Community Edition*[28]. En cuanto al diagrama general del sistema que no utiliza *UML*, se ha diseñado utilizando *Microsoft Visio 2016*.

[25] `http://miktex.org/`
[26] `http://www.texstudio.org/`
[27] `http://www.uml.org/`
[28] `http://astah.net/editions/community`

Capítulo 3

Realización de exámenes tipo test: planteamiento y recursos

Para comenzar a hablar sobre el esquema del proyecto, se debe describir primeramente la plantilla de tests que se diseñó como base del sistema. A pesar de existir varias aplicaciones, todo el trabajo gira en torno a la corrección de exámenes generados a partir de un modelo, por lo que un diseño adecuado se antojó fundamental.

La primera característica a definir fue el formato de fichero. Dado que se quería realizar un sistema lo más abierto y estándar posible, el desarrollo de un formato propio quedó descartado desde el principio. Entre los formatos adecuados debía ser uno suficientemente extendido, que pudiera ser modificado fácilmente por programas externos, que pudiera ser leído y alterado desde el código del programa y que ofreciera un traslado o exportación directo a *PDF*. Esto último se debe a que es el formato estándar a la hora de distribuir ficheros para ser impresos sufriendo el menos número de incompatibilidades entre diferentes equipos, sistemas operativos y programas.

Se decidió entonces el uso de ***SVG***. Ésta es una especificación de gráficos vectoriales bidimensionales que utiliza el formato *XML*[1] - *Extensible Markup Language* -. Además de cumplir todas las características antes indicadas, el emplear gráficos vectoriales permite crear exámenes independientes de resolución, que pueden ser redimensionados al tamaño y definición que sea necesario; algo muy interesante de cara a la impresión de los mismos.

Investigando a partir de esta decisión, se encontró la biblioteca *Apache Batik*, que permite abrir, modificar y exportar fácilmente ficheros *SVG* en Java. Entre sus funciones también estaba la de guardar el fichero cargado en formato *PDF*, así que rápidamente se consideró que su uso era una elección adecuada.

[1] `https://en.wikipedia.org/wiki/XML`

3.1. Especificación general

El diseño de la plantilla se realizó utilizando el programa gratuito y de código abierto *Inkscape*. Es un software muy extendido entre la comunidad del diseño vectorial y es la alternativa libre más cercana al, más extendido en el mercado laboral y más completo, *Adobe Illustrator*[2]. No obstante para las herramientas necesarias para la confección de la plantilla tiene unas opciones más que suficientes.

Primeramente, para dar una imagen general, se muestra un test de ejemplo generado a partir de la plantilla.

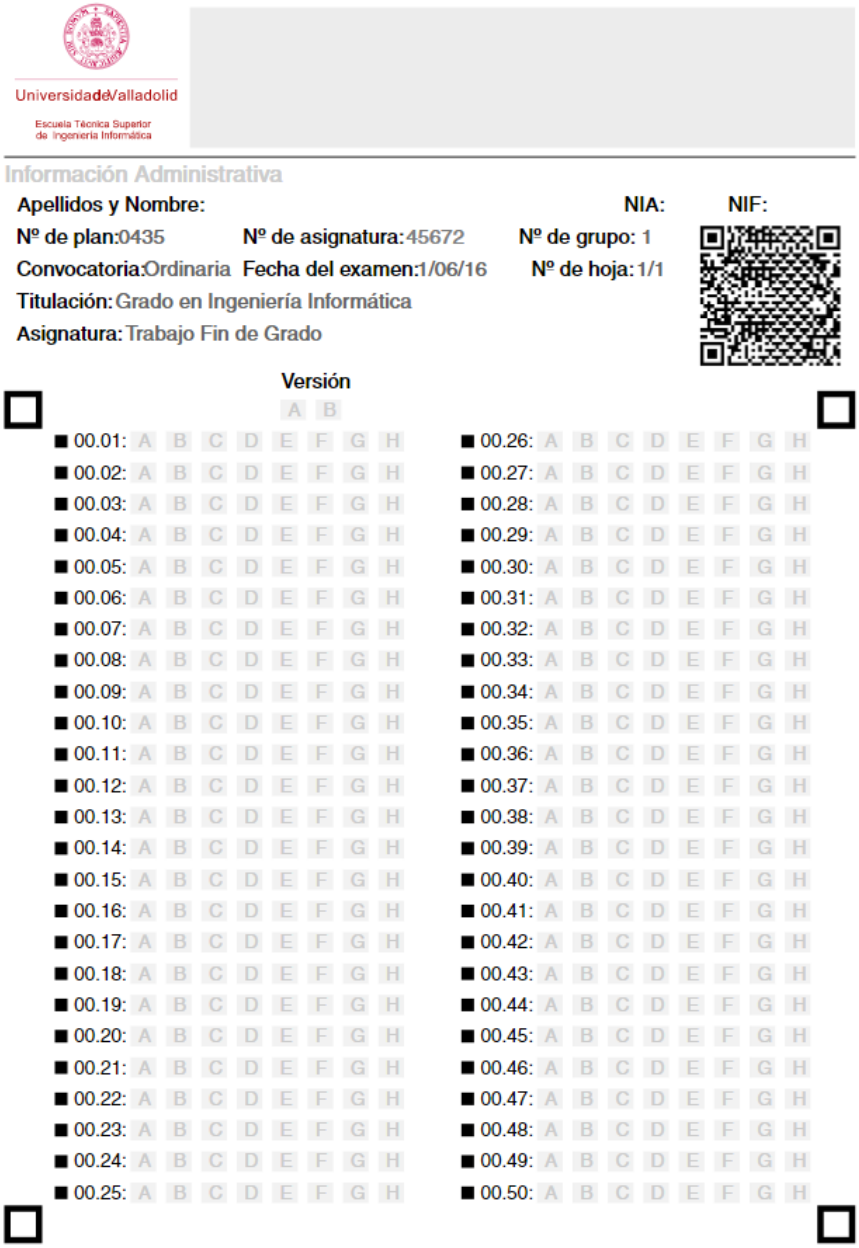

Figura 3.1: Test de ejemplo

[2] `http://www.adobe.com/es/products/illustrator.html`

Los campos básicos de la plantilla debían ser los adecuados para representar toda la información administrativa del examen; para su elección se tomó como base los datos necesarios para un test de la Universidad de Valladolid.

- **Logotipo de la Universidad de Valladolid**. Pequeña imagen situada en la esquina superior izquierda de la plantilla para indicar la procedencia del test.
- **Nombre y apellidos del alumno**. Campo en blanco que será rellenado por el alumno receptor del examen.
- **Número de identificación del alumno -*NIA*-**. Campo en blanco en el que el alumno colocara su *NIA*.
- **Número de identificación fiscal -*NIF*-**. Campo en blanco en el que el alumno escribirá su *NIF*.
- **Número de plan**. Número de identificación del plan de estudios.
- **Número de asignatura**. Número de identificación de la asignatura.
- **Número de grupo**. Número de identificación del grupo.
- **Convocatoria**. Título de la convocatoria a la que pertenece el examen.
- **Fecha del examen**. Fecha de realización del examen con el formato *DD/MM/AAAA*.
- **Titulación**. Nombre de la titulación.
- **Asignatura**. Nombre de la asignatura.

Además, se añadieron otros componentes necesarios para el uso y corrección de la plantilla:

- **Instrucciones**. Campo indicado por el profesor a la hora de generar el test. Éste permite especificar unas instrucciones o apuntes relativos a la realización y corrección del examen, o sobre cualquier otro punto que el profesor considere necesario.
- **Código *QR***. En este código se incluye parte de la información administrativa - en forma de *ID* -, y algunos datos internos necesarios para el reconocimiento del test. Estos contenidos se desarrollarán más adelante.

Las preguntas, versiones y respuestas se encuentran situadas en una sección de la plantilla denominada como *zona de test*. Ésta es la zona que es escaneada y analizada por el algoritmo de reconocimiento. Esta zona queda delimitada por unos cuadros colocados en sus esquina - denominados *cuadros de reconocimiento* e inspirados en los recuadros que contiene el código *QR* -, y su funcionalidad dentro del proyecto será indicada en capítulos posteriores. Esta zona incluye los siguientes parámetros:

- **Versiones del test**. Cada versión define una colección distinta de respuestas para cada pregunta. Pueden existir entre una y ocho versiones, y todas ellas tiene el mismo número de preguntas y respuestas por pregunta.
- **Preguntas**. Son las preguntas del test. Puede haber desde una hasta cincuenta, numeradas desde el uno.
- **Respuestas por pregunta**. Cada pregunta podrá tener entre una y ocho respuestas. Todas las preguntas tendrán el mismo número de respuestas.

Cabe destacar que la colocación de la información administrativa, el código *QR* y la *zona de test* dentro de la plantilla se debe a que a la hora de escanear y hacer foto al examen, tiene que verse en la misma imagen toda la información para obtener un resguardo de corrección adecuado, tanto para la posterior revisión del profesor como para las posibles dudas o reclamaciones que pueda plantear un alumno.

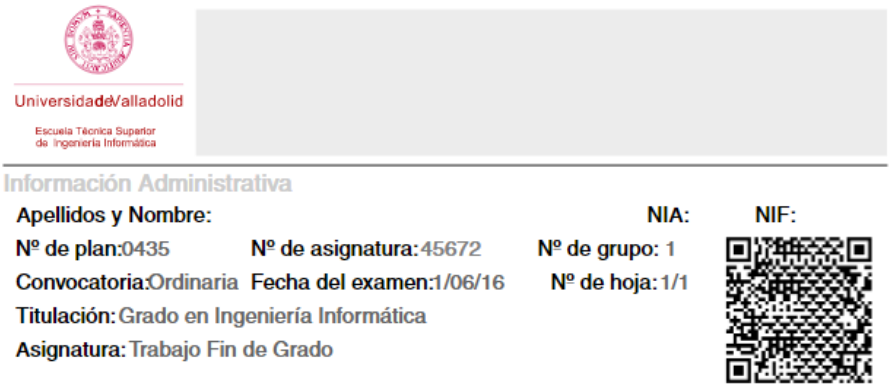

Figura 3.2: Zona superior del test

Para conseguir que los exámenes pudieran ser utilizados en un mayor número de situaciones se introdujeron dos modos distintos:

- **Oficial**. Está dirigido a exámenes con carácter oficial y cuyos resultados sean válidos en la evaluación del alumno. Permite especificar el código *NIA* de los alumnos a los que va dirigido el examen, y se creará una copia para cada uno de ellos. Una vez leído el test desde la aplicación móvil, la información del mismo se mandará al servidor para ser corregido y almacenado en el sistema. Además, en el QR no se incluye ningún tipo de información relativa a las respuestas del examen para evitar su copia por parte de los alumnos.

- **Autocorrección**. Está pensado para exámenes de entrenamiento que no tengan validez oficial y que puedan ser corregidos desde la propia aplicación móvil - sin necesidad del servidor - por los propios alumnos o el profesor. Añade las respuestas del test al código QR del examen para poder corregido de manera autónoma.

En cuanto a la tipografía, se pretendía conseguir un tipo de letra legible, agradable visualmente y compatible entre la mayor parte de equipos. Inicialmente se utilizó la fuente *Liberation Sans Narrow* para los textos de la información administrativa y *DIN* para la *zona de test*; tras numerosas pruebas en equipos con diferentes sistemas operativos y configuraciones, se notó que las fuentes no siempre eran reconocidas o no ofrecían exactamente el mismo resultado visual, por lo que se pasó a utilizar la fuente *Helvética* para todos los textos del examen.

3.2. Especificación de la *zona de test*

Esta es la sección más importante del examen de cara al algoritmo de detección. Está delimitada por cuatro recuadros y en su interior contiene la información de las preguntas, las respuestas y las versiones del test.

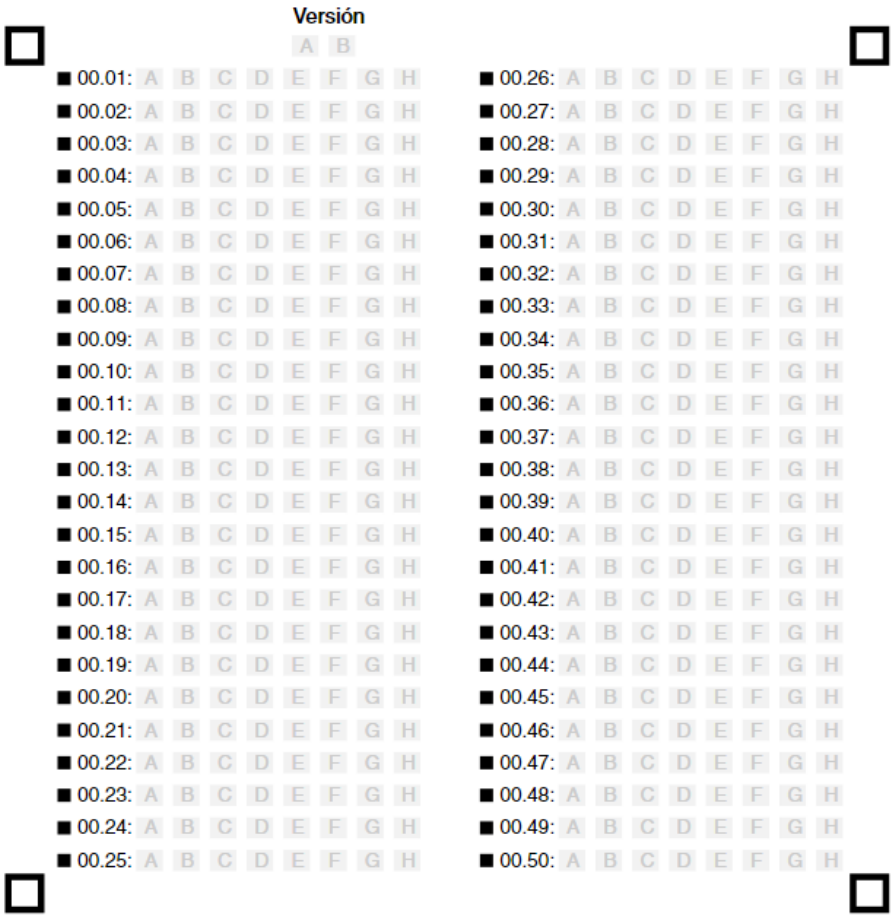

Figura 3.3: *Zona de test*

El diseño de los recuadros está inspirado por los cuadros de posición y alineamiento que se pueden encontrar en el código *QR*.

Figura 3.4: Ejemplo de código *QR*

La colocación de uno en cada esquina permite delimitar exactamente la zona de test mediante los bordes exteriores de los mismos y posibilita aplicar un algoritmo de corrección de perspectiva a la hora de escanear el examen con la cámara del dispositivo móvil -esta técnica se detallará en posteriores apartados-. El que no sean cuadrados completamente rellenos de color se debe a que el hacerlos huecos permite facilitar el reconocimiento de la zona de examen de manera inequívoca, acelerando el escaneo y evitando falsos positivos - al igual que el algoritmo anterior, se explicará más adelante -. Por lo tanto, todo lo que se encuentra dentro de la zona descrita será utilizado para la corrección del examen.

Para las medidas del documento en el fichero *SVG* - y su manejo tanto desde Inkscape como desde programas de edición de texto -, se ha utilizado el píxel como unidad, teniendo como referencia un documento *A4* de tamaño en píxeles de *744.09448 x 1052.3622*. Siguiendo esto, la zona de test y los recuadros tienen las siguientes medidas:

- **Esquina superior izquierda de la zona de test** en el punto 35, 342.
- **Ancho de la zona de test** de 673 píxeles.
- **Alto de la zona de test** de 675 píxeles.
- **Tamaño del recuadro exterior de cada esquina** de 30 x 30 píxeles.
- **Tamaño del recuadro interior de cada esquina** de 20 x 20 píxeles.

Tanto las versiones del test - en caso de que éste tenga más de una -, como las respuestas de cada pregunta quedan representadas por un rectángulo gris claro con una letra sobreimpresa en un tono gris algo más oscuro y centrada. Las medidas relacionadas son las siguientes:

- **El tamaño de cada rectángulo** es de 20 x 16 píxeles.
- **La distancia entre rectángulos** es de 8 píxeles. Ésta distancia se ha tomado teniendo en cuenta el tener una suficiente distancia entre rectángulos como para poder ser analizados sin problema y que además esta distancia no de unas opciones demasiado separadas.

Las preguntas se colocan en dos columnas de un máximo de 25 preguntas cada una. Si el test tiene menos de 26 preguntas utilizará únicamente la primera columna. La situación horizontal de cada columna es siempre la misma, estando la primera situada a 75 píxeles del límite izquierdo de la zona de test y la segunda a 75 píxeles de la mitad de la zona. Por tanto, el número de respuestas de cada pregunta y sus medidas se diseñaron teniendo en cuenta esta condición y el tamaño total de la zona de examen.

Para definir la altura de cada pregunta se utiliza un cuadrado negro de 10 x 10 píxeles. A continuación del mismo se encuentra una texto donde se indica el número de la pregunta y justo después comienzan las diferentes respuestas. La distancia entre cada pregunta en vertical es de 12 píxeles.

3.3. Especificación del contenido del código *QR*

Colocado justo encima de la esquina superior derecha de la zona de examen se encuentra un código QR. Éste se utiliza para almacenar la información necesaria para que el programa de reconocimiento tenga constancia de las características concretas del test. Los datos difieren según el tipo del test, y son los siguientes:

- **Comunes**. Ambos tipos de examen comparten varios campos que son necesarios para que sea reconocido por el software.
  - **Modo del test**. Indica el modo del examen. Puede tener los valores *Official* o *AutoCorrect*.
  - **Preguntas de la primera mitad**. Indica el número de preguntas de la primera mitad del test.
  - **Preguntas de la segunda mitad**. Indica el número de preguntas de la segunda mitad del test. Si tiene menos de 26 preguntas, este campo tendrá un valor *0*.
  - **Respuestas por pregunta**. Número de respuestas por pregunta.

    - **Número de versiones**. Indica el número de versiones que tiene el examen.

- **Oficial**. En este tipo se añaden datos necesarios para que el test sea enviado al servidor y pueda ser analizado y almacenado.

    - **Identificación única del test**. Es un ID generado por el servidor en el momento en el que se creó el examen e indica los valores principales del test en la base de datos.
    - **Dirección del servidor**. Indica la ruta donde se encuentra el servidor. Es la misma desde la que se accedió desde la aplicación de generación y visualización de exámenes.
    - ***NIA* del alumno**. Código del alumno al que está destinado el test. En este tipo cada examen está destinado a un único alumno.

- **Autocorrección**. Además de los datos comunes, en este tipo de añaden las respuestas correctas del examen para que pueda ser corregido de manera autónoma por la aplicación móvil.

El tamaño del QR en el SVG es de *135 x 135* píxeles, y su situación permite que pueda ser escaneado por la aplicación móvil mientras se está encuadrando la *zona de test*, además de salir perfectamente en la foto resguardo.

3.4. Detalles del uso de *Inkscape*

La utilización de *Inkscape* ha permitido, entre otras cosas mencionadas en puntos anteriores del documento, realizar un diseño iterativo y fluido de la plantilla de examen hasta el diseño final indicado en esta sección; pero no ha estado exento de ciertos problemas menores.

El problema principal que planteó el uso de este software, y que también hubiera ocurrido con la gran mayoría de editores de gráficos vectoriales, es que está orientado a una filosofía ***WYSIWYG***[3] - *What You See Is What You Get* -, que permite la creación de ficheros complejos mediante el uso de interfaces gráficas sencillas e intuitivas, pero que no generan siempre ficheros *SVG* bien formados ni estructuras internas *XML* sencillas. Esto provocó que en ciertos puntos del desarrollo, donde era necesario afinar las medidas y la jerarquía de todos los elementos para conseguir una generación y análisis de los tests rápida y robusta, fue imprescindible editar el fichero *SVG* en modo texto, tanto con el editor *XML* incluido en el propio *Inkscape* como con editores de texto como *NotePad++*[4].

La modificación fundamental que se realizó es la colocación de los elementos de la *zona de test* de manera jerárquica y el aprovechamiento del comando *SVG translate* para situar los elementos a distancias relativas de sus padres. Esto fue especialmente útil para la colocación de las preguntas, respuestas y versiones dentro de la *zona de test*.

[3] `https://en.wikipedia.org/wiki/WYSIWYG`
[4] `https://notepad-plus-plus.org/`

Capítulo 4

Planificación

La definición de tareas y la acotación del tiempo utilizado en cada una durante el desarrollo de un proyecto, es algo fundamental de cara a conseguir completarlo adecuadamente y en un tiempo razonable. Para abordar el trabajo se ha utilizado de manera ligera una estrategia de desarrollo ágil *SCRUM*[1].

4.1. Método *SCRUM*

Esta estrategia se basa en el desarrollo incremental, el solapamiento de las diferentes fases de desarrollo - en lugar de un esquema secuencial más tradicional -, y la continua iteración para conseguir abordar posibles cambios en el diseño durante el avance del trabajo y para tener una mayor comunicación con el cliente. Es adecuada principalmente para equipos con un número controlado de trabajadores, donde los requisitos sean desconocidos o inciertos y donde sea necesaria una continua prueba del software en desarrollo.

En este tipo de gestión existen varios roles, pudiéndose considerar como principales los siguientes:

- **Product owner**. Es la persona implicada en establecer un puente entre el usuario final, los responsables del negocio y el equipo de desarrollo.
- **Scrum master**. Es la persona encargada de apoyar al equipo de desarrollo y conseguir respetar las reglas del proceso *SCRUM*, para que se cumplan los objetivos.
- **Equipo de desarrollo**. Es el grupo de personas que trabaja en el proyecto.

En cuanto al propio trabajo que se realiza, la técnica se basa en el *Sprint*. Éstos son periodos de un tiempo definido - habitualmente entre una y cuatro semanas -, y definen un ciclo de desarrollo corto con una serie de características a implementar. Al inicio de cada *Sprint* se realiza una reunión de planificación donde se identifican los requisitos y las tareas a realizar para llevarlos a cabo. Cada día del *Sprint* se ejecuta una reunión de unos quince minutos llamada *Daily Scrum* donde se indica y valora lo realizado el día anterior y lo que se va a realizar en el actual. Además, al final de cada *Sprint* se hacen reuniones dónde se revisa y analiza el trabajo realizado y las características que finalmente se han conseguido implementar.

[1] `https://en.wikipedia.org/wiki/Scrum_(software_development)`

4.2. Aplicación al proyecto

Volviendo al proyecto que ocupa este documento, y teniendo en cuenta que ha sido realizado por una única persona, se puede percibir que no es posible realizar una diferenciación de roles al no existir un equipo. Por lo tanto se puede considerar que el alumno es la persona que ha realizado el proyecto - como conjunto de los diferentes roles que propone *SCRUM* -, y que el tutor es el cliente del mismo - a pesar de realizar tareas que pueden considerarse dentro de los roles -.

Como se indicará en el apartado dedicado a la evolución del proyecto, éste ha ido creciendo durante varios años de manera no continuada, pero el desarollo final ha transcurrido durante un año. Se han organizado *Sprints* de una duración habitual de cuatro semanas, teniendo al final de los mismos una reunión donde se ha evaluado el trabajo realizado, las dificultades encontradas y se ha organizado el trabajo a realizar durante el siguiente *Sprint*.

El tiempo estimado para la realización total del proyecto ha sido de unas 600 horas, estando dedicadas cerca de 50 a cada *Sprint*. En cuanto a la escritura de la memoria, ésta se realizó principalmente durante las etapas finales del trabajo, donde el tiempo de desarrollo se dedicaba principalmente a la implementación de las últimas características, la mejora de las ya existentes y la corrección de los últimos errores y detalles.

A modo de retrospectiva, cabe destacar los puntos que finalmente han requerido una mayor cantidad de tiempo de cara a mejorar el desarrollo y planificación de un proyecto similar en el futuro:

- **Algoritmo de reconocimiento y corrección**. Esta es la parte del desarrollo que ha requerido una mayor investigación, implementación y prueba hasta conseguir un balance adecuado de validez de los resultados, rendimiento en tiempo real y complejidad del algoritmo.
- **Configuración de los proyectos para el uso nativo de *OpenCV***. El uso de bibliotecas nativas ha requerido el uso de *JNI* y la configuración del *NDK* en el proyecto *Android*, ocupando una cantidad notable de tiempo de cara a conseguir un ciclo de desarrollo y prueba fluido.
- **Manejo de la plantilla *SVG* y conversión a *PDF***. A pesar de utilizar la biblioteca *Apache Batik* como intermediario para el manejo de las plantilla *SVG* y su conversión a formato *PDF*, el tener que modificar la plantilla desde el código, el manejo de la codificación de los caracteres y las fuentes ha requerido un elevado tiempo de desarrollo hasta conseguir un resultado adecuado y visualmente correcto.

Como apoyo a lo descrito se han utilizado programas de organización de tareas y notas como *Trello*[2] o *Google Keep*[3], sistemas de almacenamiento en la nube como *Dropbox*[4] y *Google Drive*[5], y *SourceTree*[6] junto a *Git*[7] como software de control de versiones.

[2] `https://trello.com/`
[3] `https:/keep.google.com`
[4] `https://www.dropbox.com/`
[5] `drive.google.com`
[6] `https://www.sourcetreeapp.com/`
[7] `https://git-scm.com/`

Parte I

Análisis del problema

Capítulo 5

Análisis y requisitos

Tras los capítulos introductorios, se detallan en esta parte los requisitos, actores, casos de uso y diagramas de secuencia identificados en el problema. Para ello se ha decidido utilizar el lenguaje de modelado *UML*[1] - *Unified Modeling Language* -, ya que provee el estándar y el tipo de diagramas necesarios.

5.1. Requisitos

En este primer apartado se especifican los requisitos que componen el problema y definirán el diseño de la solución del mismo. Durante las etapas iniciales del modelado y desarrollo del proyecto éstos se han ido modificando junto con los casos de uso, pero en este documento se incluyen antes para definir las bases de los capítulos posteriores.

5.1.1. Funcionales

1. El sistema permitirá generar exámenes de tipo test.
   - 1.1 Los exámenes podrán tener hasta ocho versiones.
   - 1.2 Los exámenes podrán tener hasta cincuenta respuestas.
   - 1.3 Los exámenes podrán tener hasta ocho respuestas por pregunta.
   - 1.4 Los exámenes incluirán una sección con la información administrativa.
     - 1.4.1 El test contendrá un campo para el nombre y apellidos del alumno.
     - 1.4.2 El test contendrá un campo para el código *NIA* del alumno.
     - 1.4.3 El test contendrá un campo para el código *NIF* del alumno.
     - 1.4.4 El test contendrá el número de identificación del plan de estudios.
     - 1.4.5 El test contendrá el número de identificación de la asignatura.
     - 1.4.6 El test contendrá el número de grupo al que va dirigido.
     - 1.4.7 El test contendrá el nombre de la convocatoria.
     - 1.4.8 El test contendrá la fecha de realización mismo.
     - 1.4.9 El test contendrá el nombre de la titulación correspondiente.
     - 1.4.10 El test contendrá el nombre de la asignatura correspondiente.
   - 1.5 Los exámenes incluirán una sección para indicar instrucciones para la realización del mismo.

[1] `http://www.uml.org/`

1.6 Los exámenes tendrán dos modos disponibles.

1.6.1 Dispondrán de un modo *Oficial*, que genera exámenes válidos para el profesor.

1.6.2 Dispondrán de un modo *Autocorrección*, que genera exámenes válidos para el profesor.

1.7 En modo *Oficial*, el sistema permitirá incluir los códigos *NIA* de todos los alumnos a los que irá dirigido el examen.

2. El sistema almacenará los tests *Oficiales* generados.

3. El sistema corregirá los exámenes de tipo *Oficial* de manera independiente al alumno.

4. El sistema permitirá corregir uno o todos los exámenes almacenados a petición del usuario.

5. El sistema permitirá visualizar los tests *Oficiales* almacenados.

5.1 El sistema permitirá filtrar los exámenes según su número de plan, su número de asignatura, año, convocatoria, identificación y número de alumno.

5.2 El sistema mostrará los datos de generación de cada test.

5.3 El sistema permitirá mostrar la foto resguardo de la corrección realizada desde el dispositivo móvil.

5.4 El sistema permitirá mostrar la foto de análisis de la corrección realizada desde el dispositivo móvil.

5.5 El sistema permitirá exportar la información de una o todas las correcciones.

5.6 El sistema mostrará la versión del test seleccionada por el alumno en cada corrección.

5.7 El sistema mostrará el porcentaje de respuestas acertadas de cada corrección.

5.8 El sistema mostrará las respuestas elegidas por el alumno en cada corrección.

6. El sistema dispondrá de una aplicación para dispositivos móviles, para el escaneo de exámenes.

7. El sistema permitirá corregir al momento los exámenes de tipo *Autocorrección*.

5.1.2. No funcionales

1. Los exámenes se exportarán en formato *SVG* y *PDF*.

2. Los exámenes se generarán por defecto en formato *A4*.

3. Los tests incluirán el logotipo de la escuela.

4. Las preguntas del test estarán numeradas del uno al cincuenta.

5. Las respuestas de cada pregunta estarán nombradas con letras mayúsculas de la *A* a la *H*.

6. Las versiones del test estarán nombradas con letras mayúsculas de la *A* a la *H*.

7. La zona de versiones, preguntas y respuestas del test quedará delimitada en sus esquinas por recuadros.

8. Los exámenes contendrán un código *QR*.

9. El sistema dispondrá de una aplicación para generar y visualizar los tests.

9.1 En modo *Oficial*, La aplicación permitirá añadir los código *NIA* de los alumnos en formato *CSV*.

9.2 La aplicación permitirá especificar la dirección del servidor y guardará dicha información en formato *JSON*.

9.3 Las correcciones de los exámenes se exportarán en formato *CSV*.

9.4 La aplicación mostrará las respuestas de cada corrección siguiendo un código de colores para facilitar su lectura.

10. Las aplicaciones de generación y visualización y el servidor deberán poder ejecutarse en *Windows*, *MacOSX* y *Linux*.
11. El sistema dispondrá de un servidor para el almacenado y corrección de tests *oficiales*.
12. La aplicación servidor dispondrá de métodos *REST* para el almacenado, corrección y visualización de exámenes.
13. La aplicación para dispositivos móviles funcionará en dispositivos con sistema operativo *Android 5.0* hacia adelante.
14. La aplicación para dispositivos móviles dispondrá de flash si el hardware lo permite.
15. La aplicación para dispositivos móviles podrá el escaneado y la corrección del test en tiempo real.
16. En exámenes de modo *Autocorrección*, la aplicación para dispositivos móviles mostrará los resultados de la corrección en una lista.

5.2. Actores

El sistema está diseñado principalmente para el ámbito académico, por lo que aunque cualquier potencial usuario podría instalarlo, desplegarlo y utilizarlo, el prototipo de actores que utilizarían el sistema se puede clasificar en dos tipos:

- **Profesor**. Es el usuario que diseña y genera los exámenes y, en caso de hacerlo en modo *Oficial*, el que corregirá los tests y analizará el resultado de los mismos.
- **Alumno**. Es el usuario que recibirá, realizará y podría escanear los exámenes desde su dispositivo móvil.

5.3. Casos de uso

Tras la definición de requisitos, se indican los diferentes casos de uso generados. En primer lugar se muestra el diagrama general de casos de uso:

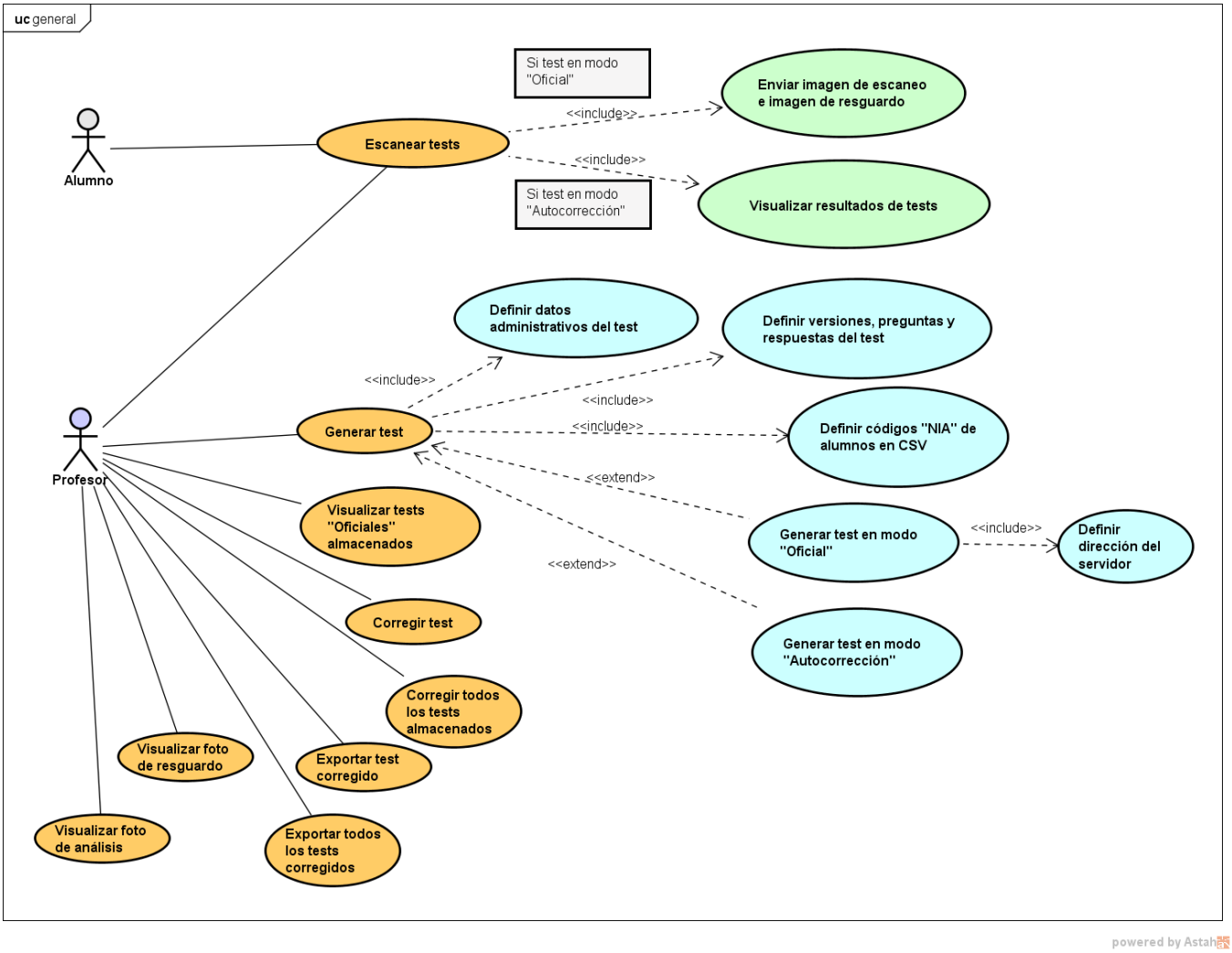

Figura 5.1: Diagrama de casos de uso general detallado

Para describir los principales casos de uso expuestos en el diagrama general se va a utilizar el siguiente diseño en formato de tabla:

Título	**Título del caso de uso.**
Descripción	Descripción del caso de uso.
Actores	Actores involucrados en el caso de uso.
Precondiciones	Precondiciones para la realización del caso de uso.
Proceso Descripción secuencial del proceso seguido por el caso de uso.	
Excepciones Descripción de estados excepcionales durante el proceso	

Tabla 5.1: Tabla general de casos de uso

5.3.1. Generar test

Título	Generar test.
Descripción	El profesor diseña y genera un test.
Actores	Profesor.
Precondiciones	Ninguna.

Proceso

1. El usuario introduce los datos administrativos del test.
2. El usuario define las versiones, preguntas y respuestas del test.
3. El usuario elige el modo del test.
   - En modo *Oficial* debe introducir los códigos *NIA* de los alumnos en *CSV*. Además, puede definir la dirección del servidor.
   - En modo *Autocorrección* no debe realizar ninguna configuración añadida.
4. El usuario pulsa el botón de generación.
   - En modo *Oficial*, la aplicación contactará con el servidor para recibir un número de identificación para el test generado.
5. El usuario elige donde en que directorio guardará el test generado.
   - En modo *Oficial* se guardará una copia del test para cada alumno.

Excepciones

- Si el usuario introduce algún dato no válido, el programa se lo notificará.
- Si el test a generar es de modo *Oficial* y el programa no puede contactar con el servidor o recibe una respuesta errónea, el test no generará y le será notificado al usuario.

Tabla 5.2: Caso de uso: Generar test

5.3.2. Visualizar tests *Oficiales* almacenados

Título	**Visualizar tests *Oficiales* almacenados.**
Descripción	El profesor comprueba los tests *Oficiales* del servidor.
Actores	Profesor.
Precondiciones	Haber generado y escaneado al menos un test.

Proceso

1. El usuario pide lo datos de los tests al servidor.
2. El usuario filtra los datos de los test para seleccionar el que desee.
3. El usuario comprueba los datos mostrados sobre el test.
   - Se muestra en pantalla los datos de generación del test.
   - Dependiendo del estado de corrección del test, se muestran más datos:
     - Si no está corregido, se muestra un botón que permite pedir la corrección al servidor -indicado en el siguiente caso de uso-.
     - Si está corregido, se muestran además los datos de la corrección.

Excepciones

- Si no existen tests almacenados en el servidor, el usuario será notificado.
- Si la aplicación no puede contactar con el servidor o recibe una respuesta errónea, no se mostrarán los datos de los tests y el usuario será notificado.

Tabla 5.3: Caso de uso: Visualizar tests *Oficiales* almacenados

5.3.3. Corregir test

Título	Corregir test.
Descripción	El profesor pide al servidor la corrección de un test concreto.
Actores	Profesor.
Precondiciones	▪ Haber generado y escaneado al menos un test. ▪ Existir al menos un test sin corregir en el servidor. ▪ Haber realizado el caso de uso *Visualizar test Oficiales almacenados en el servidor*.
Proceso	1. El usuario selecciona uno de los tests no corregidos. 2. El usuario pide la corrección al servidor. 3. El servidor corrige el examen y lo envía de vuelta al programa. 4. El usuario es notificado con los datos recibidos.
Excepciones	▪ Si la aplicación no puede contactar con el servidor o recibe una respuesta errónea, no se podrá corregir el test y el usuario será notificado.

Tabla 5.4: Caso de uso: Corregir test

5.3.4. Corregir todos los tests almacenados

Título	Corregir todos los tests almacenados.
Descripción	El profesor pide la corrección de todos los tests.
Actores	Profesor.
Precondiciones	▪ Haber generado y escaneado al menos un test. ▪ Existir al menos un test sin corregir en el servidor.
Proceso 1. El usuario pulsa el botón para pedir la corrección de todos los tests almacenados. 2. El servidor realiza la operación y notifica al usuario. 3. Si desea ver las correcciones, el usuario realizará el caso de uso *Visualizar test Oficiales almacenados en el servidor*.	
Excepciones ▪ Si el usuario pide la corrección y todos los tests están ya corregidos, será notificado. ▪ Si la aplicación no puede contactar con el servidor o recibe una respuesta errónea, no se podrán corregir los tests y el usuario será notificado.	

Tabla 5.5: Caso de uso: Corregir todos los tests almacenados

5.3.5. Exportar test corregido

Título	**Exportar test corregido.**
Descripción	El profesor exporta los datos de una corrección concreta.
Actores	Profesor.
Precondiciones	▪ Haber generado y escaneado al menos un test. ▪ Existir al menos un test corregido en el servidor. ▪ Haber realizado el caso de uso *Visualizar test Oficiales almacenados en el servidor*.
Proceso	1. El usuario selecciona uno de los tests corregidos. 2. El usuario pulsa el botón para exportar los datos de al corrección a un fichero *CSV*. 3. El usuario selecciona el fichero donde guardará los datos.
Excepciones	▪ Si la aplicación no puede exportar la corrección en formato *CSV*, el usuario será notificado.

Tabla 5.6: Caso de uso: Exportar test corregido

5.3.6. Exportar todos los tests corregidos

Título	**Exportar todos los tests corregidos.**
Descripción	El profesor exporta todas las correcciones.
Actores	Profesor.
Precondiciones	▪ Haber generado y escaneado al menos un test. ▪ Existir al menos un test corregido en el servidor. ▪ Haber realizado el caso de uso *Visualizar test Oficiales almacenados en el servidor*.
Proceso	1. El usuario pulsa el botón que exporta todos los tests corregidos. 2. El usuario selecciona el fichero donde guardará los datos.
Excepciones	▪ Si la aplicación no puede exportar la corrección en formato *CSV*, el usuario será notificado.

Tabla 5.7: Caso de uso: Exportar todos los tests corregidos

5.3.7. Visualizar foto de resguardo

Título	Visualizar foto de resguardo.
Descripción	El profesor pide la visualización de la foto de resguardo.
Actores	Profesor.
Precondiciones	▪ Haber generado y escaneado al menos un test. ▪ Existir al menos un test corregido en el servidor. ▪ Haber realizado el caso de uso *Visualizar test Oficiales almacenados en el servidor*.
Proceso	1. El usuario selecciona uno de los tests corregidos. 2. El usuario pulsa el botón que pide la visualización de la foto de resguardo. 3. El servidor recoge la foto de resguardo del repositorio y la envía a la aplicación. 4. La foto se guarda en al carpeta de ejecución de la aplicación. 5. La foto se muestra al usuario mediante el programa de visualización de imágenes por defecto del sistema operativo.
Excepciones	▪ Si la aplicación no puede contactar con el servidor o recibe una respuesta errónea, no se podrán mostrar la imagen y el usuario será notificado.

Tabla 5.8: Caso de uso: Visualizar foto de resguardo

5.3.8. Visualizar foto de análisis

Título	Visualizar foto de análisis.
Descripción	El profesor pide la visualización de la foto de análisis.
Actores	Profesor.
Precondiciones	▪ Haber generado y escaneado al menos un test. ▪ Existir al menos un test corregido en el servidor. ▪ Haber realizado el caso de uso *Visualizar test Oficiales almacenados en el servidor*.
Proceso	1. El usuario selecciona uno de los tests corregidos. 2. El usuario pulsa el botón que pide la visualización de la foto de análisis. 3. El servidor recoge la foto de resguardo del repositorio y la envía a la aplicación. 4. La foto se guarda en al carpeta de ejecución de la aplicación. 5. La foto se muestra al usuario mediante el programa de visualización de imágenes por defecto del sistema operativo.
Excepciones	▪ Si la aplicación no puede contactar con el servidor o recibe una respuesta errónea, no se podrán mostrar la imagen y el usuario será notificado.

Tabla 5.9: Caso de uso: Visualizar foto de análisis

5.3.9. Escanear tests

Título	Escanear tests.
Descripción	El usuario escanea un test desde la aplicación móvil.
Actores	Profesor o Alumno.
Precondiciones	Haber realizado al menos una vez el caso de uso *Generar test*.

Proceso

1. El usuario comienza el escaneo de uno de los tests, encuadrándolo como indica el programa.
   - Si ya había escaneado uno antes, debe pulsar el botón para realizar un nuevo escaneo.
   - Si desea utilizar el *Flash* -y su dispositivo dispone de dicha capacidad-, pulsará el botón de *Flash*.
2. El programa analiza la imagen hasta leer correctamente el QR y la *zona de test*.
3. El programa termina de escanear el test. Dependiendo de su modo, pueden mostrarse dos comportamientos distintos:
   - Si es de modo *Oficial*, la aplicación contactará con el servidor y enviará los datos escaneados. Cuando el servidor reciba los datos, el usuario será notificado.
   - Si es de modo *Autocorrección*, la aplicación corregirá el test y mostrará los resultados en forma de tabla al usuario.

Excepciones

- Si la lectura de los datos del código *QR* es errónea, el usuario será notificado y se deberá encuadrar de nuevo el test.
- Si la lectura de los datos de la *zona de test* es errónea, el usuario será notificado y se deberá encuadrar de nuevo el test.
- En caso de un test en modo *Oficial*, si la aplicación no puede contactar con el servidor o recibe una respuesta errónea, los datos del escaneo no se habrán almacenado en el servidor y el usuario será notificado.

Tabla 5.10: Caso de uso: Escanear tests

Capítulo 6

Diagrama de clases de análisis

Desde una perspectiva de alto nivel y teniendo en cuenta los actores y los casos de uso, a continuación se muestra un diagrama de clases general, que permite visualizar la relación entre los diferentes elementos principales del problema:

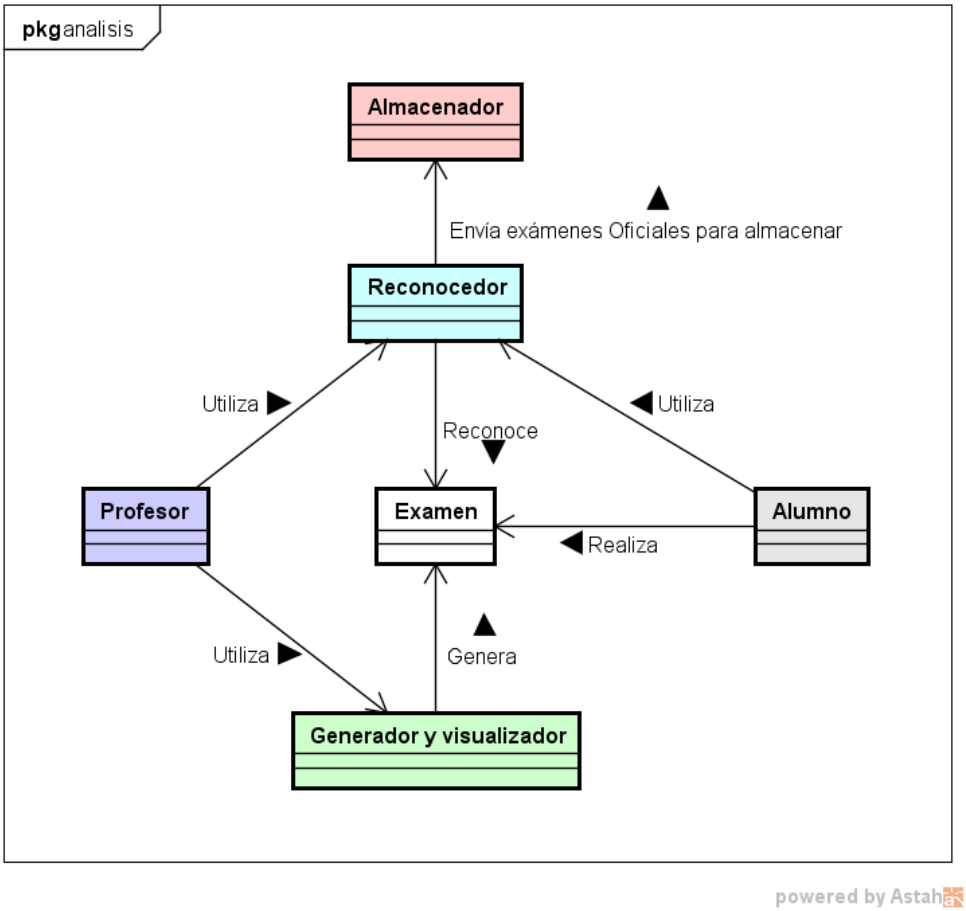

Figura 6.1: Diagrama de clases de análisis

Las clases mostradas están relacionadas con los componentes descritos a lo largo de los capítulos anteriores y tienen las siguientes características:

Profesor Es el usuario que diseña y genera los exámenes; además, se encargará de analizar sus resultados si éstos son de tipo *Oficial*.

Alumno Es el usuario que recibe y realiza los exámenes generados por el profesor. Si el test es de tipo *Autocorrección*, podrá corregirlo al momento utilizando el *Reconocedor*.

Examen Es el componente principal del proyecto y alrededor del cual funcionan el resto de elementos. Es generado por el *Profesor* mediante el *Generador*, completado por el *Alumno* y reconocido por el *Reconocedor*. Si es de tipo *Oficial*, el *Reconocedor* lo enviará al *Almacenador* para que los guarde.

Generador y visualizador Es el componente encargado de generar los *Exámenes*.

Reconocedor Se encarga de reconocer los *Exámenes*, y corregirlos si son de tipo *Autocorrección*.

Almacenador Es responsable de almacenar los *Exámenes Oficiales*.

Parte II

Diseño de la solución propuesta

Capítulo 7

Planteamiento general

Este capítulo del documento está dedicado a otorgar una visión formal del sistema como conjunto. Dejando de lado momentáneamente el lenguaje *UML*, se muestra a continuación un esquema general del sistema:

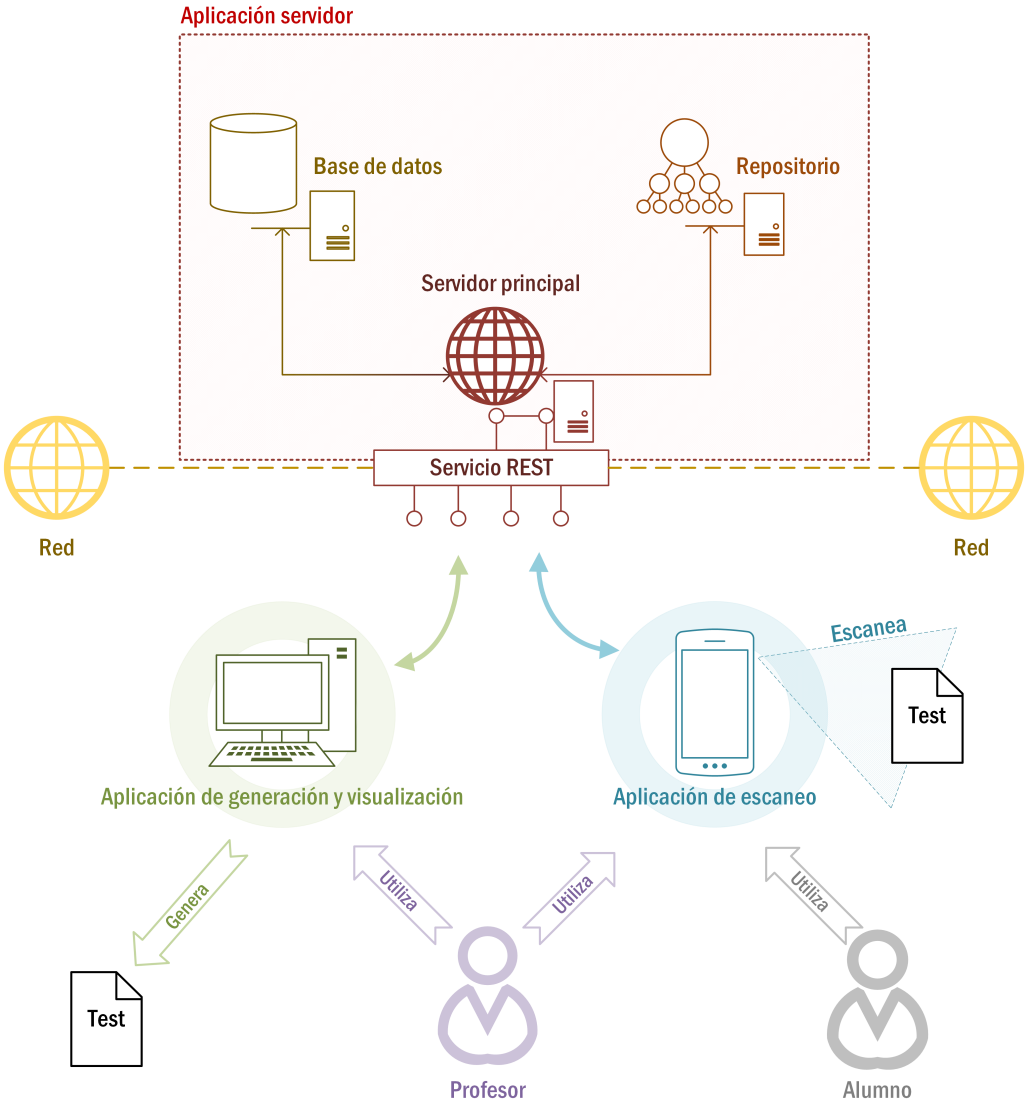

Figura 7.1: Esquema general del sistema

El sistema sigue una implementación ligera de una arquitectura en tres capas[1]. La definición formal de ésta separa la lógica de diseño de la lógica de negocio mediante tres capas:

- **Capa de presentación**. Es la capa que presenta la información al usuario y recoge su información - se podría considerar como la interfaz gráfica -. Esta capa se comunica con la de negocio.
- **Capa de negocio**. Es la capa que incluye la lógica real de la aplicación. Se comunica con la capa de presentación y con la de datos. Por una parte envía a la capa de presentación la información que debe se presentada al usuario y recibe la información introducida por el mismo a través de la capa de presentación; y por otra parte se comunica con la capa de datos para guardar y recoger información.
- **Capa de datos**. Es la capa que contiene los datos del sistema y la que se encarga de acceder a ellos, ya sea para guardarlos o para recogerlos y enviarlos a la capa de datos.

Además, define el término *nivel* para referirse a la separación física que tienen las diferentes capas, pudiendo existir desde un único *nivel* - todo el sistema incluido en un único hardware -, hasta todos los que se consideren necesarios.

Siguiendo lo mostrado en el diagrama, el sistema adapta la división expuesta de la siguiente manera:

- **Capa de presentación**. Corresponde a la aplicación de generación y visualización y la aplicación de escaneo. En este punto se puede notar porque se ha referido a la adopción de la arquitectura de tres capas como *ligera*, ya que se podría considerar que existe una capa de presentación diferente para cada una de las dos aplicaciones y que, además, dichas aplicaciones contienen componentes o secciones relativas a la *capa de negocios*. Se han considerado ambas aplicaciones como una única capa de presentación por su relación común con el servidor principal - *capa de negocios* - y porque en la visión general del sistema ambas corresponden al conjunto que tiene relación directa con el usuario. No obstante, en los capítulos posteriores dedicados a cada aplicación porr separado se realiza un análisis de la arquitectura utilizada independientemente en cada una de ellas.
- **Capa de negocio**. Corresponde a la aplicación del servidor principal. Utiliza un servicio *REST* para comunicarse con la capa de presentación y unos componentes propios y bibliotecas de terceros para conectarse con la capa de datos.
- **Capa de datos**. Corresponde al servidor de base de datos y el de repositorio. A este nivel no se considera la diferencia de la versión del sistema con servidores externos o embebidos ya que en ambos casos existe una capa de datos separada, con unos servicios propios, ya sean otorgados por servidores físicamente separados y distintos o por bibliotecas internas que realizan la misma función.

Debido a las múltiples aplicaciones que componen el sistema y a la posibilidad de utilizar servidores externos o embebidos, la separación en *niveles* contiene varios detalles:

- El que podría considerarse cono *nivel de presentación*, se compone de dos sub-niveles diferentes. Por una parte el relativo a la aplicación de generación y visualización, que típicamente será un equipo de escritorio - pero puede ser cualquier sistema compatible con *Java 8* -, y por otra un dispositivos móvil con sistema operativo *Android* en versión *5.0* o superior para la aplicación de escaneo.
- En cuanto a los niveles relativos a la *capa de negocio* y la *capa de datos*, se pueden observar varias posibilidades:

[1] `https://en.wikipedia.org/wiki/Multitier_architecture`

- En caso de utilizar servidores externos, podrían darse uno o dos *niveles*. Si las aplicaciones de todos los servidores - principal, base de datos y repositorio -, están incluidos dentro del mismo equipo servidor, la *capa de negocio* y la *capa de datos* se encontrarían en un mismo nivel. En cambio si cada aplicación servidor se sitúa en máquinas diferentes, se podrían indicar de uno a tres niveles pudiendo contener entre ellos la *capa de negocio* y la *capa de datos* de manera combinada o individual.
- Si se utilizan los servidores embebidos dentro del principal - que es la opción más directa para el despliegue del proyecto -, la *capa de negocio* y la *capa de datos* se encontrarían en un único equipo servidor, por lo que se encontrarían en un mismo *nivel*.

Volviendo al lenguaje *UML* y para facilitar la compresión del sistema completo y los tres sub-sistemas que lo forman, se incluye a continuación un diagrama general de como interactuarían los profesores y los alumnos.

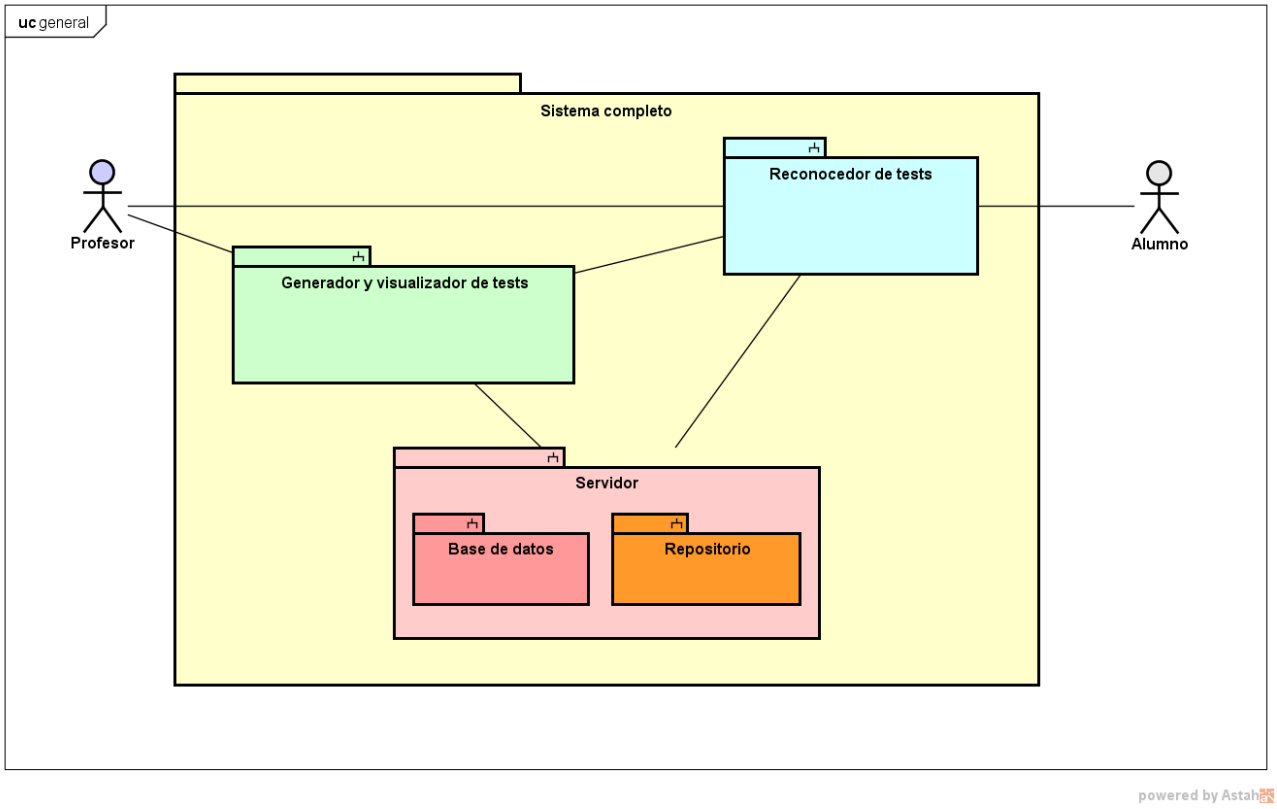

Figura 7.2: Diagrama de alto nivel

Por tanto, el profesor interactuará de manera directa o indirecta con todas las aplicaciones y el alumno únicamente utilizará el software para dispositivos móviles. De manera más detallada, los elementos mostrados son los siguientes:

Profesor Es el usuario que diseña y genera los exámenes; además, se encargará de analizar sus resultados si éstos son de tipo *Oficial*.

Alumno Es el usuario que recibe y realiza los exámenes generados por el profesor. Si el test es de tipo *Autocorrección*, podrá corregirlo al momento utilizando el *Reconocedor de tests*.

Generador y visualizador de tests Es el software que permite generar los exámenes y visualizar los *Oficiales* ya escaneados por el *Reconocedor de tests*.

Reconocedor de tests Permite escanear y reconocer los tests. Si son de tipo *Autocorrección*, los corregirá y mostrará los resultados al momento; en cambio si son *Oficiales* los enviará al *Servidor*, donde serán corregidos y almacenados.

Servidor Es la aplicación encargada de corregir y almacenar los tests *Oficiales*. En su interior contiene además una *Base de datos* y un *Repositorio*.

Base de datos Es el componente encargado de almacenar la información de los exámenes y sus correcciones.

Repositorio Es el componente encargado de almacenar las imágenes utilizadas para realizar el reconocimiento y corrección de los exámenes.

7.1. Diagramas de secuencia

Partiendo de los caso de uso explicados, en este apartado se muestran sus diagramas de secuencia para un flujo de ejecución correcto. Los esquemas, aunque siguen en parte los métodos que serán utilizados dentro del sistema, se muestran simplificados para dar una mayor legibilidad a los diagramas.

Antes de mostrar los diagramas, cabe explicar los objetos que se van a representar en ellos.

- **Actores**. Representa los actores del sistema: *Profesor* y *Alumno*.
- **Aplicación de generación y visualización**. Representa el software de generación y visualización de exámenes como un objeto completo.
- **Test generado**. Representa un test generado desde la aplicación. En este caso se realiza un símil con una de las clases que componen la aplicación -esto se desarrollará en posteriores apartados-, pero se incluye aquí como excepción debido a que los tests es uno de los componentes generales del sistema y tener constancia de ellos en el diagrama facilita la comprensión del flujo.
- **Servidor**. Representa la aplicación servidor que gestiona las peticiones *REST* y las conexiones con la base de datos y el repositorio.
- **Corrector de tests**. De manera similar al objeto *Test generado*, éste es una clase incluida dentro del código del proyecto, pero se presenta como un objeto más dado que el algoritmo de corrección utilizando *OpenCV* es un componente fundamental del proyecto. A pesar de existir una implementación en el servidor - en lenguaje *Java* -, y otra en la aplicación móvil - en lenguaje *C* -, el funcionamiento interno del algoritmo es el mismo y se considera aquí como el mismo objeto.
- **Base de datos**. Representa la base de datos donde se almacenan los tests. Este componente podría incluirse dentro del objeto servidor -con más razón si se utiliza la versión del sistema con los servidores embebidos-, pero se muestra separado para otorgar la noción de que, en el fondo, son componentes independientes que son gestionados por el servidor principal.
- **Repositorio**. Representa el repositorio de datos donde se almacenan las imágenes de los escaneos. Este componente podría incluirse dentro del objeto servidor -con más razón si se utiliza la versión del sistema con los servidores embebidos-, pero se muestra separado para otorgar la noción que, en el fondo, son componentes independientes que son gestionados por el servidor principal.
- **Aplicación de escaneo**. Representa la aplicación para dispositivos móviles encargada de escanear los tests. Es la única aplicación con la que se comunica el actor *Alumno*.

Dada la extensión en horizontal de las imágenes de los diagramas, para facilitar su lectura éstos se van a mostrar apaisados.

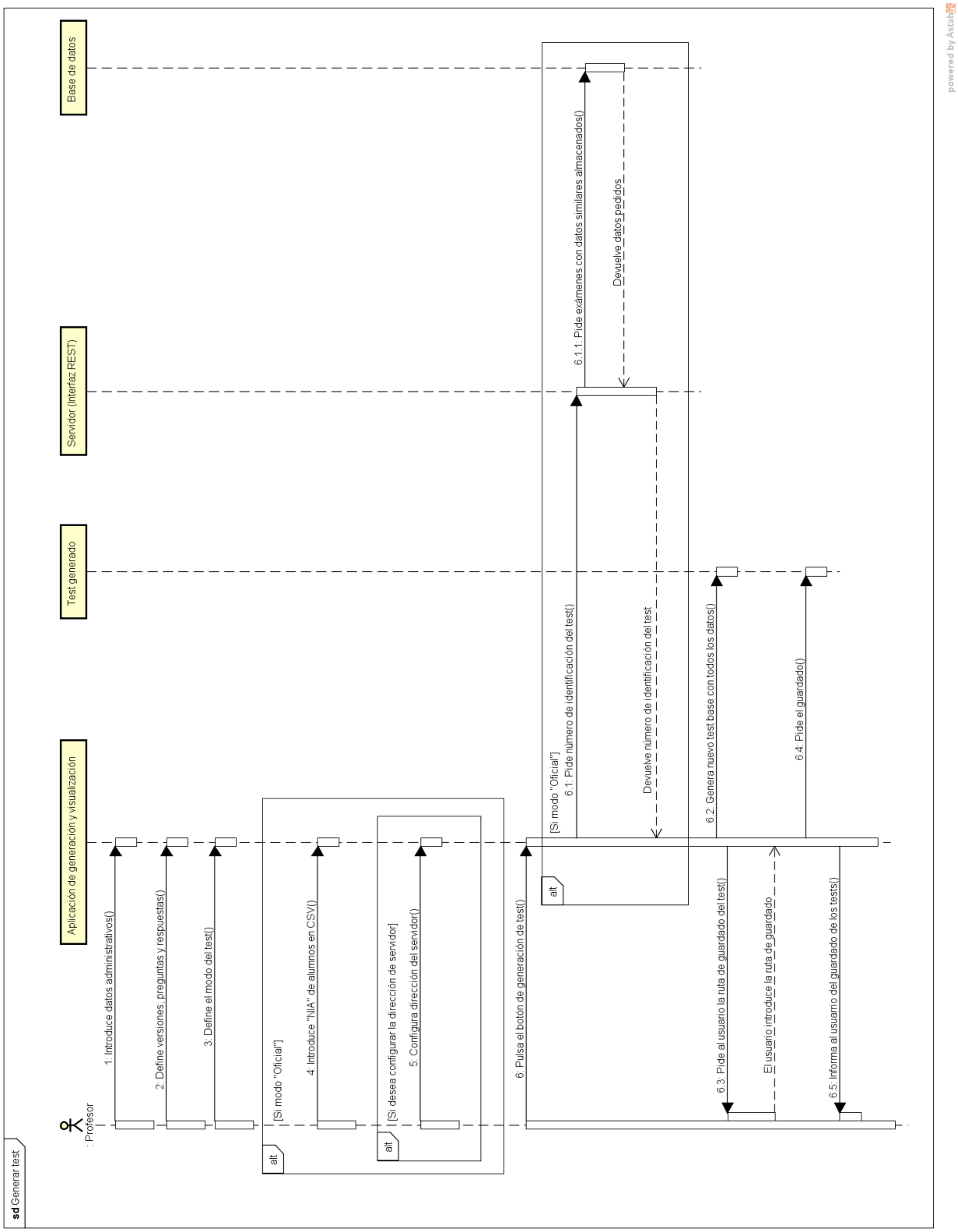

Figura 7.3: Diagrama de secuencia - Generar test

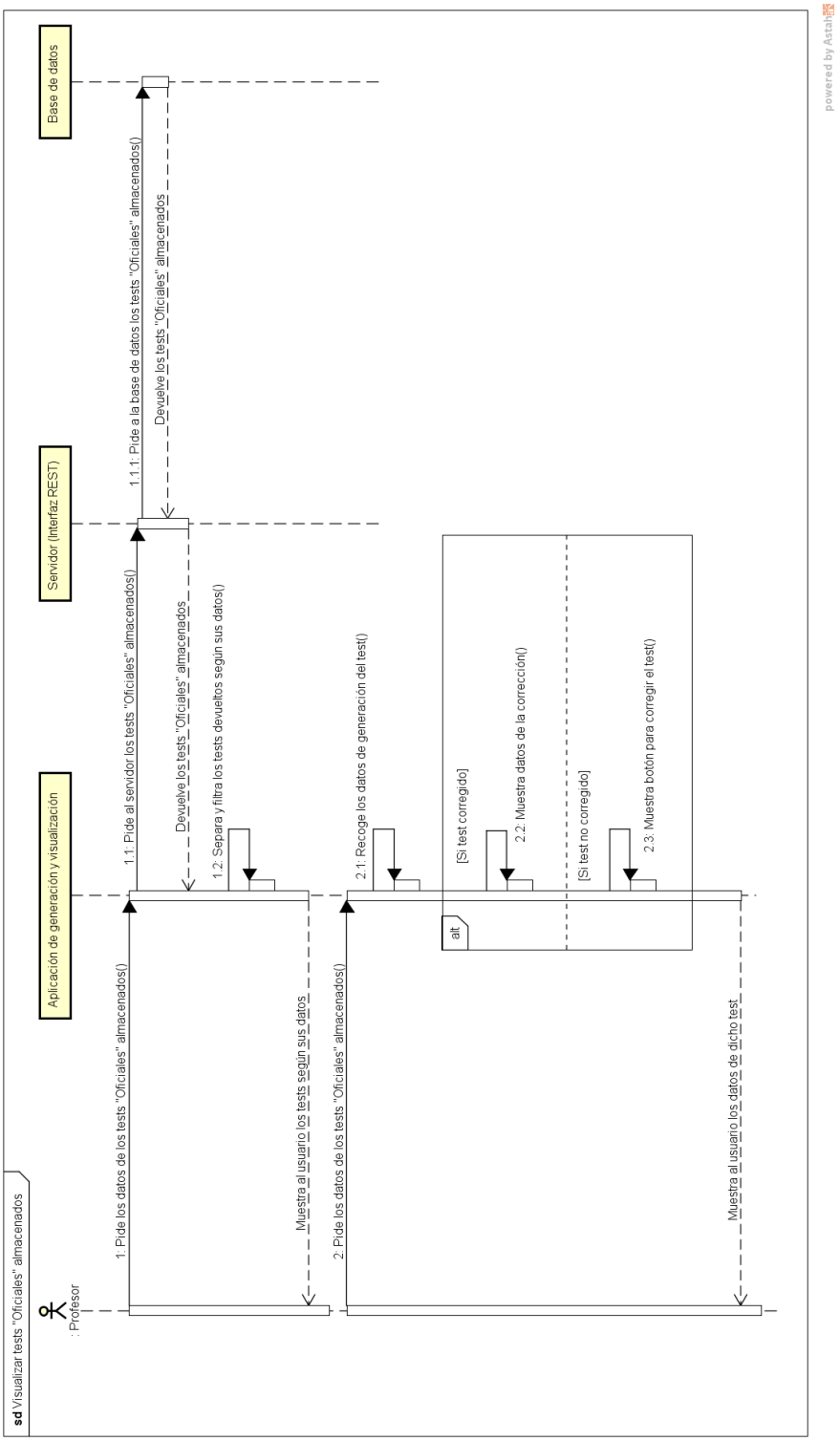

Figura 7.4: Diagrama de secuencia - Visualizar tests *Oficiales* almacenados

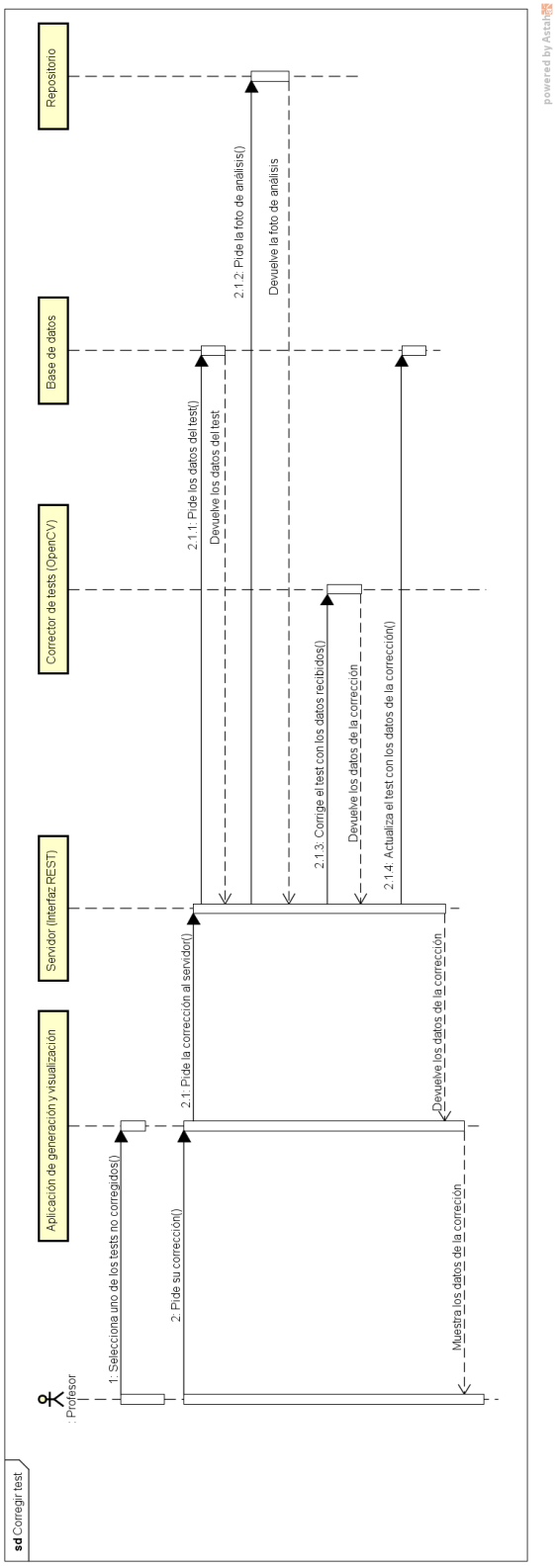

Figura 7.5: Diagrama de secuencia - Corregir test

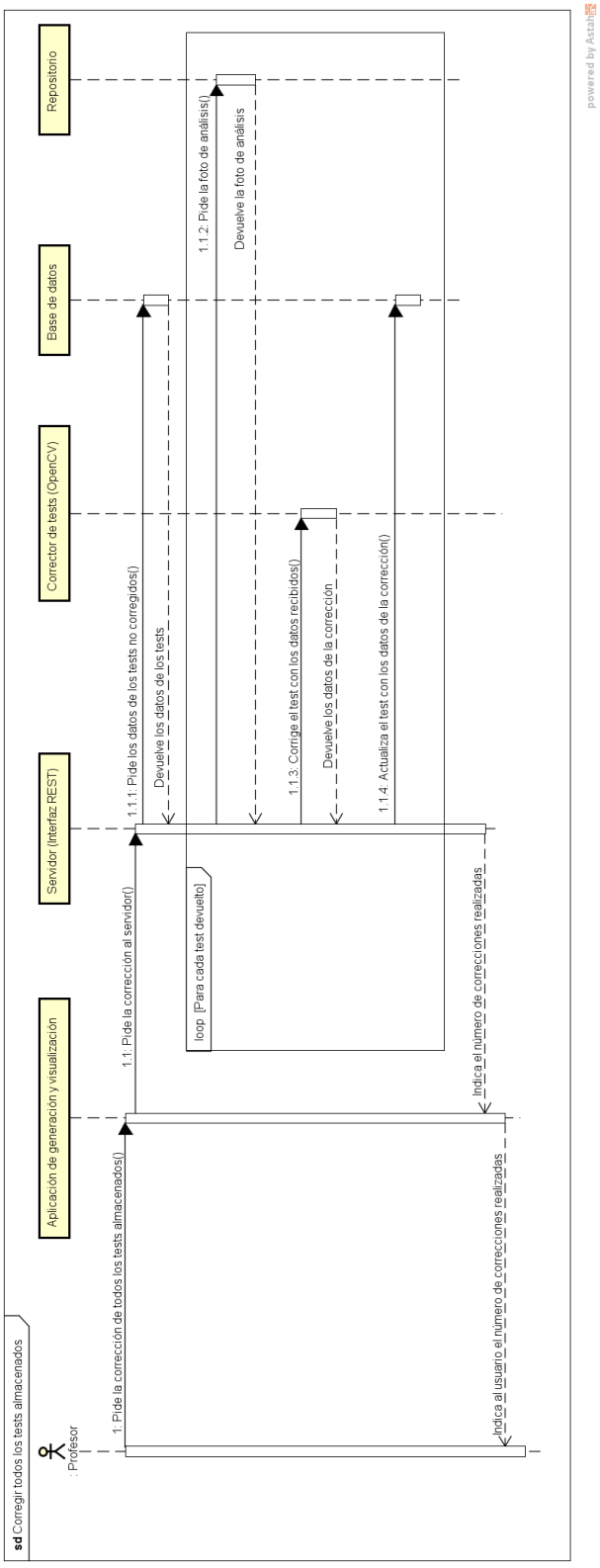

Figura 7.6: Diagrama de secuencia - Corregir todos los tests almacenados

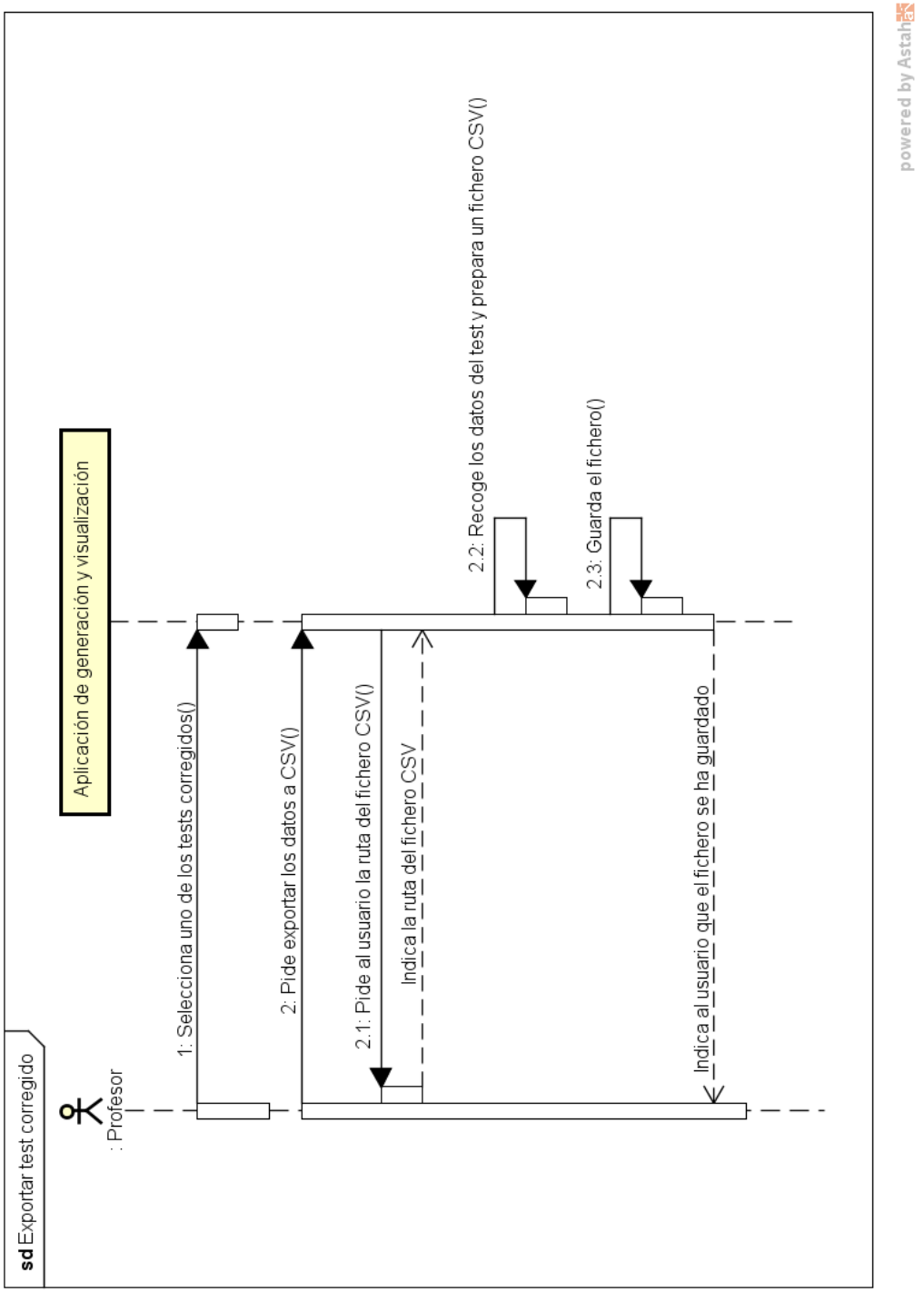

Figura 7.7: Diagrama de secuencia - Exportar test corregido

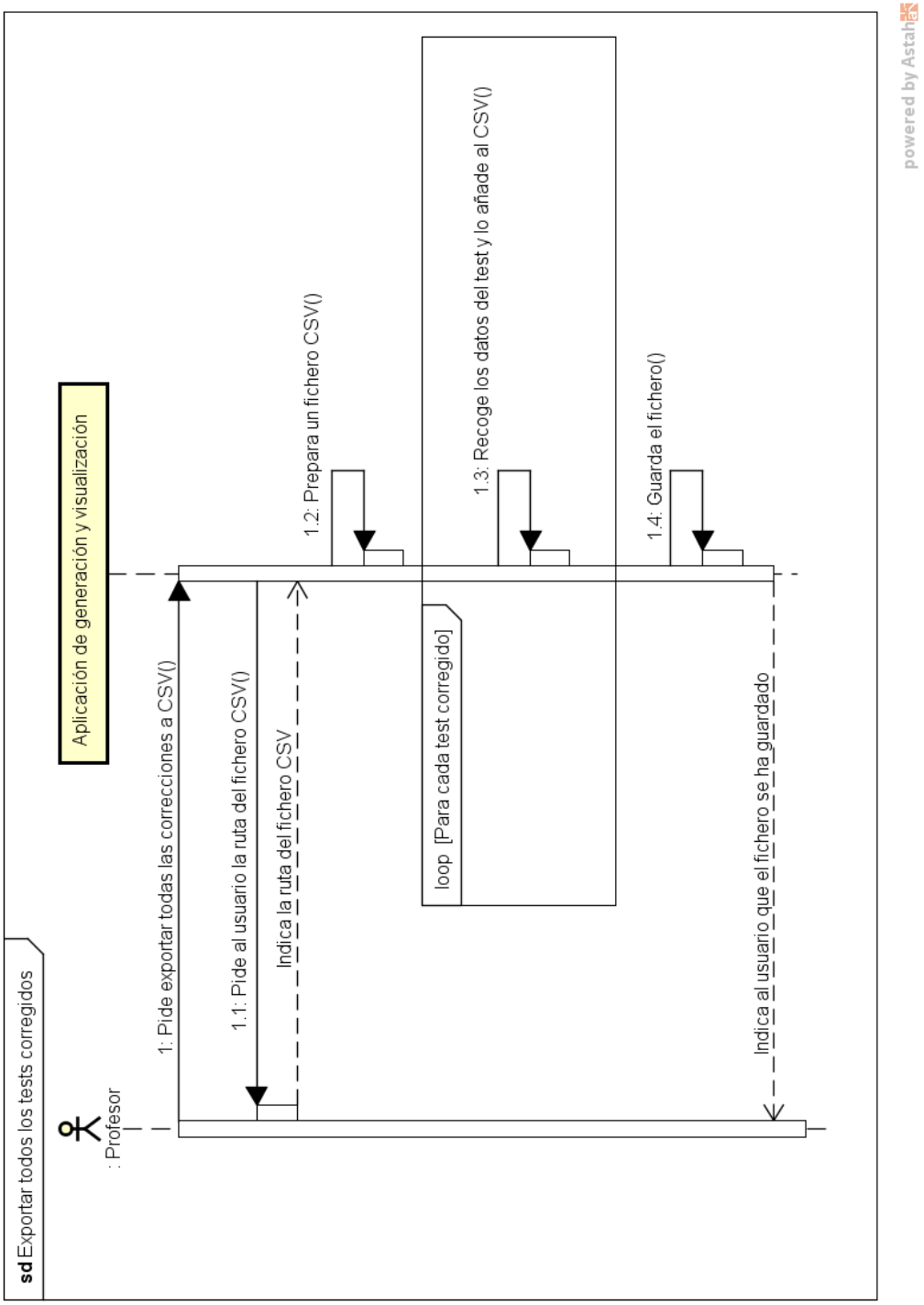

Figura 7.8: Diagrama de secuencia - Exportar todos los tests corregidos

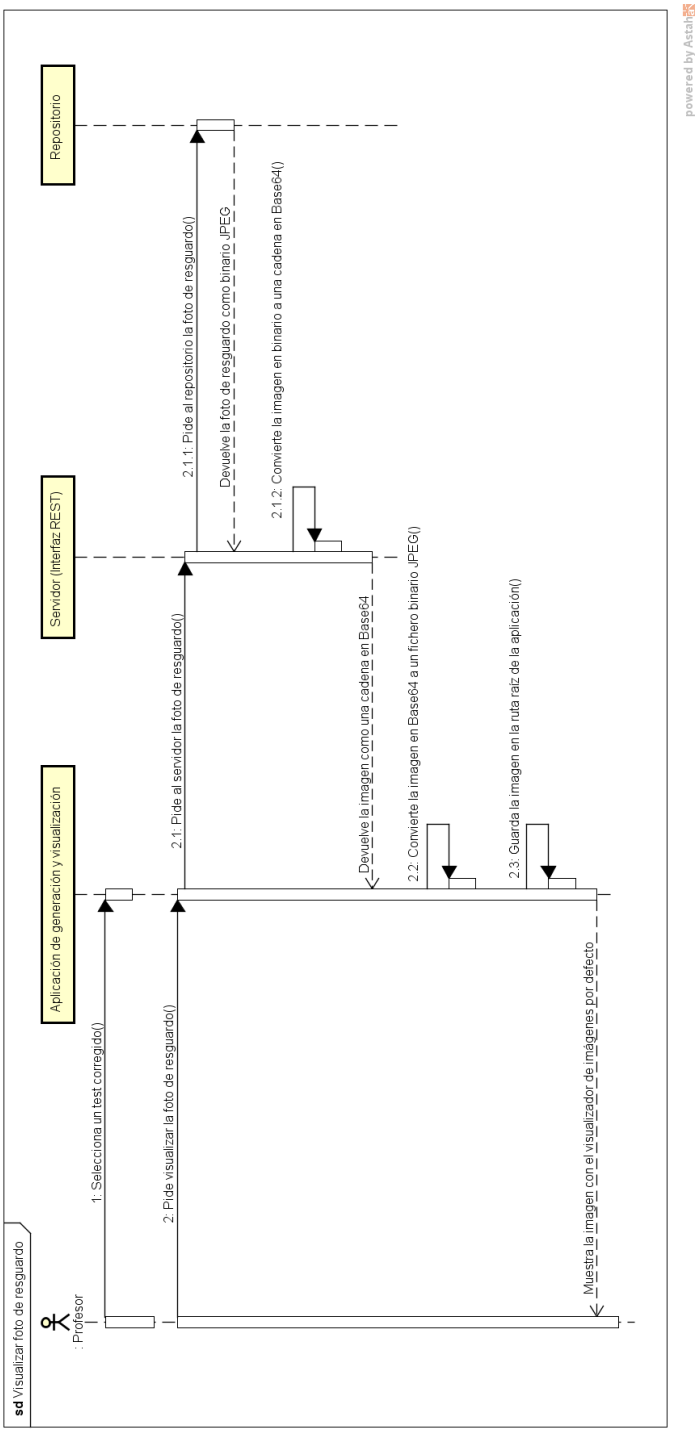

Figura 7.9: Diagrama de secuencia - Visualizar foto de resguardo

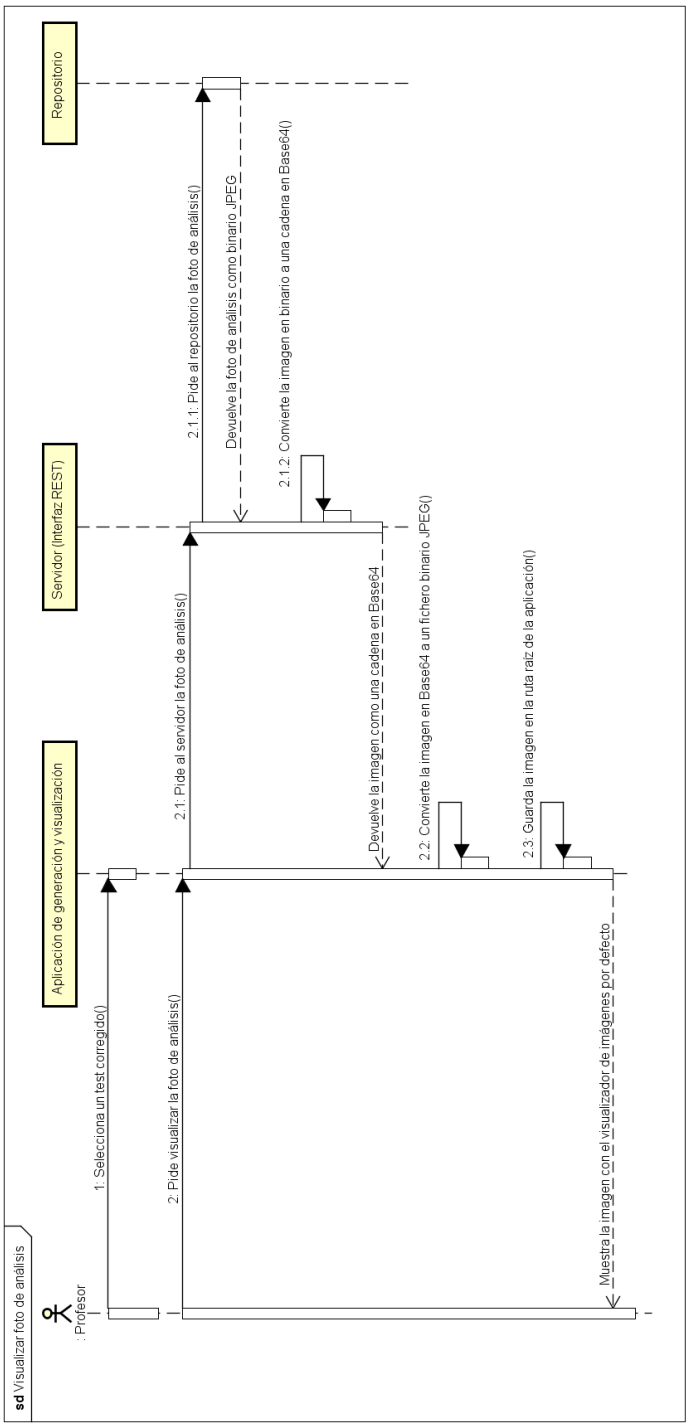

Figura 7.10: Diagrama de secuencia - Visualizar foto de análisis

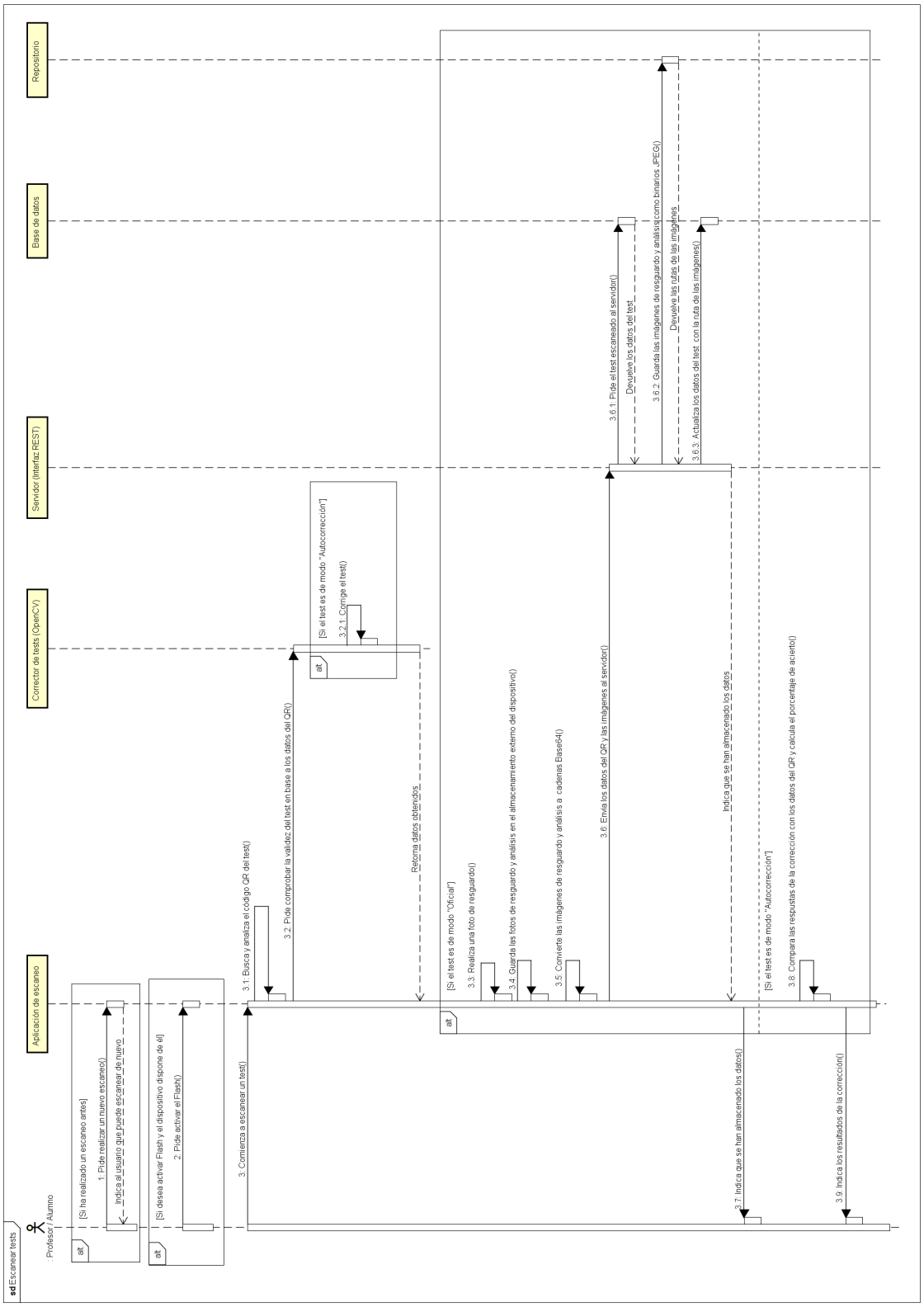

Figura 7.11: Diagrama de secuencia - Escanear tests

7.2. Diagrama de despliegue

Este diagrama se asemeja bastante con el enseñado en la sección anterior - pero utilizando el lenguaje *UML* -, por lo que se incide en uno de los aspectos que diferencia este diagrama del resto: el despliegue de los componentes del servidor. A continuación se presentan el diagrama utilizando servidores externos y el diagrama con los servidores embebidos dentro del principal.

7.2.1. Servidores externos

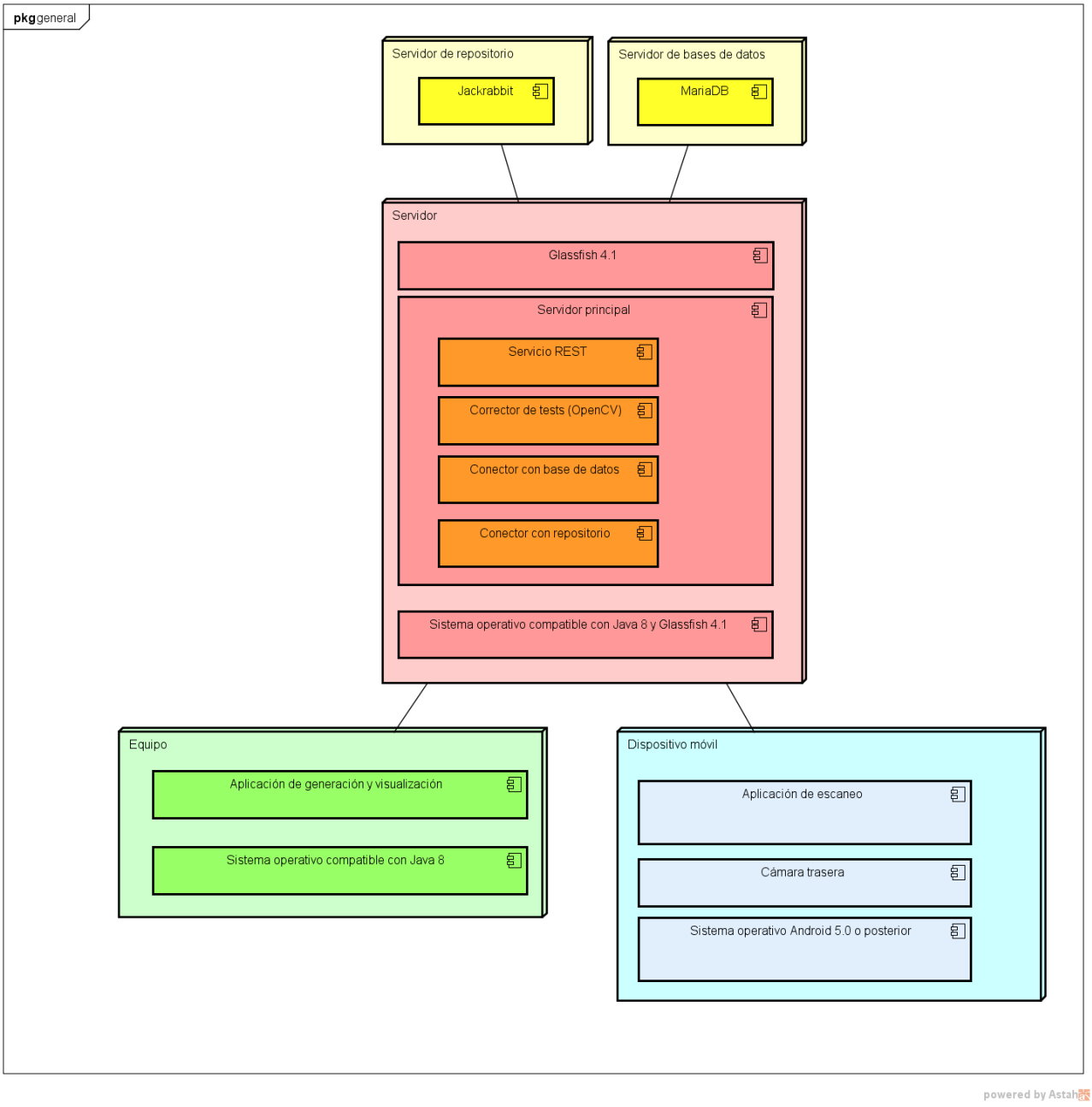

Figura 7.12: Diagrama de despliegue - Servidores externos

7.2.2. Servidores embebidos

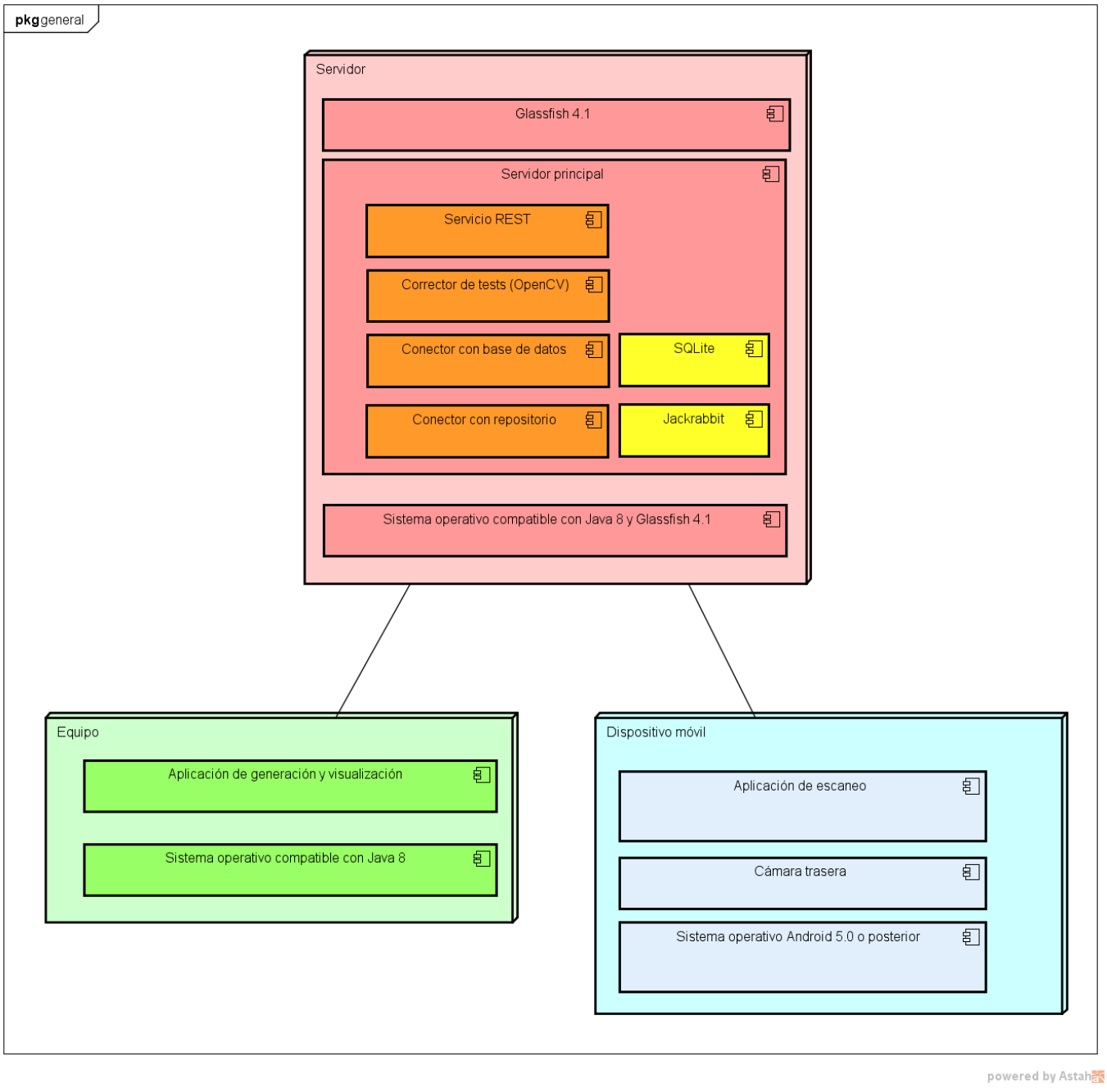

Figura 7.13: Diagrama de despliegue - Servidores embebidos

La diferencia en esta alternativa es que el servicio de base de datos y el de repositorio pasan a estar embebidos dentro del servidor principal, por lo que se convierten en un componente más del mismo. Además, se pasa de utilizar *MariaDB* a *SQLite* por los motivos expuestos en capítulos anteriores.

7.3. Diseño concreto de cada aplicación

Para terminar este capítulo cabe añadir un comentario sobre la organización de los posteriores capítulos del trabajo. Una vez mostrado el diseño general del sistema, se van a desarrollar de manera individual el diseño e implementación de cada una de las aplicaciones que componen el sistema. Ésto permite evitar una descripción excesivamente compleja del proyecto y permite describir con mayor detalle las peculiaridades de cada software.

Siguiendo lo indicado en los diagramas generales de la solución, las aplicaciones son las siguientes:

- **Aplicación de generación y visualización de tests**. Implementa el componente *Generador y visualizador* en una aplicación de escritorio que se conecta con la *Aplicación servidor*. Permite generar exámenes y visualizar los de tipo *Oficial* almacenados en el servidor.
- **Aplicación de escaneado de tests**. Implementa el componente *Reconocedor* en una aplicación para dispositivos móviles. Permite escanear y reconocer los tests. Si son de tipo *Autocorrección*, los corregirá y mostrará los resultados en pantalla. Si son de tipo *Oficial*, los enviará a la *Aplicación servidor*.
- **Aplicación servidor**. Implementa el componente *Almacenador*, y se encargar de guardar y corregir los exámenes de tipo *Oficial*.

Capítulo 8

Aplicación de generación y visualización de tests

Una vez introducida la imagen general del proyecto y el diseño de la plantilla de examen, es conveniente comenzar a detallar cada programa del sistema. Primeramente se indica el diseño, arquitectura y funcionamiento del software de generación y visualización de exámenes para mostrar secuencialmente, el orden de uso que tendría el sistema a la hora de ser utilizado por el usuario.

8.1. Introducción

Este programa es una aplicación de escritorio que permite la generación de nuevos exámenes en formato *SVG* y *PDF* a partir de la plantilla, y la visualización de los almacenados en el servidor. Dentro de esta segunda característica, incorpora también la posibilidad de corregir los tests recogidos en el servidor y la de exportar los resultados a un fichero *CSV*.

8.2. Interfaz de usuario

Se puede dividir el software en dos bloques bien diferenciados: generación y visualización. A nivel de interfaz de usuario, cada uno está contenido en una lengüeta. Ambos muestran en común una barra de opciones con los siguientes menús y componentes:

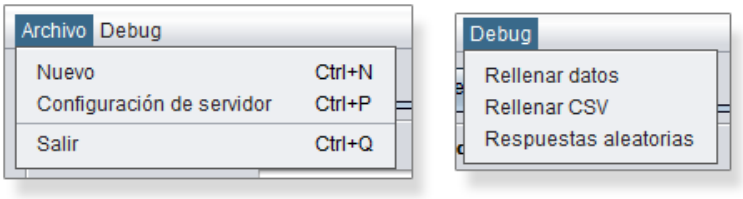

Figura 8.1: Aplicación de generación y visualización - Barra de opciones

- **Archivo**. Muestra opciones generales del programa.

- **Nuevo**. Permite generar un nuevo test, dejando todos los campos y componentes en su valor inicial.
- **Configuración de servidor**. Al pulsarlo, se mostrará un panel emergente con un campo de texto donde es posible especificar la dirección completa del servidor principal.

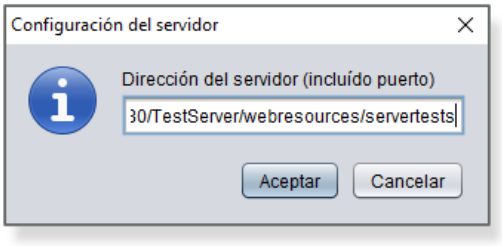

Figura 8.2: Aplicación de generación y visualización - Configuración de servidor

Este dato se guarda en el directorio raíz del programa en un fichero *JSON* de nombre '*server_config.json*'. El formato es el mostrado a continuación.

```
{"serverUrl":"http://192.168.1.106:8080/TestServer/webresources/servertests"}
```

En el caso del modo *Oficial*, esta ruta del servidor será incluída en el *QR* para ser utilizada por la aplicación móvil de reconocimiento de exámenes.

- **Salir**. Permite salir del programa.

- **Debug**. Este menú sería removido en una versión comercial o destinada al público fuera del proyecto. Incluye varias opciones para generar rápidamente tests para probar la aplicación.
  - **Rellenar datos**. Rellena todos los datos administrativos del test con información básica y válida.
  - **Rellenar CSV**. Incluye el código *NIA* de dos alumnos de ejemplo.
  - **Respuestas aleatorias**. Da respuestas aleatorias a todas las preguntas de todas las versiones del examen.

La lengüeta de generación tiene el siguiente aspecto:

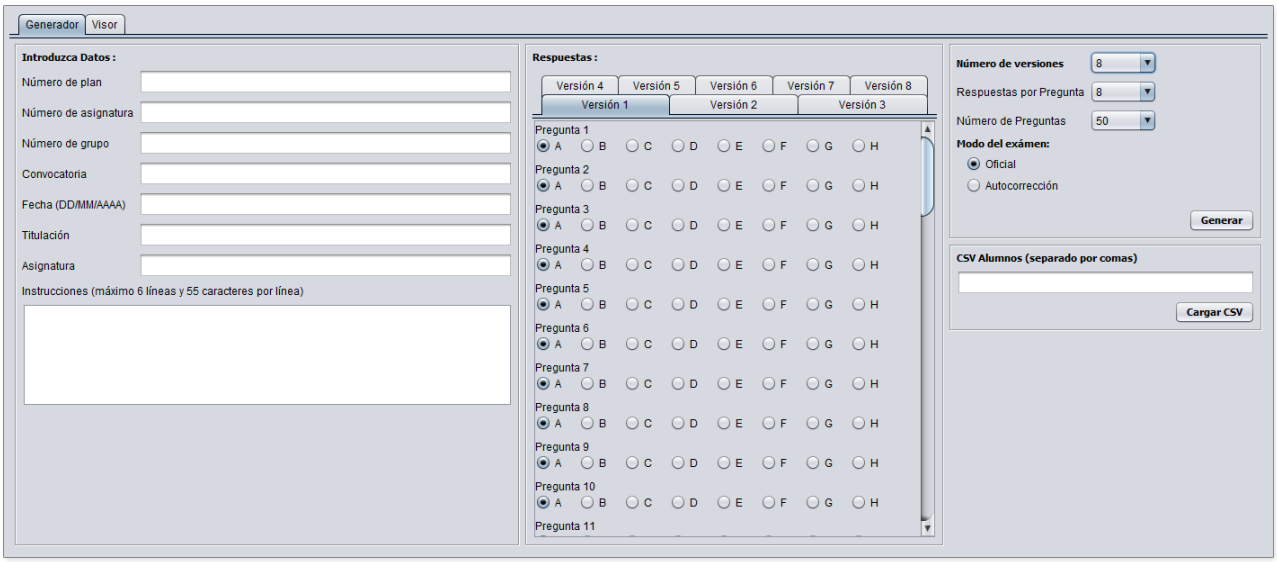

Figura 8.3: Aplicación de generación y visualización - Pestaña de generación

El panel se puede diferenciar a su vez en cuatro sub-paneles distintos, cada uno con sus componentes.

- **Panel de información administrativa**. Aquí se debe especificar la información administrativa del examen. Será la misma para todas las copias, y todos los campos son obligatorios a excepción del de *Instrucciones*.
  - **Número de plan**.
  - **Número de asignatura**.
  - **Número de grupo**.
  - **Convocatoria**.
  - **Fecha del examen**. Se debe especificar la fecha con un formato *DD/MM/AAAA*.
  - **Titulación**.
  - **Asignatura**.
  - **Instrucciones**. Este campo es opcional y permite especificar las instrucciones del test en un máximo de 6 líneas con 55 caracteres por línea.
- **Panel de preguntas y respuestas**. En este panel se muestran todas las preguntas del test con sus respuestas posibles. Utilizando los botones circulares se pueden definir la respuestas correctas. Estas preguntas se mostrarán en lengüetas, tantas como versiones tenga el examen.
- **Panel de configuración del test**. Permite definir los parámetros básicos de la *zona de test*, así como el tipo de examen.
  - **Número de versiones**. Indica el número de versiones del test -hasta un máximo de 8-.
  - **Respuestas por pregunta**. Indica el número de respuestas de cada pregunta -hasta un máximo de 8-.
  - **Número de preguntas**. Indica el número de preguntas totales del examen -hasta un máximo de 50-.
  - **Modo del examen**. Permite especificar el modo del examen -*Oficial* o *Autocorrección*-.
  - **Botón de generación de examen**. Al pulsarlo, se pedirá al usuario indicar la carpeta donde generar los ficheros *SVG* y *PDF* del test. Si se ha seleccionado el modo *Oficial*, se generarán ambos archivos para cada alumno incluido en el *Panel de datos de alumnos*.
- **Panel de datos de alumnos**. Este panel se mostrará únicamente si se ha seleccionado el modo *Oficial*. En él, se muestra un campo de texto donde es posible introducir, separado con comas siguiendo el estándar *CSV*, los códigos *NIA* de los alumnos a los que va dirigido el examen. Además, se incluye un botón *Cargar CSV*, que da la posibilidad de abrir un fichero *CSV* que contenga dichos códigos. Estos códigos se utilizarán para ser introducidos en el *QR*, de tal manera que cada copia del examen tendrá un QR diferente, dirigido exclusivamente al alumno en cuestión.

En cuanto a la lengüeta de visualización, presenta la interfaz necesaria para recibir, filtrar, gestionar y exportar los exámenes almacenados en el servidor.

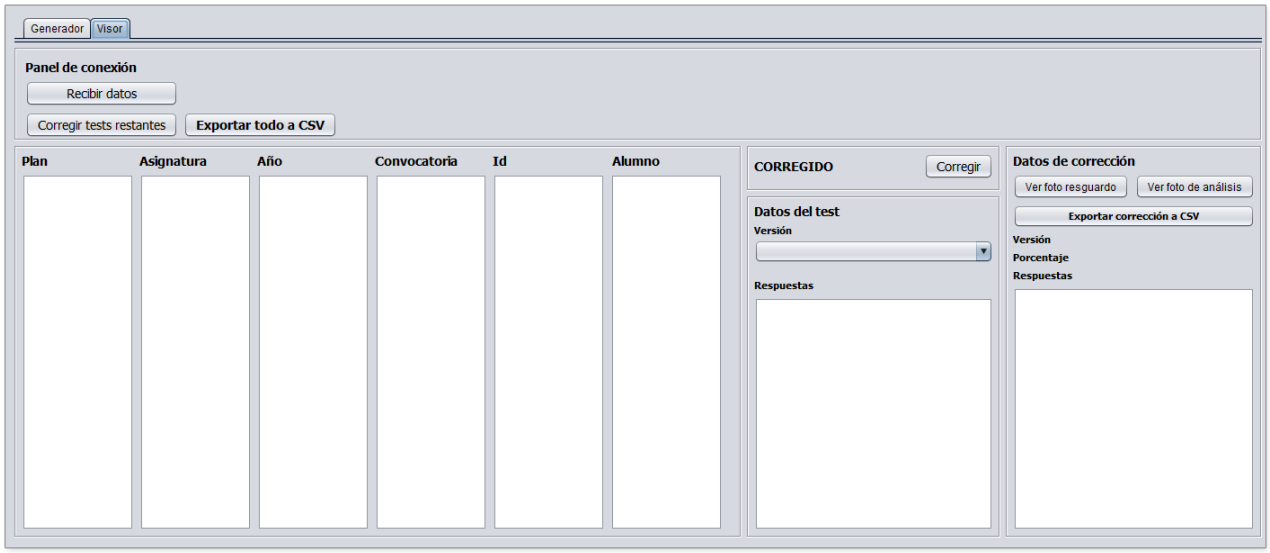

Figura 8.4: Aplicación de generación y visualización - Pestaña de visualización

Los componentes de este panel son los siguientes:

- **Panel de conexión**. Aquí se incluyen botones relacionados con la interconexión de la aplicación de escritorio con el servidor principal.
  - **Botón de recepción de datos**. Al pulsarlo envía una petición al servidor para recibir los datos de todos los exámenes almacenados. La respuesta mostrará en el campo *Plan* del *Panel de filtrado* los planes disponibles. Si no hay tests almacenados o no es posible contactar con el servidor, se informará al usuario.
  - **Botón de corrección de tests restantes**. Envía una petición al servidor para corregir todos los tests no corregidos al servidor. La respuesta indicará al usuario el número de exámenes corregidos, o, si no había tests por corregir, se mostrará que todos los test estaban corregidos.
  - **Botón *Exportar todo a CSV***. Una vez recibidos los datos, se mostrará este botón en el panel. Permite exportar los resultados de todos los exámenes recibidos a un fichero *CSV*. El formato del contenido del mismo quedará indicado en posteriores apartados.
- **Panel de filtrado**. Se encuentra dividido en seis sub-paneles: *Plan*, *Asignatura*, *Año*, *Convocatoria*, *Id* y *Alumno*. Al pulsar en uno de los valores de cada uno de ellos se irán mostrando las opciones disponibles en los campos posteriores; esto permite un filtrado rápido de todos los exámenes disponibles.
- **Panel de corrección**. una vez seleccionado el test de un alumno en el *Panel de filtrado*, se indicará aquí si el examen está corregido y, en caso de no estarlo, se mostrará una botón *Corregir* que realiza una llamada al servidor para que corrija el examen en ese momento; cuando se reciba la respuesta, se mostrarán los resultados en pantalla.
- **Panel de datos del test**. Aquí se muestran los datos iniciales del test, con las versiones y respuestas correctas que definió el profesor.
  - **Versión**. Permite elegir todas las versiones del examen.

  - **Respuestas**. Muestra las respuestas de la versión elegida.

- **Panel de datos de corrección**. Este panel se muestra únicamente si el test ha sido corregido. En él se encuentran los datos y opciones relacionados con la corrección del test realizada por el servidor.

  - **Botón de visualización de foto de resguardo**. Pide al servidor la foto de resguardo tomada desde la aplicación móvil, la descarga en la ruta raíz de la aplicación y la muestra automáticamente al usuario con el programa de visualización de imágenes por defecto definido en el sistema operativo.
  - **Botón de visualización de foto de análisis**. Realiza la misma operación que el botón anterior, pero con la foto que se utilizó para el análisis del examen.
  - **Botón *Exportar corrección a CSV***. Ejecuta el mismo procedimiento que el botón general de *Exportar todo a CSV*, pero en este caso utiliza únicamente el test actual.
  - **Versión**. Versión del test seleccionada por el alumno.
  - **Porcentaje**. Porcentaje de respuestas correctas del test.
  - **Respuestas**. Muestra las respuestas seleccionadas por el alumno. Para ayudar a analizar las respuestas, se muestran en color rojo las incorrectas, no respondidas o con múltiple respuestas, y en verde las correctas.

Toda la interfaz gráfica mostrada se ha diseñado utilizando la biblioteca *Java Swing* y utilizando en gran medida el editor gráfico que de la misma que aporta *NetBeans*.

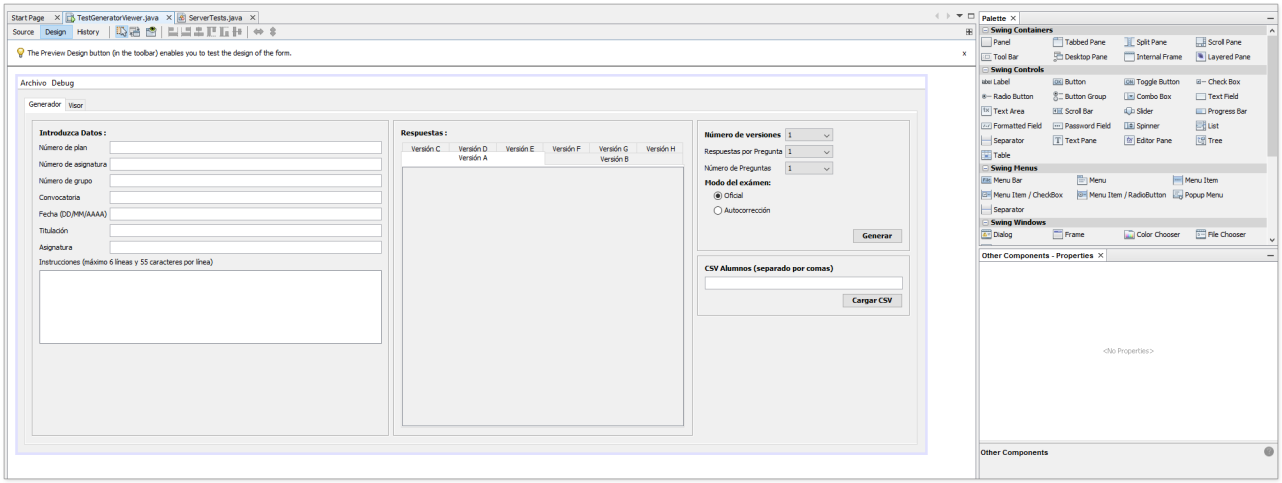

Figura 8.5: Aplicación de generación y visualización - Entorno *NetBeans*

Mientras se diseña visualmente la aplicación, *NetBeans* va generando código automáticamente para componer la interfaz. Éste código se puede ver en la clase, pero no se puede modificar libremente desde el entorno de desarrollo a excepción de ciertas características. Se ha intentado aprovechar al máximo estas capacidades para mantener la separación de modelo, vista y controlados en el proceso de trabajo, pero hay ciertos elementos que se han configurado directamente desde el código por diferentes motivos:

- Algunos componentes, como los paneles de respuestas y versiones, se generan dinámicamente según la configuración del examen elegida por el usuario. En el editor se crea un diseño base de dicho componente, y luego desde el código de la aplicación se réplica y modifica hasta conseguir el resultado adecuado.
- Los campos de texto han requerido limitar el número de caracteres que pueden ser introducidos en los mismos, por lo que se ha añadido desde el código un controlador escrito para al ocasión que controla dicha característica.

- El elemento que muestra en formato de lista las respuestas de cada corrección en el visualizador incluye un componente externo que cambia el color de los elementos de la lista según éstos correspondan con las respuestas verdaderas del test o no.

8.3. Formato de exportación *CSV*

Durante la descripción general de las opciones de la aplicación se ha hablado sobre la posibilidad de exportar los tests corregidos a formato *CSV*. La elección de este formato se debe a que es un estándar para representar datos en forma de tabla, soportado por la mayor parte de sistemas operativos y programas de edición. Dentro del fichero generado, se sigue un esquema claro para presentar la información de los tests.

Primeramente se muestra una leyenda sobre la representación de las respuestas del test dadas por los alumnos.

```
Leyenda␣de␣respuestas␣de␣alumnos.
1␣>␣Respuesta␣correcta.␣Se␣añade␣entre␣paréntesis␣la␣respuesta.
0␣>␣Respuesta␣incorrecta.␣Se␣añade␣entre␣paréntesis␣la␣respuesta.
-␣>␣No␣respondida␣o␣respuesta␣no␣reconocida.
*␣>␣Múltiples␣respuestas.
```

Para indicar los datos de los tests se utiliza un formato que sigue directamente lo mostrado en el *Panel de filtrado* del visualizador, con los siguientes campos:

1. **Plan**.
2. **Asignatura**.
3. **Año**.
4. **Convocatoria**.
5. **Test ID**. Indica el identificador interno del test dado por el servidor principal cuando fue generado.
6. **Alumno**. Código *NIA* del alumno.
7. **Versión**.
8. **Respuestas**. Después del campo anterior, se añaden 50 campos, uno para cada respuesta. Siempre se añaden 50, ya que es el total de preguntas que puede tener un test, y si se ha seleccionado la opción de exportar todos los exámenes a *CSV*, es probable que cada test tenga un número de preguntas distinto; los que tengan menos de 50, tendrán ese campo en blanco.
9. **Porcentaje**. Finalmente, se incluye un campo que indica el porcentaje de respuestas correctas.

A continuación de la leyenda se indican los datos correctos de los test -es decir, los indicados por el profesor a la hora de generarlos-. Para mantener el mismo esquema a la hora de representar tanto esto como los exámenes corregidos de los alumnos -el esquema indicado antes-, en el campo *Alumno* se indica el valor *Respuestas correctas* y en el campo *Porcentaje* un valor de *100*. Además, al ser una referencia para saber las respuestas correctas, en vez de seguir la leyenda se indican directamente las respuestas correctas. Un ejemplo sería el siguiente:

```
Plan;Asignatura;Año;Convocatoria;Test␣ID;Alumno;Versión;Respuesta␣1;
Respuesta␣2;Respuesta␣3;Respuesta␣4;Respuesta␣5;Respuesta␣6;
Respuesta␣7;Respuesta␣8;Respuesta␣9;Respuesta␣10;Respuesta␣11;
Respuesta␣12;Respuesta␣13; Respuesta␣14;Respuesta␣15;Respuesta␣16;
Respuesta␣17;Respuesta␣18;Respuesta␣19;Respuesta␣20;Respuesta␣21;
Respuesta␣22;Respuesta␣23;Respuesta␣24;Respuesta␣25;Respuesta␣26;
Respuesta␣27;Respuesta␣28;Respuesta␣29;Respuesta␣30;Respuesta␣31;
Respuesta␣32;Respuesta␣33;Respuesta␣34;Respuesta␣35;Respuesta␣36;
Respuesta␣37;Respuesta␣38;Respuesta␣39;Respuesta␣40;Respuesta␣41;
Respuesta␣42;Respuesta␣43;Respuesta␣44;Respuesta␣45;Respuesta␣46;
Respuesta␣47;Respuesta␣48;Respuesta␣49;Respuesta␣50;Porcentaje
7775;3330;2016;Ordinaria;0;Respuestas␣correctas;A;C;C;B;B;B;A;A;B;
A;B;C;A;A;C;A;A;A;A;C;A;D;B;B;D;C;A;C;A;A;A;A;A;B;A;A;A;A;A;A;A;A;
A;C;B;E;A;C;E;E;A;100
```

Por último, siguiendo el esquema general y con lo indicado en la leyenda, se incluyen los tests corregidos. Siguiendo el ejemplo anterior:

```
Plan;Asignatura;Año;Convocatoria;Test␣ID;Alumno;Versión;Respuesta␣1;
Respuesta␣2;Respuesta␣3;Respuesta␣4;Respuesta␣5;Respuesta␣6;
Respuesta␣7;Respuesta␣8;Respuesta␣9;Respuesta␣10;Respuesta␣11;
Respuesta␣12;Respuesta␣13;Respuesta␣14;Respuesta␣15;Respuesta␣16;
Respuesta␣17;Respuesta␣18;Respuesta␣19;Respuesta␣20;Respuesta␣21;
Respuesta␣22;Respuesta␣23;Respuesta␣24;Respuesta␣25;Respuesta␣26;
Respuesta␣27;Respuesta␣28;Respuesta␣29;Respuesta␣30;Respuesta␣31;
Respuesta␣32;Respuesta␣33;Respuesta␣34;Respuesta␣35;Respuesta␣36;
Respuesta␣37;Respuesta␣38;Respuesta␣39;Respuesta␣40;Respuesta␣41;
Respuesta␣42;Respuesta␣43;Respuesta␣44;Respuesta␣45;Respuesta␣46;
Respuesta␣47;Respuesta␣48;Respuesta␣49;Respuesta␣50;Porcentaje
7775;3330;2016;Ordinaria;0;alumno0001;A;-;*;1␣(C);-;-;-;-;-;-;-
;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-;-
;0␣(A);-;1␣(A);-;1␣(E);1␣(E);0␣(B);0
```

8.4. Arquitectura

Para la construcción de esta aplicación se ha utilizado un diseño híbrido entre una arquitectura de tres capas y un patrón de diseño *MVC- Modelo, Vista y Controlador* -, adaptado además a las peculiaridades de su implementación al haber utilizado *Java Swing* como biblioteca para la interfaz gráfica y *NetBeans* como entorno de desarrollo.

8.4.1. Diagrama de clases

En el diagrama *UML* mostrado a continuación se puede ver la arquitectura general de la aplicación. Cabe destacar que en este primer diagrama no se muestran parámetros ni métodos para otorgar en primer lugar una imagen general de la asociación existente entre las clases del sistema.

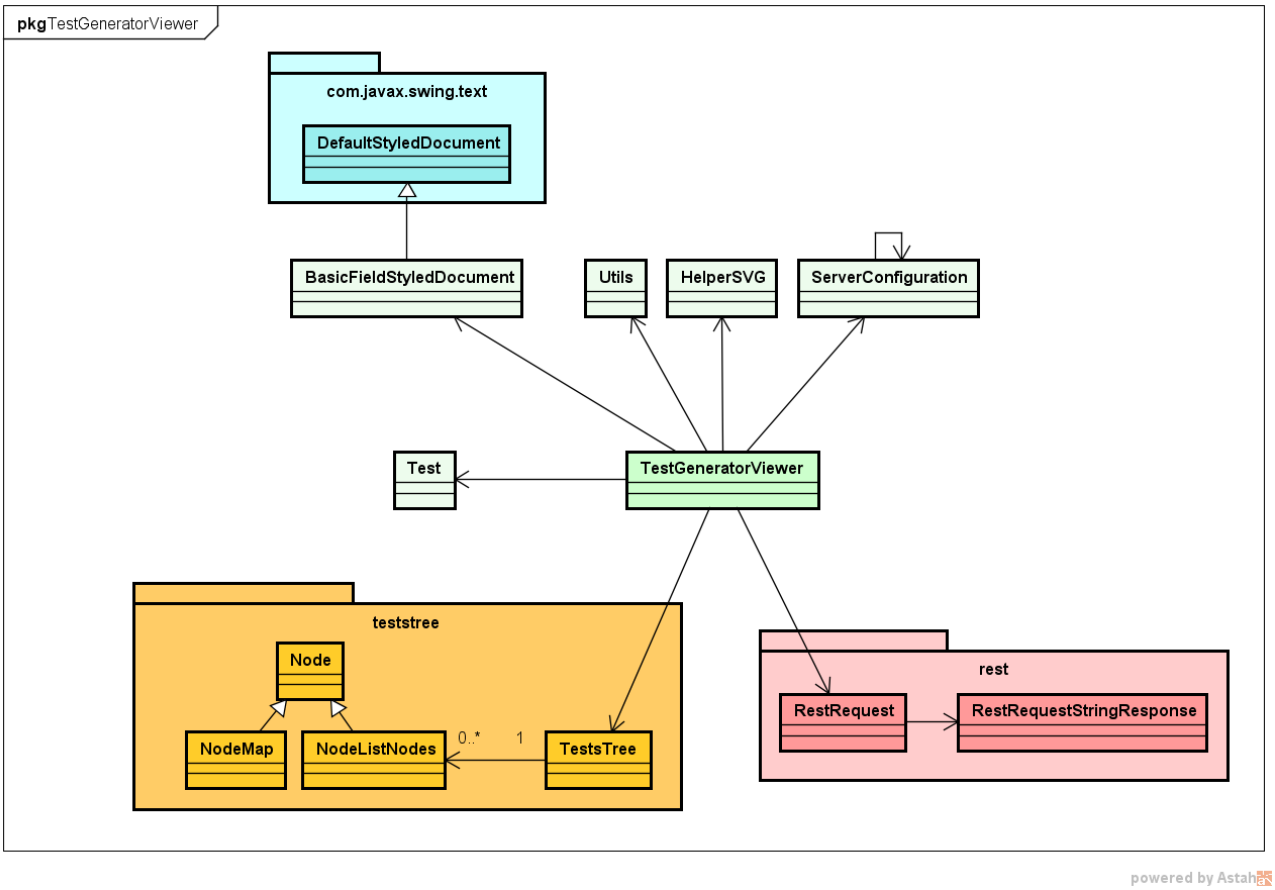

Figura 8.6: Aplicación de generación y visualización - Diagrama de clases general

Los paquetes mostrados explícitamente en el diagrama son para indicar la separación en pequeños componentes de algunas clases del código y así entender mejor dicha división y su funcionamiento. Por lo tanto, a nivel de código la nomenclatura de los paquetes es más amplia y están todos contenidos a su vez en otros paquetes.

De manera más específica, a continuación se muestra un esquema para cada paquete de los que contiene la aplicación. En este caso sí se muestran parámetro y métodos, pero donde el número de ellos era excesivo, solamente se han incluido los destacables.

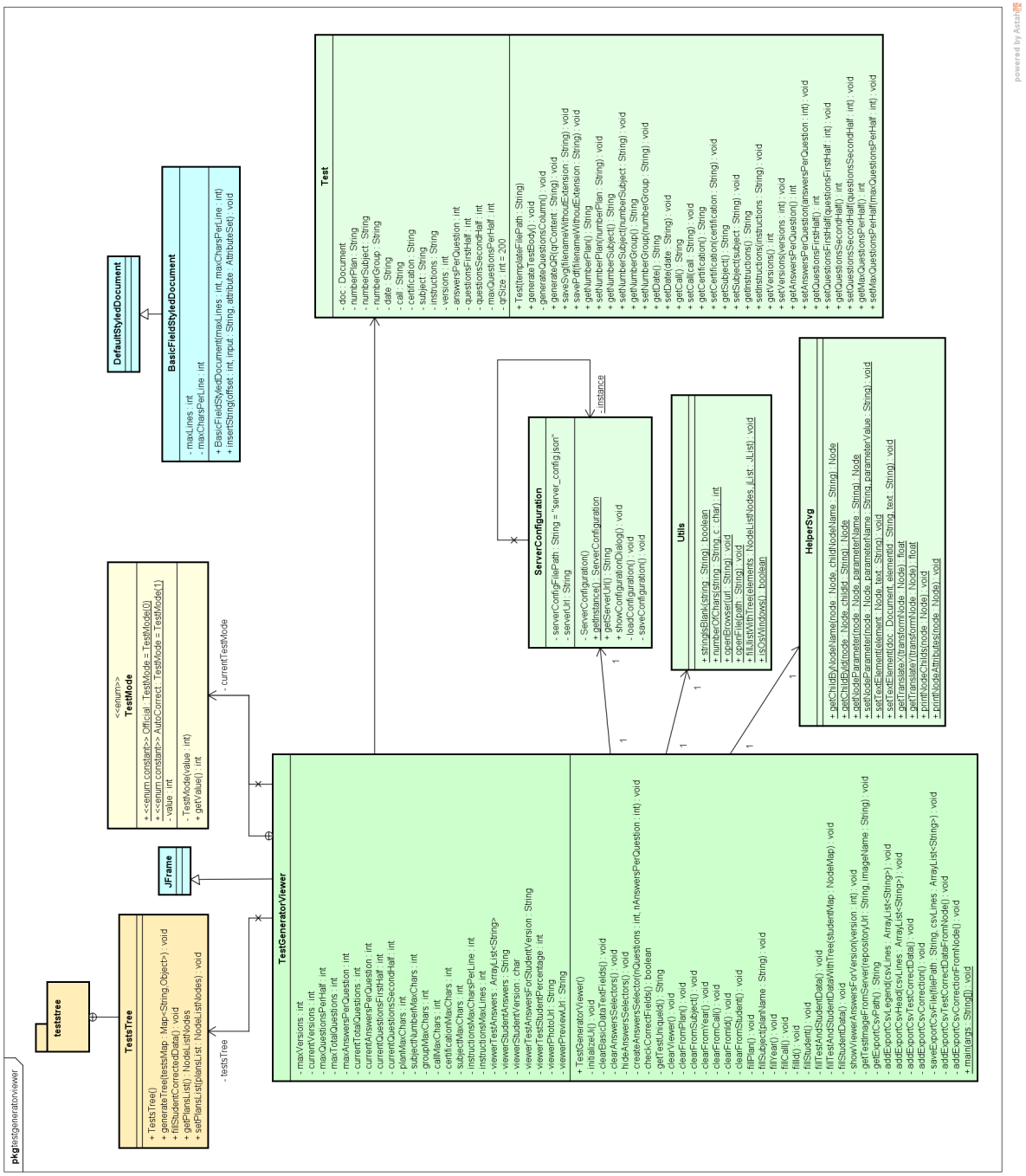

Figura 8.7: Aplicación de generación y visualización - Diagrama de clases general detallado

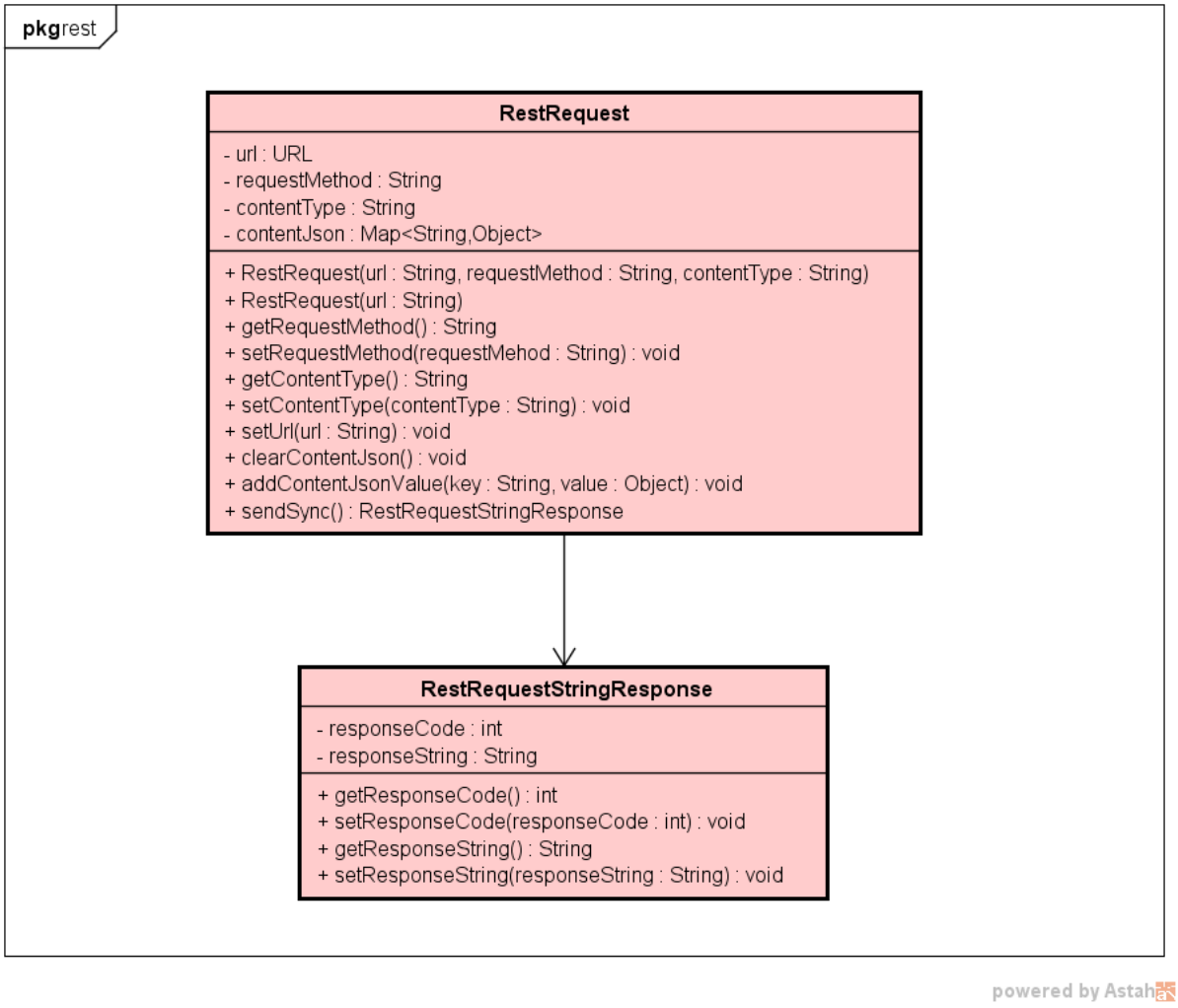

Figura 8.8: Aplicación de generación y visualización - Diagrama de clases detallado del paquete *REST*

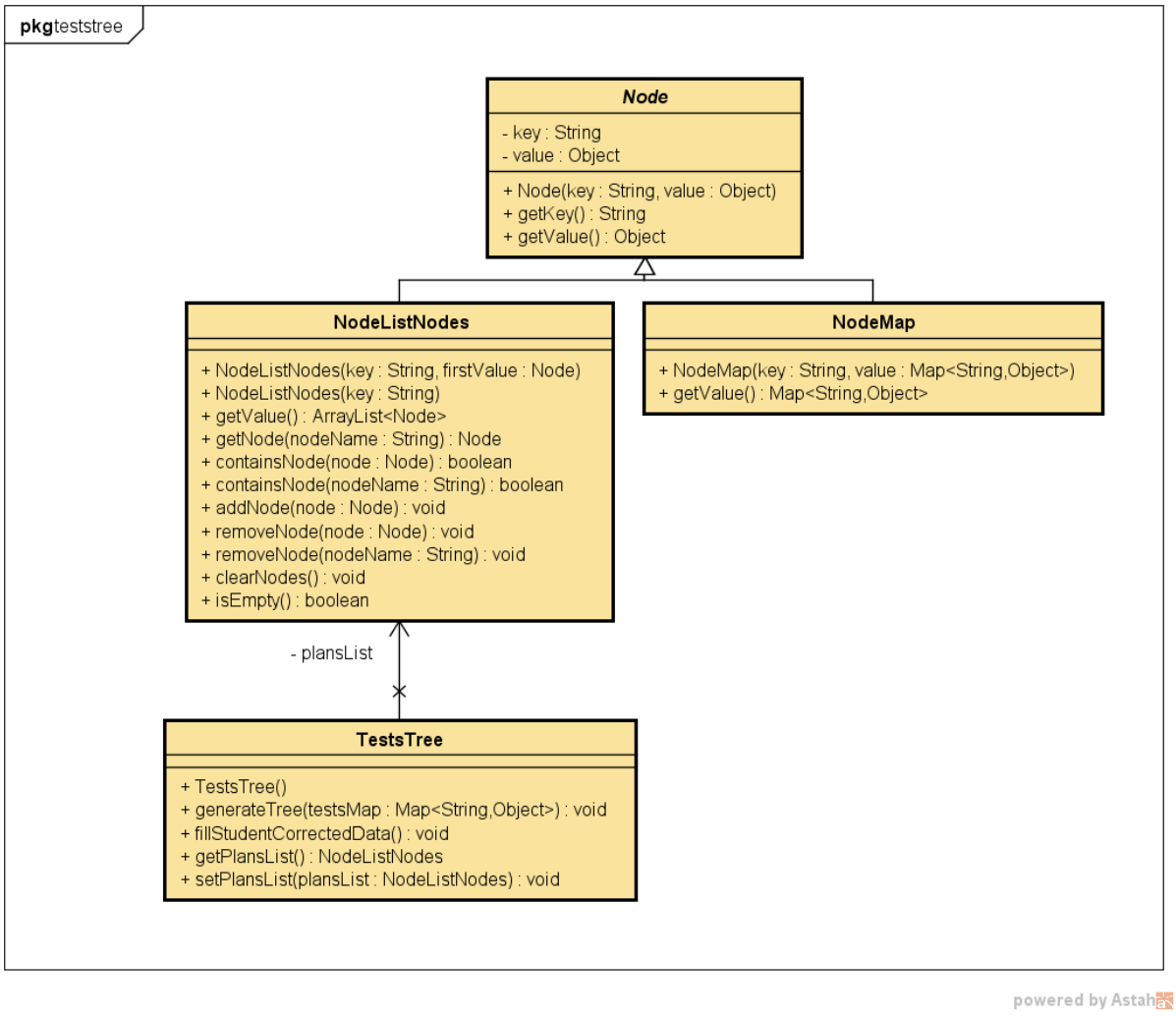

Figura 8.9: Aplicación de generación y visualización - Diagrama de clases detallado del paquete de nodos

La función de cada clase, divididas por paquetes, es la siguiente:

- **Paquete general**. Es el paquete general de la aplicación y alberga el resto de paquetes y las clases principales o sin categoría específica.

  - **TestGeneratorViewer**. Es la clase principal, contiene la mayoría de la lógica de generación y visualización de tests. Esto abarca tanto características indicadas en la descripción de la aplicación - configurar y generar tests, visualizarlos, corregirrlos, exportar correcciones, etc -, como el código de generación y configuración de la interfaz gráfica y la interacción con el usuario. Además, es la clase lanzadora de la aplicación.
  - **BasicFieldStyledDocument**. Es una pequeña clase que hereda de la clase *DefaultStyledDocument* contenida en la biblioteca *Java Swing*. Su función es dotar a los campos de texto de un máximo número de líneas y caracteres. La necesidad de esta clase surgió en las fases finales de la implementación, debido a que *Java Swing* no posee una función por defecto para limitar la introducción de texto en los campos.
  - **HelperSvg**. Provee métodos estáticos para el manejo de los componentes *SVG*, principalmente el manejo de sus nodos.

  - **ServerConfiguration**. Sigue el patrón *Singleton* e implementa el almacenamiento y gestión de la ruta del servidor. También permite mostrar al usuario un diálogo en la interfaz gráfica para que introduzca la ruta.
  - **Test**. Modela un examen, con sus campos administrativos, información de la *zona de test* y el propio documento *SVG*. Provee métodos para generar dicho documento a partir de los datos básicos y para exportarlo a ficheros en formato *SVG* y *PDF*.
  - **Utils**. De manera similar a *HelperSVG*, es una clase con métodos estáticos que provee varias funciones sencillas utilizadas comunmente por el programa; entre ellas se encuentra una para indicar si una cadena está completamente vacía - es decir, si es nula, no tiene ningún caracter, o todos los que tiene son espacios -, o una para abrir un determinado fichero con el programa por defecto dado por el sistema operativo.

- **Paquete para peticiones *REST***. Este paquete contiene dos clases para facilitar el envío de peticiones *REST* y el análisis del resultado de las mismas.

  - **RestRequest**. Permite generar y enviar una petición *REST* síncrona a una determinada dirección; además, devuelve el resultado de la misma. Permite configurar el contenido de la petición libremente o de añadir directamente valores para generar un *JSON* que son gestionados para su envío por la propia clase.
  - **RestRequestStringResponse**. Encapsula una respuesta a una petición *REST* realizada mediante la clase *RestRequest*.

- **Paquete de árbol jerárquico de nodos para tests**. Incluye los componentes necesarios para generar un árbol de nodos a partir de los datos *JSON* de los tests enviados por el servidor.

  - **Node**. Clase abstracta que modela un nodo como un par clave-valor, donde la clave es siempre una cadena y el valor puede ser cualquier objeto.
  - **NodelistNodes**. Hereda de la clase *Node*, definiendo el valor del par como una lista de objetos *Node*. Añade también métodos para facilitar la gestión de la lista.
  - **NodeMap**. Hereda de la clase *Node*, definiendo el valor del par como un mapa de pares clave-valor, donde la clave es una cadena y el valor un objeto cualquiera.
  - **TestsTree**. Clase principal de este paquete, que controla el árbol de nodos de tests. Organiza, de más general a más concreto, los test por su *número de plan*, *número de asignatura*, *año*, *convocatoria*, *identificación* y *código NIA*. De esta manera, las cinco primeras divisiones son nodos con listas de nodos - *NodeListNode* -, y la última un nodo con todos los datos del test y su corrección relativa al alumno - *NodeMap* -.

8.4.2. Patrones de diseño

Dentro del diseño y código de la aplicación se puede encontrar el uso de varios patrones de diseño:

- **Modelo, vista, controlador**. Es un patrón muy utilizado que separa el modelo lógico de los datos, su presentación al usuario y el manejo de las entradas dadas por el usuario y como éstas modifican el modelo. Se ha utilizado a varios niveles para el diseño general de la aplicación.

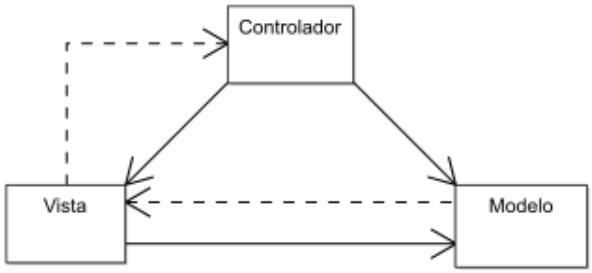

Figura 8.10: Esquema patrón *Modelo, vista, controlador*

A pesar de lo indicado, debido al funcionamiento de *Java Swing* y, en general, el de la mayor parte de *frameworks* de interfaces gráficas actuales, la separación de la vista y el controlador es difusa. Esto da pie a las siguientes peculiaridades:

- Se puede considerar que la vista se encuentra manejada por las propias bibliotecas de *Java Swing*, pero desde la clase principal *TestGeneratorViewer* se realiza la inicialización de la misma en base al diseño de la interfaz gráfica y ciertas modificaciones menores.
- El controlador principal queda implementado en la clase principal *TestGeneratorViewer*, pero se puede encontrar un pequeño controlador en la clase *ServerConfiguration*, ya que se encarga de llamar a la biblioteca *Java Swing* para pedir la ruta del servidor al usuario. De la misma manera, otras clases realizan pequeñas llamadas a dicha biblioteca para mostrar errores o tareas similares. Se realizó así para facilitar la lectura y mantenimiento del código.

De manera menos notoria que los casos anteriores, también se puede encontrar una cierta mezcla en la división con el modelo. Por una parte, en el apartado dedicado al visualizador se puede considerar que el modelo se encuentra en el servidor y que esta aplicación simplemente maneja el modelo recibido - es decir, la tarea del controlador -; pero, al organizar los tests en el árbol de nodos y poder exportarlos como *CSV*, se ha valorado como un pequeño modelo. En cuanto al apartado del generador, se puede encontrar el modelo en del examen en la clase *Test* y un pequeño modelo relativo al manejo de la ruta del servidor en la clase *ServerConfiguration*.

- **Singleton**. Este patrón permite garantizar la existencia de una única instancia de una determinada clase durante toda la ejecución de un programa, así como la proporciona un punto de acceso global desde toda la aplicación.

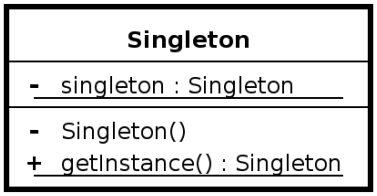

Figura 8.11: Esquema patrón *Singleton*

Se ha utilizado para implementar la clase *ServerConfiguration*, ya que dicha configuración es única en la aplicación y el acceso a la ruta del servidor es necesario desde varios puntos de la aplicación.

8.5. Implementación

Entrando ya concretamente en el código utilizado en las clases escritas, en esta sección se van a mostrar algunos detalles del mismo que se han considerado fundamentales o particularmente interesantes. En general no se va a mostrar código específico sobre la presentación de la aplicación debido a que no es el objeto de este proyecto y a que, como ya se ha mencionado, utiliza la biblioteca *Java Swing* y el entorno de desarrollo *NetBeans*, que proveen un ámbito bien definido sobre el que implementar las peculiaridades relativas a: la distribución y uso de botones, campos de texto, etc.

8.5.1. Test

Un punto interesante del código de la clase *Test*, es como se generan las versiones, preguntas y respuestas del examen a partir de la plantilla. Inicialmente, ésta tiene el siguiente aspecto:

Figura 8.12: Plantilla de test sin modificar

Como se puede observar, hay un componente de cada tipo ya generado en la plantilla:

- Una versión, nombrada *A*.
- Una pregunta número cero.
- Una respuesta a la pregunta anterior, nombrada *A*.

Dado que un fichero vectorial *SVG* es en el fondo un fichero *XML*, accediendo a los datos de cada componente, y con algunas medidas de referencia - indicadas en parte en capítulos anteriores de este documento -, es posible modificar los componentes existentes y generar nuevos a partir de ellos. Utilizando estas técnicas es como se termina de generar el test final que ha diseñado el usuario.

Primeramente, se extraen algunos datos básicos de la plantilla, accediendo a parámetros de los nodos de ciertos elementos. Como ejemplo, se muestra la extracción de algunos de ellos mediante el uso de la clase *HelperSVG*:

```
1  //Get references for version.
2  Node versionTitle = doc.getElementById("versionTitle");
3  Node versionRoot = doc.getElementById("versionRoot");
4  Node versionRootTransform = HelperSvg.getNodeParameter(versionRoot, "transform");
5  Node versionCircle = HelperSvg.getChildById(versionRoot, "versionCircle");
6  Node versionCircleWidth = HelperSvg.getNodeParameter(versionCircle, "width");
7
8  //Get references for questions ans answers.
9  Node questionRoot = doc.getElementById("questionRoot");
10 Node questionRootTransform = HelperSvg.getNodeParameter(questionRoot,
       "transform");
11 Node answerRoot = HelperSvg.getChildById(questionRoot, "answerRoot");
12 Node answerRootTransform = HelperSvg.getNodeParameter(answerRoot, "transform");
13 Node answerCircle = HelperSvg.getChildById(answerRoot, "answerCircle");
14 Node answerCircleWidth = HelperSvg.getNodeParameter(answerCircle, "width");
15
16 //Get corner size.
17 Node topLeftCorner = doc.getElementById("topLeftCorner");
18 float cornerHeight = Float.parseFloat(
19 HelperSvg.getNodeParameter(topLeftCorner, "height").getNodeValue());
```

Tras esto, se utilizan los datos introducidos por el usuario para generar las versiones, preguntas y respuestas del test. Tomando como referencia la generación de versiones, se realiza lo siguiente:

```
1  //If just one version, remove version marks.
2  if (getVersions() < 2) {
3    root.removeChild(versionTitle);
4    root.removeChild(versionRoot);
5  } else {
6    //Generate all version marks.
7    for (int i = 0; i < getVersions() - 1; i++) {
8      //Clone original answer.
9      Node clone = versionRoot.cloneNode(true);
10     //Translate in X.
11     HelperSvg.setNodeParameter(clone, "transform",
12       "translate(" + totalVersionOffset + ", " + versionY + ")");
13     //Change letter (beginning in B).
```

```
14      Node versionLetter = HelperSvg.getChildById(clone, "versionLetter");
15      HelperSvg.setTextElement(versionLetter,
16        "" + Character.toChars(66 + i)[0]);
17      //Add to root.
18      root.appendChild(clone);
19      //Increase offset.
20      totalVersionOffset += versionOffset;
21    }
22  }
```

La creación de preguntas y respuestas funciona de manera similar. En primer lugar se generan las respuestas que tiene cada pregunta en la pregunta inicial, para así poder clonar directamente para el resto - ya que todas las preguntas del examen tiene el mismo número de respuestas -.

```
1  //Get answers circle width.
2  float circleWidth = Float.parseFloat(answerCircleWidth.getNodeValue());
3  //Get initial answer X translate.
4  float totalAnswerOffset = HelperSvg.getTranslateX(answerRootTransform);
5  //Increase by answer offset.
6  float answerOffset = circleWidth + circleWidth * 0.4f;
7  totalAnswerOffset += answerOffset;
8  //Generate all answers (first answers is already in the template).
9  for (int i = 0; i < getAnswersPerQuestion() - 1; i++) {
10   //Clone original answer.
11   Node clone = answerRoot.cloneNode(true);
12   //Translate in X.
13   HelperSvg.setNodeParameter(clone, "transform",
14     "translate(" + totalAnswerOffset + ", 0)");
15   //Change letter (beginning in B).
16   Node answerLetter = HelperSvg.getChildById(clone, "answerLetter");
17   HelperSvg.setTextElement(answerLetter,
18     "" + Character.toChars(66 + i)[0]);
19   //Add to the question.
20   questionRoot.appendChild(clone);
21   //Increase offset.
22   totalAnswerOffset += answerOffset;
23 }
```

A continuación se clona la pregunta inicial hasta cubrir todas las definidas en el test, cada vez con una de las mitades de la *zona de test*. En este caso, al ser una tarea algo más larga que las dos anteriores y al realizarse una vez en cada parte, se encapsuló en un método.

```
1  /**
2  * Generate a columns of questions.
3  *
4  * @param root Test general node.
5  * @param questionRoot Questions root node.
6  * @param questionsInColumn Number of questions.
7  * @param initialQuestionY Initial question Y.
8  * @param firstQuestionNumber First question mark number.
9  * @param questionRootTranslateX Questions root X translation.
10 * @param questionOffsetY Y offset between questions.
11 * @param cloneFirstQuestion Clone or not first question.
```

```
12  */
13  private void generateQuestionsColumn(Node root, Node questionRoot,
14  int questionsInColumn,
15  float initialQuestionY,
16  int firstQuestionNumber,
17  float questionRootTranslateX,
18  float questionOffsetY,
19  boolean cloneFirstQuestion) {
20    //Set initial question number.
21    int questionNumber = firstQuestionNumber;
22    //Set initial Y.
23    float questionRootTranslateY = initialQuestionY;
24    //Add all questions.
25    Node clone = questionRoot;
26    for (int i = 0; i < questionsInColumn; i++) {
27      //Create new clone.
28      if (cloneFirstQuestion || i > 0) {
29        clone = questionRoot.cloneNode(true);
30      }
31      //Translate.
32      HelperSvg.setNodeParameter(clone, "transform",
33        "translate(" + questionRootTranslateX + ","
34        + questionRootTranslateY + ")");
35      //Change question number.
36      Node questionText = HelperSvg.getChildById(clone, "questionText");
37      HelperSvg.setTextElement(questionText, "00."
38        + (questionNumber < 10 ? "0" + questionNumber : questionNumber)
39        + ":");
40      questionNumber++;
41      //Add to the document.
42      if (cloneFirstQuestion || i > 0) {
43        root.appendChild(clone);
44      }
45      //Increase translate Y.
46      questionRootTranslateY += questionOffsetY;
47    }
48  }
```

8.5.2. Utils

De los métodos de ayuda incluidos en esta clase, se pueden destacar los métodos para abrir un enlace en el navegador por defecto del sistema operativo y el que permite abrir un fichero con el programa por defecto. El método para abrir el navegador es el siguiente:

```
1   /**
2   * Open given URL in default browser. If it fails, error messages are shown.
3   *
4   * @param url URL to be opened.
5   */
6   public static void openBrowser(String url) {
7     //Open URL in default browser.
8     if (url != null) {
9       Desktop desktop = Desktop.isDesktopSupported() ? Desktop.getDesktop() : null;
10      if (desktop != null && desktop.isSupported(Desktop.Action.BROWSE)) {
```

```
11          try {
12            desktop.browse(new URI(url));
13          } catch (URISyntaxException | IOException ex) {
14            JOptionPane.showMessageDialog(null, "No es posible abrir el navegador
                 para la URL '"
15              + url + "'.",
16              "No hay URL", JOptionPane.ERROR_MESSAGE);
17          }
18        }
19    } else {
20      JOptionPane.showMessageDialog(null, "URL no encontrada.",
21      "No hay URL", JOptionPane.ERROR_MESSAGE);
22    }
23  }
```

Para abrir el navegador se utiliza la clase *Desktop* dada por la biblioteca *AWT*[1] - *Abstract Window Toolkit* - de *Java*. Esta clase sirve principalmente para ofrecer un interfaz gráfico de usuario - de manera similar a como hizo posteriormente *Java Swing* -, pero también provee ciertas funciones para abstraer y facilitar la comunicación entre el código *Java* y ciertas características propias del sistema operativo.

Por su parte, el método para abrir el fichero muestra un esquema similar pero utilizando otro método de la clase *Desktop*:

```
1   /**
2   * Open file with default OS software.
3   * @param path File path as String.
4   */
5   public static void openFile(String path) {
6     //Open file in default software.
7     if (path != null) {
8       Desktop desktop = Desktop.isDesktopSupported() ? Desktop.getDesktop() : null;
9       if (desktop != null && desktop.isSupported(Desktop.Action.OPEN)) {
10        try {
11        desktop.open(new File(path));
12        } catch (IOException ex) {
13          JOptionPane.showMessageDialog(null, "No es posible abrir el programa por
                 defecto para '"
14              + path + "'.",
15              "Error al abrir", JOptionPane.ERROR_MESSAGE);
16        }
17      }
18    } else {
19      JOptionPane.showMessageDialog(null, "Ruta no encontrada.",
20      "No existe la ruta", JOptionPane.ERROR_MESSAGE);
21    }
22  }
```

8.5.3. RestRequest

Lo más destacable de esta clase es como se generan y manejan las peticiones *REST* que se realizan al servidor. Para ello se utiliza la clase *HttpURLConnection* de la biblioteca *Java Net*, que permite realizar peticiones *HTTP* de

[1] https://en.wikipedia.org/wiki/Abstract_Window_Toolkit

varios tipos. Como indica el nombre de esta clase, se especializa en realizar peticiones *REST*, ya que es el interfaz utilizado por el servidor principal del sistema.

```
//Prepare REST request.
HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
urlConnection.setDoOutput(true);
urlConnection.setRequestMethod(requestMethod);
//Put content if defined.
byte[] contentBytes = null;
if (contentJson != null) {
  String contentString = JSONUtil.toJSON(contentJson);
  if (contentString != null) {
    contentBytes = contentString.getBytes();
    if (contentBytes != null) {
      urlConnection.setRequestProperty("Content-Type", contentType);
      urlConnection.setRequestProperty("charset", "utf-8");
      urlConnection.setRequestProperty("Content-Length",
      Integer.toString(contentBytes.length));
    } else {
      System.err.println("Not generated JSON content bytes.");
    }
  } else {
    System.err.println("Not generated JSON content string.");
  }
}
```

Tras comprobar que los datos del contenido de la petición están definidos, se convierte el mapa de parámetros *JSON* en una cadena que se envía en *bytes* como contenido de la petición. La clase además incluye métodos para añadir nuevos parámetros a dicho mapa.

En cuanto al envío y recepción, se realiza una conexión síncrona *HTTP* y después se recoge el resultado y su código; éstos se empaquetan dentro de la clase *RestRequestStringResponse*, que será devuelta al terminar el método.

```
//Send REST content (if exists).
if (contentBytes != null) {
  try (DataOutputStream wr = new
      DataOutputStream((urlConnection.getOutputStream()))) {
    wr.write(contentBytes);
  } catch (Exception e) {
    System.err.println("Error sending data to server!");
  }
}

//Prepare response.
RestRequestStringResponse response = new RestRequestStringResponse();

//Get response code.
response.setResponseCode(urlConnection.getResponseCode());

//Get response string.
BufferedReader br = new BufferedReader(new InputStreamReader(
  urlConnection.getInputStream()));
  String out, completeOut = "";
```

```
20 while ((out = br.readLine()) != null) {
21   completeOut += out;
22 }
23 br.close();
24
25 //If not response, put null.
26 response.setResponseString(!completeOut.equals("") ? completeOut : null);
27
28 //Disconnect.
29 urlConnection.disconnect();
```

Capítulo 9

Aplicación de escaneado de tests

Desde un punto de vista puramente técnico, el que posiblemente es el punto más importante del proyecto es el escaneo en tiempo real que se realiza sobre los exámenes. Tanto éste como su corrección se realizan sobre una imagen con una resolución de 480 píxeles horizontales y 640 verticales. Como se ha indicado anteriormente, este algoritmo está implementado en diferentes lenguajes tanto en esta aplicación como en el servidor, pero la escritura original del código es la realizada en lenguaje *C* para esta aplicación móvil. Además de ser la implementación inicial, ésta tiene especial interés porque se optimizó para conseguir que funcionará en tiempo real sobre las imágenes recibidas por la cámara del dispositivo.

9.1. Introducción

Este software se ha escrito en *Java* utilizando el entorno de desarrollo *Eclipse* con el fin de ser utilizado en dispositivos *Android*. En una fase inicial del proyecto, se planteó realizarlo en otro lenguaje o con ciertos componentes y bibliotecas externas - como, por ejemplo, *RoboVM*[1], ahora mismo discontinuada -, que permitieran un desarrollo multiplataforma inmediato. Según el trabajo avanzaba, surgió la necesidad de desarrollar en código nativo del dispositivo - *C*, en el caso de *Android* -, para conseguir un rendimiento óptimo en el tratamiento de imagen en tiempo real. Esto hacía casi imposible la escritura de una aplicación totalmente multiplataforma, y dado que este es un proyecto acotado y realizado en solitario, se decidió que era más adecuado realizar la mejor aplicación posible centrada en un espectro de dispositivos más reducido..

9.2. Interfaz de usuario

Esta aplicación es, junto con la de generación y visualización, la otra que tiene interacción directa con el usuario. En este caso, la interfaz de usuario es mucho más sencilla, ya que necesita únicamente de acciones básicas para su funcionamiento. La aplicación muestra un diseño vertical para poder abarcar la mayor parte del test a la hora de ser escaneado.

Inicialmente, la aplicación abre con una pantalla *splash* - pantalla de inicio de una aplicación -, que muestra un logotipo discreto mientras ésta es cargada. En los dispositivos en los que se ha probado no dura más de unos segundos.

[1] `https://robovm.com/`

Figura 9.1: Aplicación de escaneo - Pantalla de inicio

Una vez abierta, se muestra en pantalla la interfaz básica de la aplicación.

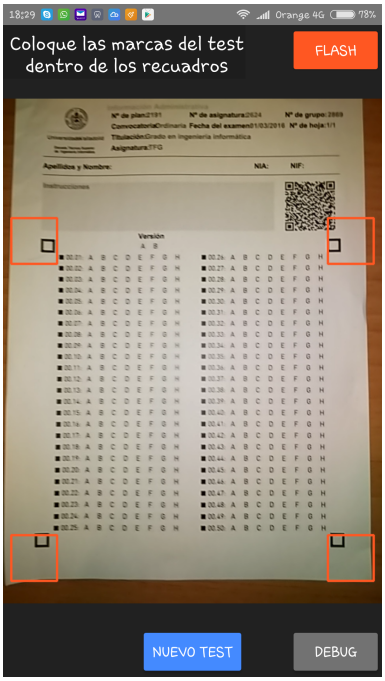

Figura 9.2: Aplicación de escaneo - Pantalla principal

Los componentes son los siguientes:

- **Instrucciones**. Indica brevemente como se ha de encuadrar el test para que sea reconocido.
- **Zona de escaneo**. Es la zona principal de la aplicación. En ella se muestra, en un recuadro proporcional a la

resolución de las imágenes de la cámara - *480x640* -, lo que esta viendo la cámara del dispositivo en tiempo real. Cerca de los bordes de la zona se muestran los *recuadros de referencia*, que son las zonas en las que el usuario deberá situar los recuadros de la *zona de test* del examen que desea escanear.

- **Botón *Flash***. Este botón permite activar y desactivar el flash del dispositivo para mejorar la iluminación en situaciones especialmente oscuras y donde el programa presente problemas para realizar el reconocimiento. Solo se mostrará si el dispositivo incluye dicha característica.
- **Botón *Nuevo test***. Este botón permite escanear un nuevo test si se desea escanear más de uno en la misma sesión o cancelar el escaneo actual.
- **Botón *Debug***. Muestra información interna sobre el escaneo. Se incluye para poder analizar más a fondo lo que ocurre durante la ejecución del algoritmo, pero e la hora de distribuir la aplicación sería removido.

Al ocurrir ciertos eventos - como el pulsar el botón de *Nuevo test* o al escanear satisfactoriamente un examen -, se muestran en pantalla diálogos de texto básicos que informan sobre lo que está realizando la aplicación en ese momento.

Si el test escaneado es de modo *Autocorrección*, una vez se reconozca y corrija se mostrará en pantalla una lista con los resultados obtenidos y el porcentaje total de aciertos.

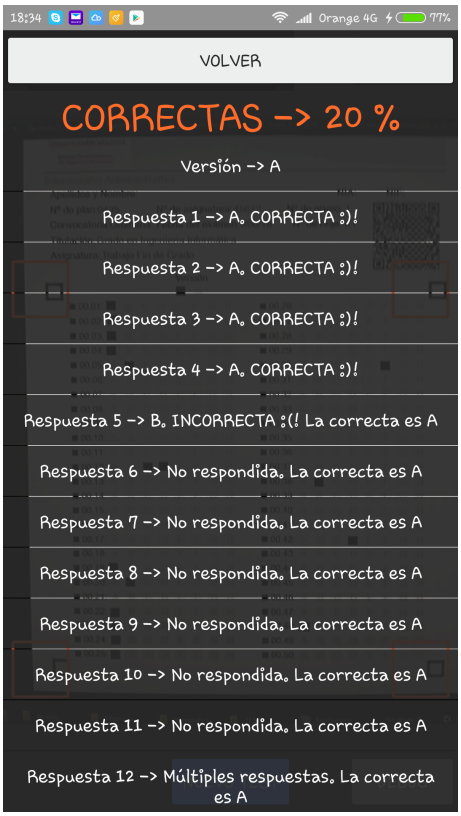

Figura 9.3: Aplicación de escaneo - Resultados

9.3. Método de escaneo

Antes de comenzar a hablar sobre la arquitectura de la aplicación, es conveniente realizar una explicación teórica sobre el funcionamiento del algoritmo de reconocimiento y corrección de exámenes. Aunque se indican ciertos datos pautados por las bibliotecas y técnicas utilizadas, los detalles sobre el código se mostrarán en la sección de *Implementación* de esta aplicación y la del servidor, ya que aquí se pretenda dar una imagen general del funcionamiento del algoritmo, que es similar en ambas implementaciones.

El algoritmo completo de escaneo se puede dividir en tres partes bien diferenciadas: *Reconocimiento* y *Corrección* o *Envío*. La primera de ellas se ejecuta siempre y en primer lugar, la segunda sólo se realiza en la aplicación móvil si el test es de tipo *Autocorrección* - en el servidor también se ejecuta cuando se le pide la corrección de uno o más tests - y la tercera se ejecuta si el test es de tipo *Oficial* - para enviar los datos al servidor -.

9.3.1. Reconocimiento

Efectuado en primer lugar, este proceso permite identificar si lo que se desea escanear es un test válido para el sistema y además encuentra las respuestas elegidas por el alumno. Se puede dividir a su vez en cuatro partes:

1. **Análisis del código QR**. Este proceso, al igual que otros indicados a continuación, se aprovecha tanto del uso del código *Java* de la aplicación - más sencillo y con acceso directo a ciertas bibliotecas externas y características de *Android* -, como del uso del código nativo en *C* - más rápido en su ejecución, especialmente en el tratamiento de imágenes en parte gracias al uso de la biblioteca *OpenCV* -. A partir de la imagen que capta en tiempo real la cámara del dispositivo se extrae desde el código en *C* la fracción de la imagen donde debería estar el código QR. Ésta se pasa entonces a la biblioteca de reconocimiento de códigos de barras *ZXing*, que se encarga de descifrar, si es posible, el mensaje incluído en el QR. Todo este proceso se repite continuamente hasta que es posible extraer el mensaje, y además dicho mensaje está bien formado y contiene la información adecuada dle test - esta información se ha descrito en secciones anteriores -.

2. **Validación de la *zona de test***. Con los datos del test recogidos tras el análisis del QR, se pasa a buscar las cuatro esquinas de la *zona de test*. Para ello, se aplica un filtro de binarización a la imagen para facilitar su búsqueda - esto se explica más detenidamente en el apartado de *Implementación* -. Con las esquinas encontradas se efectúa además una corrección de perspectiva para adaptar la información escaneada a las proporciones reales del examen. Este proceso se repite hasta que se encuentren las cuatro esquinas.

3. **Búsqueda de versión**. Con la *zona de test* encontrada y su perspectiva corregida, se pasa a comprobar si el alumno ha seleccionado una de als versiones del examen. Si el alumno no ha marcado ninguna versión, no se continuará el reconocimiento del test. por otra part, si el examen se ha diseñado con una única versión, este paso no se realizará.

4. **Búsqueda de respuestas**. En este último paso se buscan todas las preguntas del examen y, para cada una de ellas, se buscan las respuestas dadas por el alumno. Este paso se repite hasta que el algoritmo encuentre todas las preguntas del test, momento en el cual se daría el test por reconocido y terminaría el uso del código nativo en *C* y la librería *OpenCV*.

9.3.2. Corrección

Con los datos de reconocimiento obtenidos y pasados de nuevo al lado *Java* del código, si el test es de tipo *Autocorrección* se ejecuta este proceso. En primer lugar se realiza una comparación entre las respuestas elegidas por el alumno y las respuestas correctas del test encontradas en el código QR. Se pueden dar las siguientes posibilidades:

- **Respuesta correcta**. La contestación del alumno coincide con la respuesta correcta.
- **Respuesta incorrecta**. La contestación del alumno no coincide con la respuesta correcta.
- **Múltiples respuestas**. El alumno ha seleccionado más de una respuesta, por lo que se considera inválida.
- **Respuesta no encontrada**. No se ha encontrado respuesta.

Una vez corregido el test, se muestran por pantalla los resultados obtenidos en una lista y se calcula y enseña el porcentaje de aciertos.

9.3.3. Envío

Si el test se de tipo *Oficial*, no se realiza la corrección en el dispositivo. En cambio, se toma una foto resguardo del escaneo a una resolución de 768 píxeles de largo y 1024 de alto; mayor que los *480x640* del escaneo para guardar el resguardo con mayor detalle. Posteriormente, esta foto se envía al servidor junto a la imagen utilizada para el escaneo y el número de identificación del test y el código *NIA* del alumno encontrados en el código QR.

9.4. Arquitectura

De manera similar a la aplicación de generación y visualización, se han seguido las líneas básicas de un patrón *MVC*, dejando en este caso la *Vista* en su gran mayoría a los diseños de interfaz de usuario en *XML* que permite *Android* y con un *Modelo* muy reducido debido a que esta aplicación se puede considerar un intermediario hacia el servidor en el caso de los tests en modo *Oficial* o como una utilidad para el usuario en el caso de los de modo *Autocorrección*.

9.4.1. Diagrama de clases

Primeramente se muestra un diagrama de clases general en lenguaje *UML*.

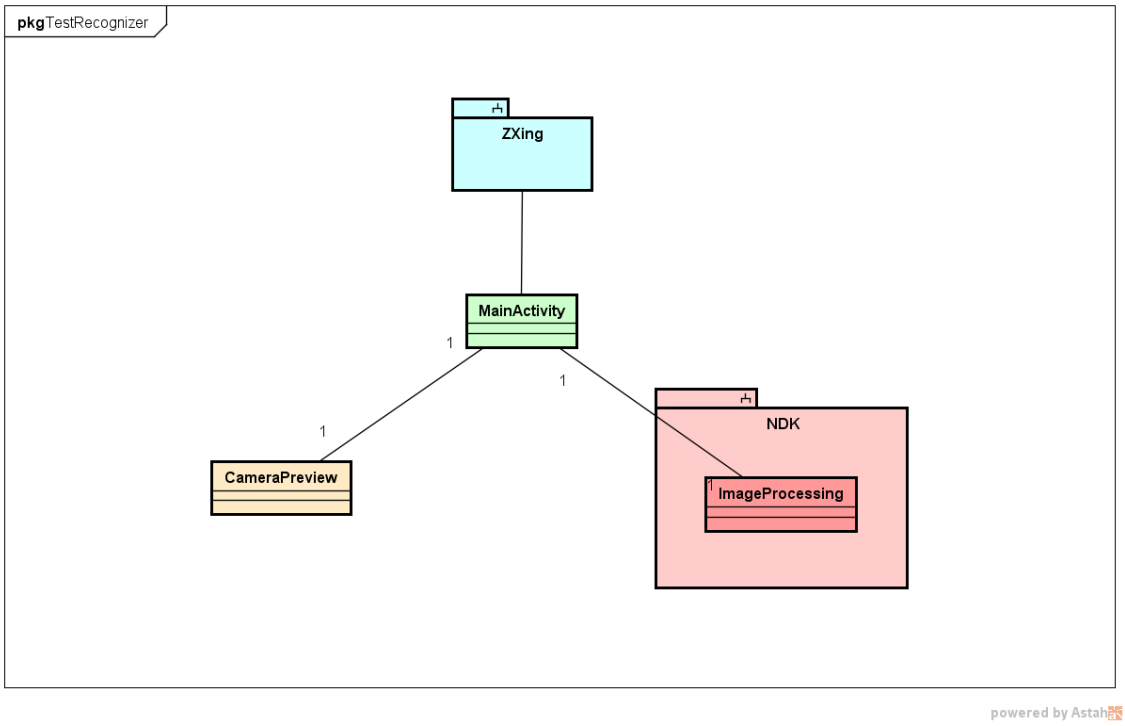

Figura 9.4: Aplicación de escaneo - Diagrama de clases general

Como se puede comprobar, el esquema de clases es mucho más sencillo que el de la aplicación anterior. Cabe destacar además el uso de la clase nativa *ImageProcessing*; se ha identificado dentro del sub-sistema *NDK* para denotar que es un clase concebida como un componente independiente en un lenguaje diferente al básico en *Android* y que es llamado a través del sistema interfaz que provee*NDK*.

Para analizar más a fondo cómo están compuestas las diferentes clases, a continuación se muestra un diagrama más detallado, derivado del anterior. Se presenta en formato apaisado y no incluye todos los parámetros para facilitar la legibilidad del diagrama.

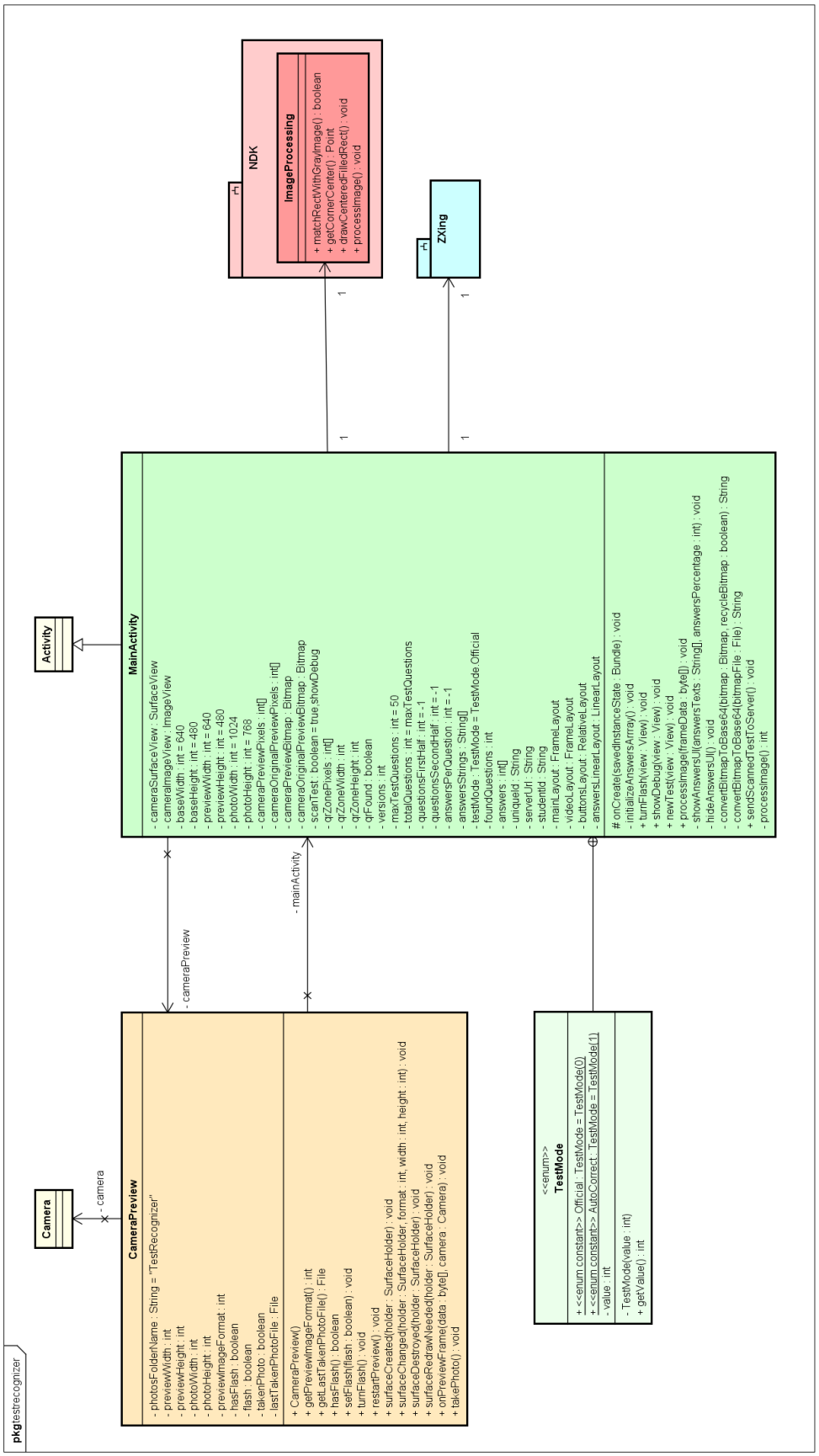

Figura 9.5: Aplicación de escaneo - Diagrama de clases general detallado

A continuación se explican los componentes mostrados:

- **MainActivity**. Hereda de la clase *Activity* de *Android*, y es la pantalla principal y única de la aplicación. Incluye la lógica primordial de este sistema y algunos métodos definidos para ser llamados al pulsar ciertos botones en la interfaz gráfica. También es la encargada de interactuar con el código nativo en *C* y de realizar parte del reconocimiento y corrección de los tests.
- **CameraPreview**. Es la clase encargada de conectar la aplicación con la cámara del dispositivo. Permite configurar y recoger en tiempo real la imagen que está viendo la cámara, realizar una foto a una determinada resolución - limitada por las características de la cámara -, y encender o apagar el flash - si el dispositivo dispone de él -.
- **ImageProcessing**. Es una clase escrita en código nativo *C*, que se compila y ejecuta mediante el uso del *NDK* y *JNI - Java Native Interface*[2] -. Incluye un método público que es llamado desde la clase *MainActivity* utilizando el interfaz puente que proporciona *JNI*. Utiliza la biblioteca *OpenCV*, escrita también en código nativo. Mediante su uso es posible ejecutar el algoritmo de reconocimiento y corrección explicado anteriormente. Al ser una clase en código nativo, no es directamente compatible con todos los tipos de arquitectura hardware disponibles en *Android*, así que es compilada en las principales arquitecturas disponibles tales como *x86*, *mips*, *armeabi-v7a* o *arm64-v8a*.
- ***ZXing***. Aunque es una biblioteca externa, se muestra en el diagrama como un sub-sistema para indicar la importancia de su uso en la aplicación. Se encarga de analizar y devolver el resultado del código QR.

9.5. Implementación

El código utilizado para implementar esta aplicación incluye algunos segmentos fundamentales en la concepción del sistema, así como otros que muestran un cierto interés por las técnicas utilizadas para adaptarse a ciertas características.

9.5.1. MainActivity

La primera porción interesante de código de esta clase es el análisis del código QR.

```
1  //Decode QR.
2  if(!qrFound) {
3    RGBLuminanceSource qrLuminanceSource = new RGBLuminanceSource(qrZoneWidth,
         qrZoneHeight, qrZonePixels);
4    BinaryBitmap qrBinaryBitmap = new BinaryBitmap(new
         HybridBinarizer(qrLuminanceSource));
5    String qrText = null;
6    try {
7      //Decode QR.
8      qrText = new QRCodeReader().decode(qrBinaryBitmap).getText();
9      //Mark QR as found.
10     qrFound = true;
```

En las líneas mostradas, se utiliza la imagen correspondiente al QR - *qrZondePixels* -, devuelta por el código nativo y se convierte y analiza mediante la biblioteca *ZXing*. El inicio del tratamiento de excepción mostrado termina más adelante recogiendo el posible fallo de reconocimiento del QR, en cuyo caso se seguiría buscando.

[2] https://en.wikipedia.org/wiki/Java_Native_Interface

Otro punto interesante es lo que ocurre una vez se reconoce correctamente el examen. Si es de tipo *Autocorrección*, se corrige el test y se muestra el resultado al usuario, y si es *Oficial*, se toma la foto resguardo y se envía al servidor. Cabe aclarar que en los textos informativos de esta sección del código se han suprimido los acentos para evitar problemas con dichos caracteres.

```
1  //Check found all questions.
2  if(foundQuestions == totalQuestions) {
3    //Stop test scanning.
4    scanTest = false;
5
6    //Check correction mode.
7    if(testMode == TestMode.AutoCorrect) {
8      //Check version.
9      int version = answers[0];
10     if(version > -1 && version < answersStrings.length) {
11       //Compare results to generate strings.
12       int correctAnswers = 0;
13       if(totalQuestions > answers.length - 1) totalQuestions = answers.length - 1;
14       String[] answersTexts = new String[totalQuestions + 1];
15       answersTexts[0] = "Version -> " + (char) (version + 65);
16       for(int i = 1;i < totalQuestions + 1;i++) {
17         //-2 -> Multiple answers.
18         //-1 -> Not recognized.
19         //0  -> Not answered.
20         //1..8 -> Answer.
21         if(answers[i] <= -2) {
22           answersTexts[i] = "Respuesta " + i + " -> Multiples respuestas. La
                 correcta es " + answersStrings[version].charAt(i - 1);
23         } else if(answers[i] == -1) {
24           answersTexts[i] = "Respuesta " + i + " -> No reconocida. La correcta
                 es " + answersStrings[version].charAt(i - 1);
25         } else if(answers[i] == 0) {
26           answersTexts[i] = "Respuesta " + i + " -> No respondida. La correcta
                 es " + answersStrings[version].charAt(i - 1);
27         } else if(answers[i] > 0 && answers[i] <= answersPerQuestion) {
28           answersTexts[i] = "Respuesta " + i + " -> " + (char) (answers[i] + 64);
29           if((int) answersStrings[version].charAt(i - 1) - 64 == answers[i]) {
30             answersTexts[i] += ". CORRECTA :)!";
31             correctAnswers++;
32           } else {
33             answersTexts[i] += ". INCORRECTA :(! La correcta es " +
                   answersStrings[version].charAt(i - 1);
34           }
35         } else {
36           answersTexts[i] = "Respuesta " + i + " -> La respuesta esta fuera del
                 limite. La correcta es " + answersStrings[version].charAt(i - 1);
37         }
38       }
39
40       //Show answers to user .
41       showAnswersUI(answersTexts,
42         (int) ((float) correctAnswers / (float) totalQuestions * 100));
43     } else {
44       //Show error as not correct version.
45       Toast.makeText(this, "Not correct version of number of answers",
             Toast.LENGTH_SHORT).show();
```

```
46      }
47    } else if(testMode == TestMode.Official) {
48      //Save photo and send to server (callback will be called).
49      cameraPreview.takePhoto();
50
51      //Show alert.
52      Toast.makeText(this, "Enviando datos al servidor...",
            Toast.LENGTH_SHORT).show();
53    }
54  }
```

En el caso del tipo *Autocorrección*, se comprueba si la versión seleccionada entra dentro de los límites de las versiones que tiene realmente el examen, y después se van comparando cada una de las respuestas del alumno con las correctas del test, añadiendo un texto informativo que será mostrado al usuario junto con el porcentaje de respuestas correctas al final.

En el caso *Oficial*, se pide a la instancia de la clase *CameraPreview* tomar una foto resguardo y se indica al usuario que se están enviando los datos al servidor. El método de envío de los datos es llamado desde la clase *CameraPreview* una vez se ha guardado la foto resguardo. Este método se encuentra en esta clase principal y es asíncrono, heredando de la clase *AsyncTask* de *Android*, ya que éste no permite realizar peticiones *HTTP* síncronas para evitar parar el flujo de ejecución de la interfaz de usuario. A continuación se muestra la sección del código de dicho método donde se realiza la petición *HTTP*.

```
1   //Encode to use POST json jersey.
2   String urlParameters =
3   "{\"uniqueId\":\"" + uniqueId + "\"," +
4   "\"studentId\":\"" + studentId + "\"," +
5   "\"base64Image\":\"" + photoBitmapBase64 + "\"," +
6   "\"base64Preview\":\"" + previewBitmapBase64 + "\"}";
7
8   byte[] postData = urlParameters.getBytes(StandardCharsets.UTF_8);
9
10  //Request id using server REST api.
11  java.net.URL validateTest = new URL(serverUrl + "/upload_scanned_test");
12  HttpURLConnection urlConnection = (HttpURLConnection)
        validateTest.openConnection();
13  urlConnection.setDoOutput(true);
14  urlConnection.setRequestMethod("POST");
15  urlConnection.setRequestProperty( "Content-Type", "application/json");
16  urlConnection.setRequestProperty( "charset", "utf-8");
17  urlConnection.setRequestProperty( "Content-Length",
        Integer.toString(postData.length));
18
19  //Send POST data.
20  DataOutputStream wr = null;
21  try {
22    wr = new DataOutputStream((urlConnection.getOutputStream()));
23    wr.write(postData);
24  } catch(Exception e) {
25    e.printStackTrace();
26    urlConnection.disconnect();
27    return "Error sending POST data to server.";
28  } finally {
29    //Close output stream.
30    if(wr != null) wr.close();
```

```
31    }
```

Tanto lo mostrado, como el código de recepción de la petición es muy similar al indicado para el envío de peticiones *REST* en la aplicación de generación y visualización.

Por último el código necesario para posteriormente llamar a la clase nativa *ImageProcessing* es el siguiente:

```
1  //Load native custom library.
2  static {
3    System.loadLibrary("ImageProcessing");
4  }
5
6  //Define process image C++ mehod.
7  private native int processImage(
8    int testMode,
9    int width, int height, byte[] frameData,
10   int[] outPixels, int[] outOriginalPixels,
11   int versions,
12   int questionsFirstHalf, int questionsSecondHalf, int answersPerQuestion,
13   int[] qrZonePixels, int qrZoneWidth, int qrZoneHeight,
14   int[] answers,
15   boolean scanTest,
16   boolean showDebug);
```

De manera estática - es decir, se llama una única vez al ejecutarse la aplicación -, se carga la biblioteca nativa *ImageProcessing*. Ésta incluye un único método que pueda ser llamado desde el código *Java*, y queda definido tal y como se muestra. En él, se pasan los datos básicos de la imagen capturada por la cámara, los datos del test contenidos en el QR, un parámetro de control parra indicar si se debe escanear el test - ya que el código realiza otras labores aprovechando la velocidad del código nativo y las características que otorga *OpenCV* -, y un parámetro para indicar si se debe mostrar información de depuración.

9.5.2. ImageProcessing

Siguiendo el método puente *Java* mostrado justo arriba, su implementación en el código nativo es la siguiente:

```
1  extern "C"
2  jint
3  Java_com_sea_testrecognizer_MainActivity_processImage(JNIEnv* env, jobject thiz,
4    jint testMode,
5    jint width, jint height, jbyteArray frameData,
6    jintArray outPixels, jintArray outOriginalPixels,
7    jint versions,
8    jint questionsFirstHalf, jint questionsSecondHalf, jint answersPerQuestion,
9    jintArray qrZonePixels, jint qrZoneWidth, jint qrZoneHeight,
10   jintArray answers,
11   jboolean scanTest,
12   jboolean showDebug) {
```

El nombrado del método sigue la convención definida por *JNI* para definir una función:

1. El nombre de la función debe empezar con *Java_*.

2. A continuación debe incluirse el nombre del paquete separado por '_' en vez de '.'.

3. Por último debe añadirse el nombre de la clase donde se ha definido el método nativo, seguido el nombre de dicho método y separados ambos por "_".

Se pueden encontrar tipos de parámetros propios de *JNI* - *jint*, *jintArray*, etc -, que son representaciones de los tipos propios de *Java*. Además se añade los parámetros *env* de tipo *JNIEnv** y *thiz* de tipo *jobject*. El primero es un puntero a una estructura que almacena todos los punteros a funciones *JNI*[3], y el segundo es una referencia al objeto que ha llamado esta función - en este caso, uno de tipo *MainActivity* -.

A lo largo de esta clase se realiza una implementación del algoritmo descrito anteriormente, utilizando varias de las características que da *OpenCV*. Inicialmente se intentó realizar enteramente en *Java* y sin el uso de bibliotecas externas, pero el manejo de imágenes en *Android* es especialmente lento y utiliza demasiada memoria, por lo que el paso a *C* y la ayuda otorgada por *OpenCV* fue fundamental.

La imagen dada por la previsualización de la cámara de *Android* está codificada en formato *YUV NV21*[4]. Éste es una variación del formato *YUV*, que codifica una imagen teniendo en cuenta su luminancia -*Y*- y dos componentes de crominancia - *U* y *V* -. *OpenCV* permite convertir y utilizar multitud de formatos, pero para facilitar la comprensión del algoritmo y el uso de diferentes métodos, se utilizan dos formatos:

- **Escala de grises**. Utilizado para los métodos que requieren imágenes sin color, y en general para la mayor parte de los procesos del algoritmo, ya que es más rápido y óptimo trabajar en dicho formato ya que lo que se busca en última instancia son sombras y manchas.

- **BGRA con 8 bits por canal**. Es el formato de color utilizado por la clase *Bitmap* en *Android*. Tiene los componentes de color en orden inverso al habitual *RGB*, y dedica el últiomo canal a la transparencia - *Alpha* -.

Otro detalle del formato *YUV NV21* es que la matriz de píxeles que componen la imagen contiene un único canal - en vez de los tres o cuatro habituales en las variantes de *RGB* o *RGBA* -, que ocupa el mismo ancho que la imagen pero 1.5 veces su altura.

OpenCV tiene una clase básica llamada *Mat*[5]; ésta representa un *array* de *n* dimensiones que puede incluir elementos de varios tipos. Entre otras cosas, se utiliza para almacenar las matrices de las imágenes.

Siguiendo todo lo descrito, la primera tarea importante a realizar en el método es incluir los datos de la imagen de la cámara recibidos desde *Java* en un objeto *Mat*.

```
1 //Get a pointer to first array element.
2 jbyte *pFrameData = env->GetByteArrayElements(frameData, 0);
3
4 //Create source matrix with size and format of camera preview (NV21).
5 //Usually YUV images are 1 channel images with 1.5*height (RGB with no alpha).
6 //CV_8UC1 means array of 8 bit single channel array.
7 Mat source(height + height/2, width, CV_8UC1, (uchar *) pFrameData);
```

[3] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html
[4] https://en.wikipedia.org/wiki/YUV
[5] http://docs.opencv.org/2.4/modules/core/doc/basic_structures.html

Como se muestra, primeramente se recoge un puntero al primer elemento de la matriz de la imagen original y posteriormente se crea el objeto *Mat*.

Posteriormente, se copia el *Mat* anterior a otro, convirtiendo el formato a *BGRA*.

```
1 //Convert from YUV NV21 (android camera default format) to color in general
     bitmap encoding.
2 Mat sourceColor;
3 cvtColor(source, sourceColor, CV_YUV2BGRA_NV21);
```

La imagen dada por *Android* está en formato apaisado, y para trabajar en el mismo espacio de posiciones que el usado al generar el test se debe realizar una rotación para dejar la imagen en vertical. Para ello, se crean nuevos objetos *Mat* y uno de ellos se convierte a formato de escala de grises para aplicar los posteriores filtros.

```
1 //Transpose and flip to rotate -90.
2 Mat sourceColorRot;
3 transpose(sourceColor, sourceColorRot);
4 flip(sourceColorRot, sourceColorRot, 1);
5
6 //Convert to grayscale in general encoding.
7 Mat sourceGrayRot;
8 cvtColor(sourceColorRot, sourceGrayRot, CV_BGRA2GRAY);
```

Con las imágenes base conseguidas, se aplica el primer filtro para realizar el reconocimiento del test y se clona en otro objeto *Mat* para aplicar posteriormente la corrección de perspectiva.

```
1 //Perform binarization using threshold function.
2 Mat sourceThreshold;
3 adaptiveThreshold(sourceGrayRot, sourceThreshold, 255,
     ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY, 11, 4);
4
5 //Create future warped matrix.
6 Mat warped = sourceThreshold.clone();
```

La técnica aplicada es una binarización[6], utilizando un filtro *Gaussiano*. Esta técnica se usa para separar puntos de interés de una imagen; en este caso, utilizando un umbral en la escala de grises y aplicando el filtro *Gaussiano* se consigue una imagen final con píxeles con valor negro o blanco puro. La matriz generada tras aplicar esta técnica se utilizará posteriormente para realizar una corrección de perspectiva, separar la *zona de test* y buscar las versiones, preguntas y respuestas del examen. *OpenCV* proporciona varias implementaciones de estas técnicas, como se puede comprobar en su documentación [18].

El umbral y filtro necesario para conseguir un buen resultado con esta técnica varía según el tipo de imagen - su composición, colores utilizados, etc -, y la iluminación presente; por lo que la configuración utilizada es un valor medio obtenido tras realizar pruebas para conseguir un escaneo de tests rápido y que produzca un porcentaje mínimo de *falsos positivos*. Los factores principales que se han tenido en cuenta en este caso es que se trabaja sobre un examen cuya *zona de test* ha sido diseñada en escala de grises, con un fondo blanco y con una iluminación habitual en una mesa de escritorio, que puede ser mejorada mediante el uso del flash si fuera necesario; además, se ha comprobado que estos valores son también válidos para escanear exámenes directamente desde una pantalla de ordenador con una calibración estándar.

[6] https://en.wikipedia.org/wiki/Binary_image

Obviando por un momento la imagen binarizada obtenida, se vuelve a utilizar la de escala de grises para buscar las cuatro esquinas de la *zona de test*. Para ello se aplica el algoritmo de *Canny* para la detección de bordes[7] - para este algoritmo *OpenCV* también proporciona una implementación[17] -. Éste utiliza primeramente un filtro *Gaussiano* - de manera similar a la binarización antes realizada -, para suavizar la imagen y, por tanto, el ruido derivado de la misma y que dificulta la detección de bordes. Sobre esta imagen filtrada, el algoritmo encuentra el gradiente de intensidad para detectar los bordes en dirección horizontal, vertical y diagonal, descarta los píxeles que no se consideran parte de los bordes - hasta dejar finas líneas candidatas a ser bordes reales -, y aplica dos umbrales sobre dichos resultados.

```
1  //Canny edge detection.
2  Mat edges;
3  Canny(sourceGrayRot, edges, 180, 200);
4
5  //Crop corner zone from last image.
6  Mat croppedCornerZoneTopLeft = edges(Rect(cornerZoneTopLeftX, cornerZoneTopLeftY,
       cornerZoneWidth, cornerZoneHeight));
7  Mat croppedCornerZoneTopRight = edges(Rect(cornerZoneTopRightX,
       cornerZoneTopRightY, cornerZoneWidth, cornerZoneHeight));
8  Mat croppedCornerZoneBottomRight = edges(Rect(cornerZoneBottomRightX,
       cornerZoneBottomRightY, cornerZoneWidth, cornerZoneHeight));
9  Mat croppedCornerZoneBottomLeft = edges(Rect(cornerZoneBottomLeftX,
       cornerZoneBottomLeftY, cornerZoneWidth, cornerZoneHeight));
10
11 //Calculate corner centers.
12 Point cornerTopLeftCentroid = getCornerCenter(croppedCornerZoneTopLeft);
13 Point cornerTopRightCentroid = getCornerCenter(croppedCornerZoneTopRight);
14 Point cornerBottomRightCentroid = getCornerCenter(croppedCornerZoneBottomRight);
15 Point cornerBottomLeftCentroid = getCornerCenter(croppedCornerZoneBottomLeft);
16
17 //Check if really found corners.
18 if(cornerTopLeftCentroid.x >= 0 && cornerTopLeftCentroid.y >= 0 &&
       cornerTopRightCentroid.x >= 0 && cornerTopRightCentroid.y >= 0
19    && cornerBottomRightCentroid.x >= 0 && cornerBottomRightCentroid.y >= 0 &&
          cornerBottomLeftCentroid.x >= 0 && cornerBottomLeftCentroid.y >= 0) {
```

Sobre la nueva matriz generada tras aplicar el algoritmo de *Canny*, se consiguen pequeñas matrices en las posiciones donde se encuentran las *marcas de encuadre* donde el usuario debe situar las esquinas de la *zona de test* al escanear el examen con la cámara. A partir de estas nuevas cuatro matrices, se buscan los centroides de cada recuadro, y si se detectan todos, la *zona de test* ha sido reconocida y se puede continuar con el escaneo de las versiones, preguntas y respuestas. En caso de que no se encuentren todos los bordes, el proceso se seguiría repitiendo con cada imagen en tiempo real de la cámara hasta encontrarlos.

Antes de continuar con la búsqueda de versiones, preguntas y respuestas, cabe mostrar el método *getCornerCenter* llamado en la sección anterior de código, y es que este método se encuentra también en la clase *ImageProcessing*.

```
1  //Return point with coner detected center in mat.
2  Point getCornerCenter(Mat &cornerMat) {
3    //Find contours (two nested contours).
4    vector<vector<Point> > contours;
5    vector<Vec4i> hierarchy;
```

[7] https://en.wikipedia.org/wiki/Canny_edge_detector

```
6     findContours(cornerMat, contours, hierarchy, CV_RETR_TREE,
          CV_CHAIN_APPROX_SIMPLE);
7
8     //Check corners found.
9     if (contours.size() > 0 && hierarchy.size() >= 4) {
10      //Calculate moments by contours.
11      int index = contours.size() > 1 ? 1 : 0;
12      Moments mom;
13      //Get contours until one is correct.
14      do {
15        mom = moments((vector<Point>) contours[index]);
16        index++;
17      } while(mom.m00 == 0.0 && index < contours.size() - 1);
18
19      //Find centroid by moments.
20      Point centroid = Point(mom.m10 / mom.m00, mom.m01 / mom.m00);
21      //Return centroid.
22      return centroid;
23    } else {
24      //Not detected, return 'null' point.
25      return Point(-1, -1);
26    }
27  }
```

Este método devuelve los centroides detectados en los recuadros, o un punto considerado como nulo si no es posible encontrarlos. Una de las peculiaridades del diseño de los recuadros - basada en las esquinas del código QR -, es que son bordes de un cuadrado. Este planteamiento permite evitar gran cantidad de los *falsos positivos* durante el reconocimiento. Mediante el uso de *OpenCV*, se aplica aquí una búsqueda de contornos[8] sobre la matriz dedicada a cada esquina - cabe recordar que estas matrices parten de otra que se le ha aplicado el algoritmo de *Canny* -. Como el diseño es el del borde de un recuadro, el algoritmo tendrá detectar cuatro contornos para asegurarse que realmente es una esquina, lo cual es difícil de conseguir en una imagen que no contenga realmente uno de los recuadros válidos; si además, se suma que esto tiene que ocurrir a la vez con las cuatro esquinas de la *zona de test*, se puede notar que el número de *falsos positivos* desciende en gran medida.

Si se encuentran los cuatro contornos, se utilizan los momentos de los mismos para calcular el centroide de la esquina.

En este punto puede surgir la cuestión de que, aunque buscar las cuatro esquinas con el método indicado reduce los errores, también aumenta el tiempo de ejecución del algoritmo, y para definir un rectángulo solo es necesario conocer dos esquinas de una diagonal. Por una parte, tras realizar distintas pruebas y con el avance del proyecto se pudo comprobar que el tiempo añadido no era determinante, y por otra surgió la necesidad de utilizar una técnica de corrección de perspectiva que requería conocer el centro de las cuatro esquinas.

Los cálculos efectuados para buscar las versiones, preguntas y respuestas del test se realizan en base a las medidas originales del test aplicadas de manera proporcional sobre una *zona de test* escaneada que tiene unas dimensiones de 420 píxeles de ancho por 421 de alto - su tamaño proporcional en la imagen de la cámara con una resolución de *480x6480* -. Teniendo en cuenta que el usuario que escanea el test casi nunca va a conseguir colocar la cámara del dispositivo de forma exactamente paralela al test, prácticamente siempre va a existir algún tipo de distorsión en la imagen, ya sea en forma de rotación en alguno de los tres ejes o por parte de factores ambientales como la iluminación. Para minimizar esto y además agilizar la búsqueda del *fotograma perfecto* en el escaneo en tiempo real, se aplica un corrección de perspectiva.

[8] `http://docs.opencv.org/2.4/doc/tutorials/imgproc/shapedescriptors/find_contours/find_contours.html`

Este tipo de correcciones también están implementadas en *OpenCV*[9], y la utilizada en este caso - la llamada *warpPerspective* -, requiere de cuatro puntos de referencia. Estos puntos son los centros de las cuatro esquinas de la *zona de test*, y a continuación se muestra su uso en el código:

```
1  //Set perspective warp points matrix.
2  Point2f sourcePoints[4];
3  sourcePoints[0] = Point2f(cornerTopLeftCentroid.x + cornerZoneTopLeftX,
       cornerTopLeftCentroid.y + cornerZoneTopLeftY);
4  sourcePoints[1] = Point2f(cornerTopRightCentroid.x + cornerZoneTopRightX,
       cornerTopRightCentroid.y + cornerZoneTopRightY);
5  sourcePoints[2] = Point2f(cornerBottomRightCentroid.x + cornerZoneBottomRightX,
       cornerBottomRightCentroid.y + cornerZoneBottomRightY);
6  sourcePoints[3] = Point2f(cornerBottomLeftCentroid.x + cornerZoneBottomLeftX,
       cornerBottomLeftCentroid.y + cornerZoneBottomLeftY);
7  Point2f finalPoints[4];
8  finalPoints[0] = Point2f(testZoneTopLeftX + cornerWidth * 0.5f, testZoneTopLeftY
       + cornerHeight * 0.5f);
9  finalPoints[1] = Point2f(testZoneTopLeftX + testZoneWidth - cornerWidth * 0.5f,
       testZoneTopLeftY + cornerHeight * 0.5f);
10 finalPoints[2] = Point2f(testZoneTopLeftX + testZoneWidth - cornerWidth * 0.5f,
       testZoneTopLeftY + testZoneHeight - cornerHeight * 0.5f);
11 finalPoints[3] = Point2f(testZoneTopLeftX + cornerWidth * 0.5f, testZoneTopLeftY
       + testZoneHeight - cornerHeight * 0.5f);
12
13 //Calculate perspective transform.
14 Mat warpMat = getPerspectiveTransform(sourcePoints, finalPoints);
15
16 //Change perspective.
17 warpPerspective(sourceThreshold, warped, warpMat, warped.size());
18
19 //Crop test zone.
20 Mat croppedTest = warped(Rect(testZoneTopLeftX, testZoneTopLeftY, testZoneWidth,
       testZoneHeight));
```

Utilizando la imagen binarizada, los centroides de las cuatro esquinas detectadas y las medidas en base a los datos reales del test, se aplica una corrección de perspectiva que corrige las distorsiones de la imagen con una pérdida de calidad mínima; y más teniendo en cuenta que lo aquí de analiza son píxeles blancos o negros en *manchas* con un tamaño mínimo. En base a la imagen corregida y con las medidas reales del test, se crea una nueva matriz con la *zona de test* aislada.

A continuación se detecta si el alumno ha seleccionado una de las versiones del examen. Si el test incluye una única versión, este punto se pasa, y en caso contrario se buscan recuadros coloreados desde la zona relativa a la primera versión hasta el número de versiones del test. Siempre se busca en todos los recuadros para detectar si el alumno ha seleccionado más de una versión, en cuyo caso el test no será válido y no se escaneará.

```
1  //Detect version (if more than one).
2  int versionX = firstVersionTopLeftX;
3  int versionsDetected = 0;
4  int currentVersion = -1;
5  if(versions > 1) {
6    for(int version = 0;version < versions;version++) {
7      //If version detected, save and check questions and answers.
```

[9] http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html

```
8       if(matchRectWithGrayImage(versionWidth, versionHeight, croppedTest,
9         20, firstVersionTopLeftY, versionX)) {
10        versionsDetected++;
11        currentVersion = version;
12      }
13      //Increase versions positions.
14      versionX += versionOffset;
15    }
16  } else if(versions == 1) {
17    versionsDetected = 1;
18    currentVersion = 0;
19  }
20
21  //Only continue if correct version detection (or just one version).
22  if(versionsDetected == 1 && currentVersion > -1) {
```

El método utilizado para detectar cada recuadro es *matchRectWithGrayImage*. Al igual que el utiliado para buscar los centroides, es un método hecho para esta clase que se dedica a buscar rectángulos con píxeles dentro de un umbral de grises en una zona determinada.

```
1   //Custom match template method that receives a BLACK rectangle (just width and
        height values) and base GRAY image.
2   bool matchRectWithGrayImage(int templateWidth, int templateHeight, Mat &imageMat,
        int matchMinAcceptancePercentage,
3     int initialTopLeftI = 0, int initialTopLeftJ = 0, uchar equalityMargin = 10) {
4     //Color values.
5     uchar imageValue;
6     //Encountered matches.
7     int matches = 0;
8     //Iterate template matrix.
9     for (int i = 0;i < templateHeight;++i) {
10      //Get pointer to first row values.
11      uchar* imageFirstRowValue = imageMat.ptr<uchar>(i + initialTopLeftI);
12      //Iterate columns.
13      for (int j = 0;j < templateWidth;++j) {
14        //Get image gray data (1 channel), then increase.
15        imageValue = imageFirstRowValue[j + initialTopLeftJ];
16        //Check if 'equal' gray value.
17        if (0 >= imageValue - equalityMargin && 0 <= imageValue + equalityMargin) {
18          matches++;
19        }
20      }
21    }
22    //Convert percentage.
23    matchMinAcceptancePercentage = (templateWidth * templateHeight *
          matchMinAcceptancePercentage) / 100;
24
25    //Return result.
26    return matches >= matchMinAcceptancePercentage;
27  }
```

Este método itera los valores de una matriz de una imagen en escala de grises a partir de una posición dada y compara si el valor del píxel situado en esa posición está dentro de un umbral dado. La función está dedicada únicamente a imágenes en escala de grises porque es como se analiza la imagen en este algoritmo y porque el

formato de color de la matriz define qué número de filas y columnas la componen; de tal manera que matrices con otro formato requerirían otro tipo de iteración.

Una vez detectada la versión y realizados algunos ajustes básicos de configuración, se pasan a detectar las preguntas y respuestas del test. El método es muy similar al seguido para reconocer las versiones del examen. Se recorren las preguntas y respuestas en base a los datos del test y se utiliza el método *matchRectWithGrayImage* para comprobar si existe un recuadro pintado. Se van guardando las preguntas y respuestas detectadas, las últimas siguiendo el siguiente código:

- **Respuesta detectada**. Número de respuesta del 1 al 8.
- **Respuesta no detectada**. Número 0.
- **Múltiples respuestas**. Número -2.

Los valores detectados serán devueltos al código *Java* para determinar si el test ha sido o no reconocido.

```
1  //Detect questions.
2  for (int column = 0, index = 0;column < 2;column++) {
3    //Iterate questions.
4    for (int question = 0;question < (column == 0 ? questionsFirstHalf :
         questionsSecondHalf);question++,index++) {
5      //Initialize result as not recognized.
6      results[index] = -1;
7      //Search question point.
8      questionDetected = matchRectWithGrayImage(questionWidth, questionHeight,
           croppedTest,
9        30,
10       questionY,
11       questionX);
12     //Detect answers.
13     if (questionDetected) {
14       //As detected, set as not answered.
15       foundQuestions++;
16       results[index] = 0;
17       //Reset detected answers.
18       answersDetected = 0;
19       //Set answer positions.
20       answerX = questionX + firstAnswerFromQuestionTopLeftX;
21       answerY = questionY + firstAnswerFromQuestionTopLeftY;
22       //Iterate all answers.
23       for (int answer = 0;answer < answersPerQuestion;answer++) {
24         //If answers detected, increase.
25         if(matchRectWithGrayImage(answerWidth, answerHeight, croppedTest,
26           20, answerY, answerX)) {
27           answersDetected++;
28           answerLastIndex = answer;
29         }
30         //Increase answer positions.
31         answerX += answersOffsetX;
32       }
33       //Add detected answer if is autocorrect mode (1).
34       if(testMode == 1) {
35         //Check if found only one answers.
36         if(answersDetected == 1) {
```

```
            results[index] = answerLastIndex + 1;
          } else if(answersDetected > 1) {
            results[index] = -2;
          }
        }
      }
      //Increase question Y.
      questionY += questionsOffsetY;
    }
    //Change question columns and reset.
    if (column == 0) {
      questionX = firstQuestionSecondColumnTopLeftX;
      questionY = firstQuestionTopLeftY;
    }
}
```

En caso de que el examen sea de tipo *Autocorrección*, las respuestas detectadas se copian a un *array* definido en el código *Java* y pasado como parámetro al método de reconocimiento.

```
//Copy results if autocorrect mode (1).
if(testMode == 1) {
   jint * pAnswers = env->GetIntArrayElements(answers, 0);
   pAnswers[0] = currentVersion;
   for (int i = 0;i < questionsFirstHalf + questionsSecondHalf;i++) {
     pAnswers[i + 1] = results[i];
   }
   env->ReleaseIntArrayElements(answers, pAnswers, 0);
}
```

Con lo ya descrito, el algoritmo utilizado para escanear el test queda completamente descrito. Antes de terminar esta sección, se pueden destacar dos puntos más de interés ejecutados en el código nativo. Aprovechando el método, se realiza aquí también el recorte de la zona correspondiente al QR que será posteriormente pasada a la biblioteca *ZXing* desde el código *Java*. Esto se realiza en la siguiente porción de código:

```
//Crop QR zone and pass to java android bitmap array.
jint * pQrZonePixels = env->GetIntArrayElements(qrZonePixels, 0);
Mat sourceCroppedQr = sourceColorRot(Rect(qrZoneX, qrZoneY, qrZoneWidth,
    qrZoneHeight));
Mat outQr(qrZoneHeight, qrZoneWidth, CV_8UC4, (uchar *) pQrZonePixels);
Mat testQr;
cvtColor(sourceCroppedQr, testQr, CV_BGRA2RGBA);
cvtColor(sourceCroppedQr, outQr, CV_RGBA2BGRA);
env->ReleaseIntArrayElements(qrZonePixels, pQrZonePixels, 0);
```

Las técnicas aquí utilizadas son las mismas que en puntos anteriores para crear una nueva matriz a partir de una sección de otra y para realizar cambios de codificación de los píxeles.

El último segmento destacable surge de la misma filosofía de aprovechamiento que el anterior. Como la imagen mostrada en pantalla al usuario durante el escaneo pasa siempre por el código nativo, las *zonas de encuadre* de las esquinas de la *zona de test* se dibuja en esta función utilizando el método mostrado a continuación:

```
//Draw an OpenCV rectangle with center in 'center' and color 'color'.
void drawCenteredFilledRect(Mat &mat, int centerX, int centerY, int width, int
    height,
   Scalar color, int thickness = 5, int lineType = CV_FILLED) {
   rectangle(mat,
     Point(centerX - width / 2, centerY - height / 2),
     Point(centerX + width / 2, centerY + height / 2),
     color,
     thickness,
     lineType);
}
```

Este método adapta la función *rectangle*[10] de *OpenCV* para dibujar un recuadro con ciertos parámetros centrado en una posición y con un color específico. El color utilizado finalmente en el programa es el rojo, y el realizar el dibujado de estos recuadros en tiempo real en el código nativo permitió poder cambiar dinámicamente las zonas donde se buscaban las esquinas de la *zona de test* hasta que se consiguieron unas medidas adecuadas.

[10] http://docs.opencv.org/2.4/modules/core/doc/drawing_functions.html

Capítulo 10

Aplicación servidor

Llegados a este punto, se pasa a detallar la última aplicación principal que compone el sistema. Como se ha indicado en otros apartados, el servidor principal contiene a su vez un servidor de bases de datos y otro de repositorio; y éstos pueden estar embebidos o figurar como servidores externos. En cualquier caso aquí se trata al software como un todo, indicando su diseño general y peculiaridades.

10.1. Introducción

Esta es la única aplicación que no tiene interacción directa con el usuario, pero se conecta con las otras dos para almacenar y corregir los exámenes de tipo *Oficial*. De igual manera que el software de generación y visualización, está escrita en lenguaje *Java* y utilizando *Netbeans* como entorno de desarrollo. Como servidor se utiliza *GlassFish* en su versión *4.1*.

El punto más complicado en el desarrollo de esta aplicación fue la integración del algoritmo de reconocimiento y corrección que utiliza *OpenCV*. En un principio se intentó reutilizar casi completamente el código mostrado en el capítulo anterior, pero surgieron varios problemas a la hora de implementar código nativo en el servidor. Al estar trabajando en este caso en equipos de escritorio, no era necesario ni se disponía del sistema *NDK*; por una parte esto permitía compilar el código en *C* o *C++* libremente en cualquier entorno de desarrollo o compiladrr disponible para luego enlazar la biblioteca generada en el proyecto *Java* y acceder a ella utilizando el interfaz *JNI*, pero finalmente la falta de cohesión que aportaba el *NDK* hizo que esta alternativa provocara multitud de problemas y enlenteciera el progreso general.

Buscando una alternativa válida, se decidió utilizar la implementación oficial de *OpenCV* en *Java*[20]. Ésta es básicamente una interfaz puente entre las funciones en *C* y *C++* de la biblioteca original de *OpenCV* y el código *Java*. También utiliza *JNI*, y la mayor pparte de clases y funciones utilizan los mismos parámetros y tienen el mismo comportamiento, estando adaptadas a las diferentes características de *Java* como la ausencia de punteros o destructores explícitos.

Al ser un acceso a la biblioteca, requiere que ésta esté compilada y disponible en el proyecto. Para ello se compiló la librería *OpenCV* en su versión *3.10* para la plataforma en la que se estaba desarrollando el proyecto, *Windows 10 x64*, gracias al uso de *MinGW*[1] - *Minimalist GNU for Windows* -. Esto otorgó una biblioteca *DLL*[2] - *Dynamic-link library* -.

Inicialmente se intentó colocar esta librería en el directorio del proyecto, pero a la hora de cargarla desde el

[1] `http://www.mingw.org/`
[2] `https://en.wikipedia.org/wiki/Dynamic-link_library`

código *Java* nunca era encontrada. Tras analizar e investigar el caso, se encontró que el uso de bibliotecas nativas en el servidor suele provocar problemas de ese estilo, y tras varias pruebas se llegó a la conclusión que la biblioteca debía colocarse en la carpeta donde se encuentran las bibliotecas generales del sistema; es decir, en el caso de *Windows 10*, en la ruta *Windows/System32*.

10.2. Interfaz *REST*

El servidos dispone de una interfaz *REST* a la que acceden las otras dos aplicaciones del sistema. Se decidió utilizar este sistema sobre otro tipo de servicios web porque provee un interfaz sólido, de diseño simple y tiene una buena integración en *Netbeans* mediante el uso de *JAX-RS*[3] *- Java API fot RESTful Web Services* -, que da soporte en *Java* para la arquitectura *REST*.

El software dispone de los siguientes métodos *REST*:

- ***get_id***. El método recibe un objeto de tipo *Test*, que será explicado más adelante. Para generar este objeto lo que se envía realmente al servidor es un texto en formato *JSON* con los parámetro de la clase a generar, y con ello *JAX-RS* se encarga de generar la clase. La función, a partir de los datos del test enviado, genera una identificación única para él y lo almacena en la base de datos. Al final, la identificación generada se retorna como respuesta.
- ***upload_scanned_test***. Con la misma técnica que el anterior, este método recibe un objeto de tipo *Scanned-Test* con la información de un examen *Oficial* escaneado por la aplicación para dispositivos móviles. A partir de los datos del test, busca en la base de datos el examen y con toda la información guarda las imágenes de resguardo y escaneo en el repositorio y posteriormente almacena nueva información en la base de datos.
- ***get_all_tests***. Retorna, como un texto en formato *JSON*, todos los exámenes almacenados en la base de datos, recogiendo tanto la información obtenida en la generación del mismo como cuando ha sido escaneado - si el escaneo se ha realizado -.
- ***correct_test***. Recibe un texto en formato *JSON* con los datos de un test que el usuario desea corregir. A partir de ellos, la aplicación busca el test en la base de datos, recoge la imagen de análisis del repositorio, y realiza una llamada a la clase que contiene el algoritmo de reconocimiento y corrección para corregir el examen. Una vez terminada la corrección, sus datos se almacenan en la base de datos y se retornan al final de la ejecución de la función.
- ***correct_remaining_tests***. Método que utiliza internamente al anterior. Recoge todos los exámenes no corregidos de la base de datos - si existe alguno - y los manda corregir uno a uno al método anterior. Cuando termina, devuelve el número de tests corregidos.
- ***get_repository_image***. Recibe como parámetro un texto en formato *JSON* con la ruta del repositorio donde se encuentra almacenada una imagen. Ésta puede ser tanto la imagen de análisis como la foto resguardo. Una vez recogida del respositorio, ésta se codifica en una cadena en formato *Base64* y se retorna.

10.3. Almacenado en la base de datos

La base de datos *SQL* consta de dos tablas, una para almacenar los datos de los test generados y otra para los tests escaneados. Ambas tablas están enlazadas por el identificador único del examen. Su definición es la siguiente:

[3] `https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services`

- ***tests***. Es la tabla que incluye los test generados.
  - ***plan***. Campo de tipo *VARCHAR*, con una longitud máxima de 50 caracteres, que indica el número de plan.
  - ***subject***. Campo de tipo *VARCHAR*, con una longitud máxima de 50 caracteres, que indica el número de asignatura.
  - ***year***. Campo de tipo *VARCHAR*, con una longitud máxima de 50 caracteres, que indica el año.
  - ***call***. Campo de tipo *VARCHAR*, con una longitud máxima de 50 caracteres, que indica el nombre de la convocatoria.
  - ***questions_first_half***. Campo de tipo *INT*, que indica el número de preguntas en la primera mitad del test.
  - ***questions_second_half***. Campo de tipo *INT*, que indica el número de preguntas en la segunda mitad del test.
  - ***answers_per_question***. Campo de tipo *INT*, que indica el número de respuestas por pregunta.
  - ***versions***. Campo de tipo *INT*, que indica el número de versiones.
  - ***answers***. Campo de tipo *VARCHAR*, con una longitud máxima de 500 caracteres, que recoge las respuestas del test una a continuación de otra y separadas entre versiones por el caracter "-".
  - ***id***. Campo de tipo *VARCHAR*, con una longitud máxima de 50 caracteres, que indica el número de identificación concreto del test - incluido por si se suben dos exámenes de similares características -.
  - ***complete_id*** .Campo de tipo *VARCHAR*, con una longitud máxima de 200 caracteres, que indica el código de identificación completo del test. Ésta es el utilizado por el resto de aplicaciones para identificar inequívocamente un examen y está compuesto por los valores de *plan*, *subject*, *year*, *call*, e *id*, separados entre ellos por el caracter ".". Este valor es el que se retorna en el método *REST* ***get_id*** descrito anteriormente. Además, es la clave primaria de la tabla.
- ***tests_scanned***. Es la tabla que incluye los test generados.
  - ***student_id***. Campo de tipo *VARCHAR*, con una longitud máxima de 200 caracteres, que indica el código *NIA* del alumno.
  - ***test_id***. Campo de tipo *VARCHAR*, con una longitud máxima de 200 caracteres, que indica el número de identificación completo del test. Además, incluye una clave foránea que lo enlaza con el campo ***complete_id*** de la tabla *tests*; de esta manera se enlazan los datos del test escaneado con los de generación.
  - ***photo_url***. Campo de tipo *VARCHAR*, con una longitud máxima de 1000 caracteres, que recoge la ruta del repositorio para acceder a la foto resguardo.
  - ***preview_url***. Campo de tipo *VARCHAR*, con una longitud máxima de 1000 caracteres, que recoge la ruta del repositorio para acceder a la imagen de análisis.
  - ***student_answers***. Campo de tipo *VARCHAR*, con una longitud máxima de 500 caracteres, que recoge las respuestas del alumno.
  - ***student_version***. Campo de tipo *INT* que determina la versión del test seleccionada por el alumno.
  - ***corrected***. Campo de tipo *BIT* que sirve como un *booleano* para identificar rápidamente si el test está o no corregido.

Las sentencias *SQL* utilizadas para crear ambas tablas son las que se muestran a continuación:

- ***tests***.

```
CREATE TABLE `tests` (
   `plan` VARCHAR(50) NOT NULL,
   `subject` VARCHAR(50) NOT NULL,
   `year` VARCHAR(50) NOT NULL,
   `call` VARCHAR(50) NOT NULL,
   `questions_first_half` INT(10) UNSIGNED NOT NULL,
   `questions_second_half` INT(10) UNSIGNED NOT NULL,
   `answers_per_question` INT(10) UNSIGNED NOT NULL,
   `versions` INT(10) UNSIGNED NOT NULL,
   `answers` VARCHAR(500) NOT NULL,
   `id` VARCHAR(50) NOT NULL,
   `complete_id` VARCHAR(200) NOT NULL,
   PRIMARY KEY (`complete_id`)
)
COLLATE='utf8_general_ci'
ENGINE=InnoDB
;
```

- ***tests_scanned***.

```
CREATE TABLE `tests_scanned` (
   `student_id` VARCHAR(200) NOT NULL,
   `test_id` VARCHAR(200) NOT NULL,
   `photo_url` VARCHAR(1000) NOT NULL,
   `preview_url` VARCHAR(1000) NOT NULL,
   `student_answers` VARCHAR(500) NOT NULL,
   `student_version` INT(10) NOT NULL,
   `corrected` BIT(1) NOT NULL,
   INDEX `foreign_test_id` (`test_id`),
   CONSTRAINT `foreign_test_id` FOREIGN KEY (`test_id`) REFERENCES `tests`
      (`complete_id`) ON UPDATE CASCADE ON DELETE CASCADE
)
COLLATE='utf8_general_ci'
ENGINE=InnoDB
;
```

Para el manejo de la base de datos externa, se ha utilizado en gran medida el programa *HeidiSQL*[4], que permite un manejo sencillo y con interfaz de usuario para base de datos *SQL*. En cuanto a la versión con el servidor embebido, se genera en base a lo experimentado y realizado con esta aplicación en la base de datos externa.

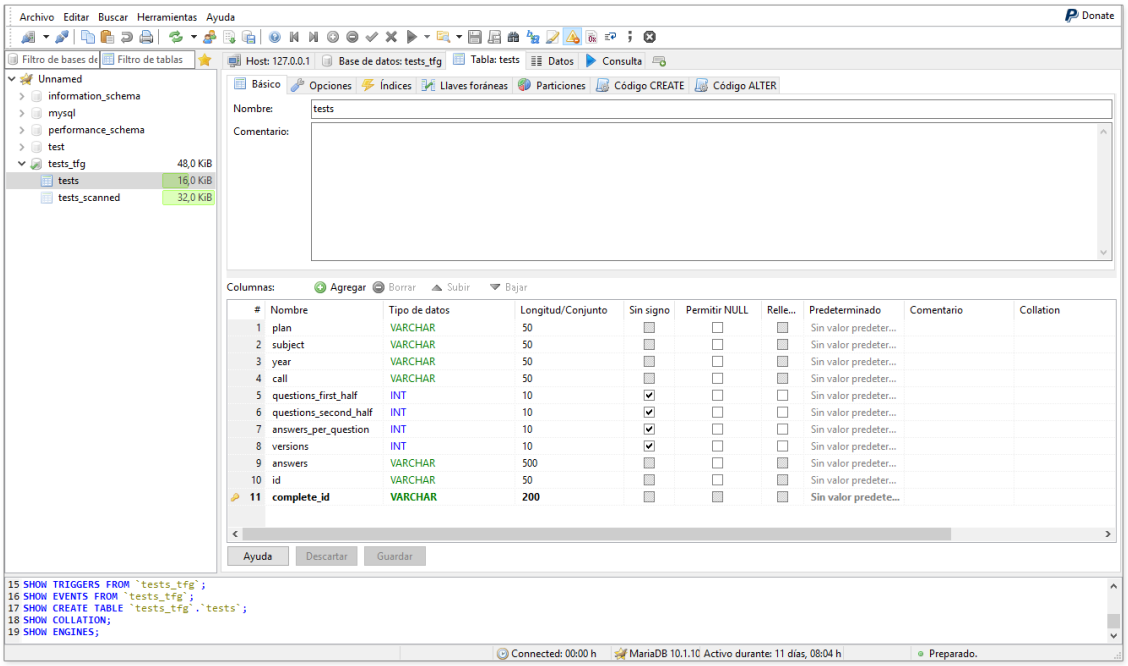

Figura 10.1: Aplicación servidor - *HeidiSQL*

10.4. Almacenado de las fotos en el repositorio

El repositorio *Jackrabbit* se utiliza para almacenar la foto de resguardo y la imagen de análisis de cada test *Oficial* escaneado en el sistema. Debido a su tarea, la jerarquía de directorios es sencilla y sigue las mismas directrices que la base de datos, ya que la dirección de nodos se puede conseguir a partir del número de identificación completo del test y el código *NIA* del alumno que escaneó el examen.

Los nodos que componen el repositorio, de más general a más concreto, son los siguientes:

1. **Nodo raíz**. Es el nodo base del repositorio.
2. **Nodo *tests***. Es el nodo padre de los tests almacenados.
3. **Nodo de plan**. Nodo organizado por el número de plan.
4. **Nodo de asignatura**. Nodo organizado por el número de asignatura.
5. **Nodo de año**. Nodo organizado por el año.
6. **Nodo de convocatoria**. Nodo organizado por el nombre de la convocatoria.
7. **Nodo de número de identificación de test**. Nodo organizado por el número de identificación del test generado al almacenar un nuevo examen en la base de datos.
8. **Nodos de imágenes**. En este punto y al mismo nivel se incluyen los nodos de las imágenes. Se nombran con el código *NIA* del alumno seguido de "*_photo*"si es la foto resguardo o "*_preview*"si es la imagen de análisis.

[4] `http://www.heidisql.com/`

10.5. Arquitectura

Al no tener una capa de interfaz con el usuario, en esta aplicación se incluye únicamente un *Controlador*, que es la interfaz *REST* principal - y el resto de clases y bibliotecas que la complementan -, y un *Modelo*, en este caso bien diferenciado, ya que se incluyen varias clases siguiendo el esquema de diseño de una clase *POJO*[5] - *Plain Old Java Object* -.

10.5.1. Diagrama de clases

A continuación se muestra el diagrama de clases general *UML* de la aplicación. Los paquetes mostrados indican la separación de las diferentes clases, no la jerarquía completa de paquetes.

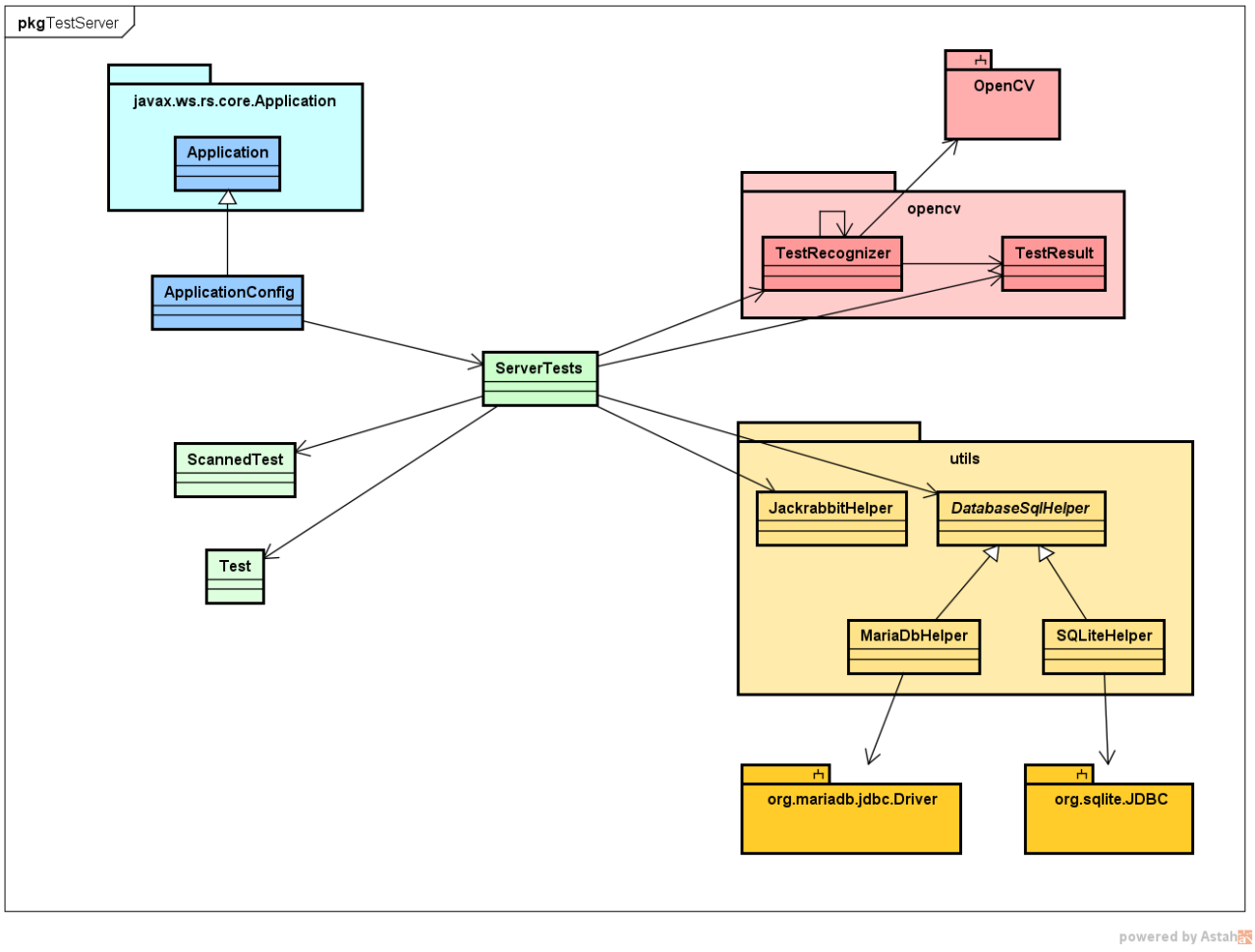

Figura 10.2: Aplicación servidor - Diagrama de clases general

[5] https://en.wikipedia.org/wiki/Plain_Old_Java_Object

La clase central del software es *ServerTests*, y alrededor de ella se construyen varias clases para proporcionar métodos de ayuda en la realización de las funciones *REST*.

Al igual que en las aplicaciones anteriores, ahora se muestran versiones más detalladas y separadas por paquetes del diagrama anterior para poder visualizar ciertas dependencias, métodos y parámetros:

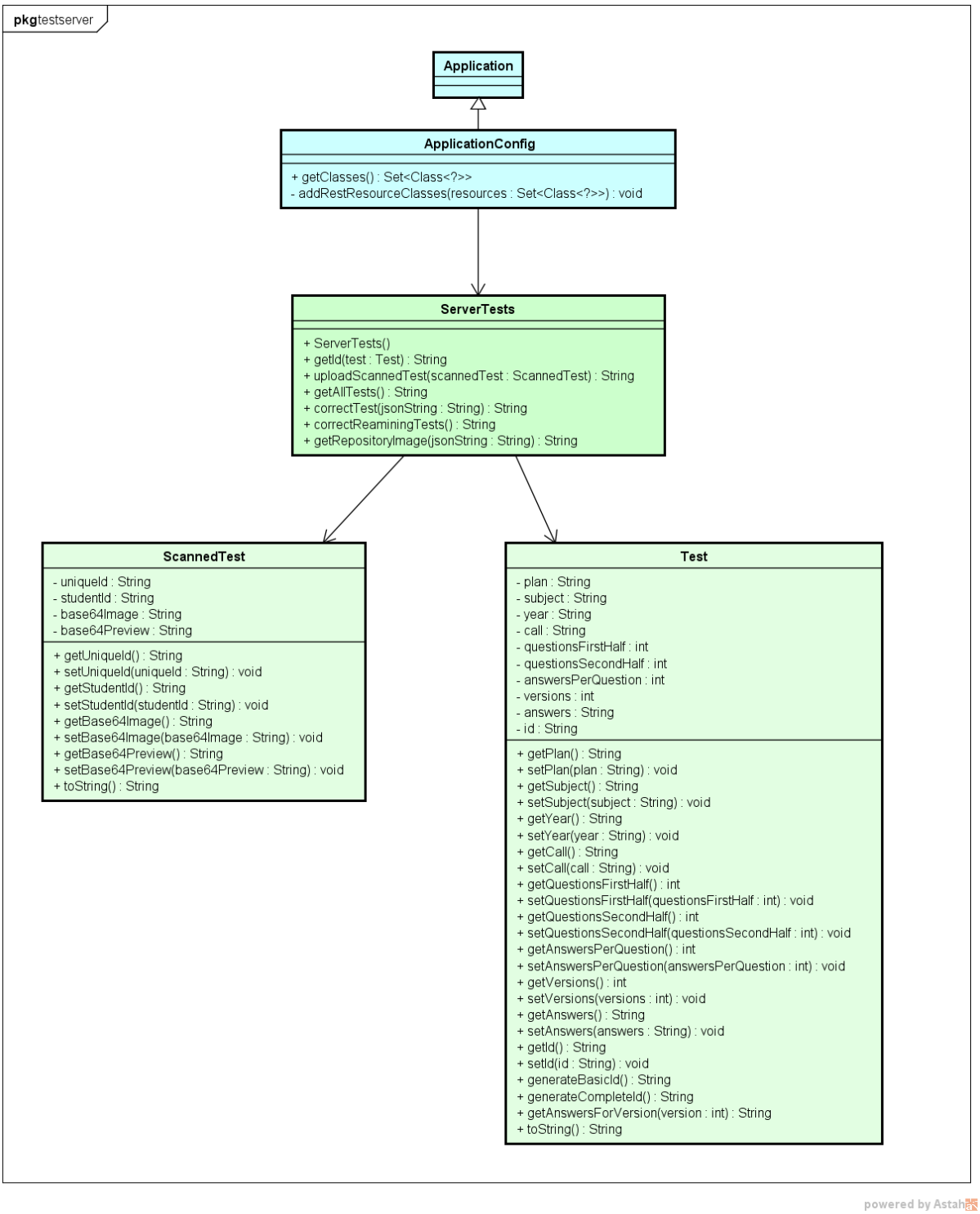

Figura 10.3: Aplicación servidor - Diagrama de clases general detallado

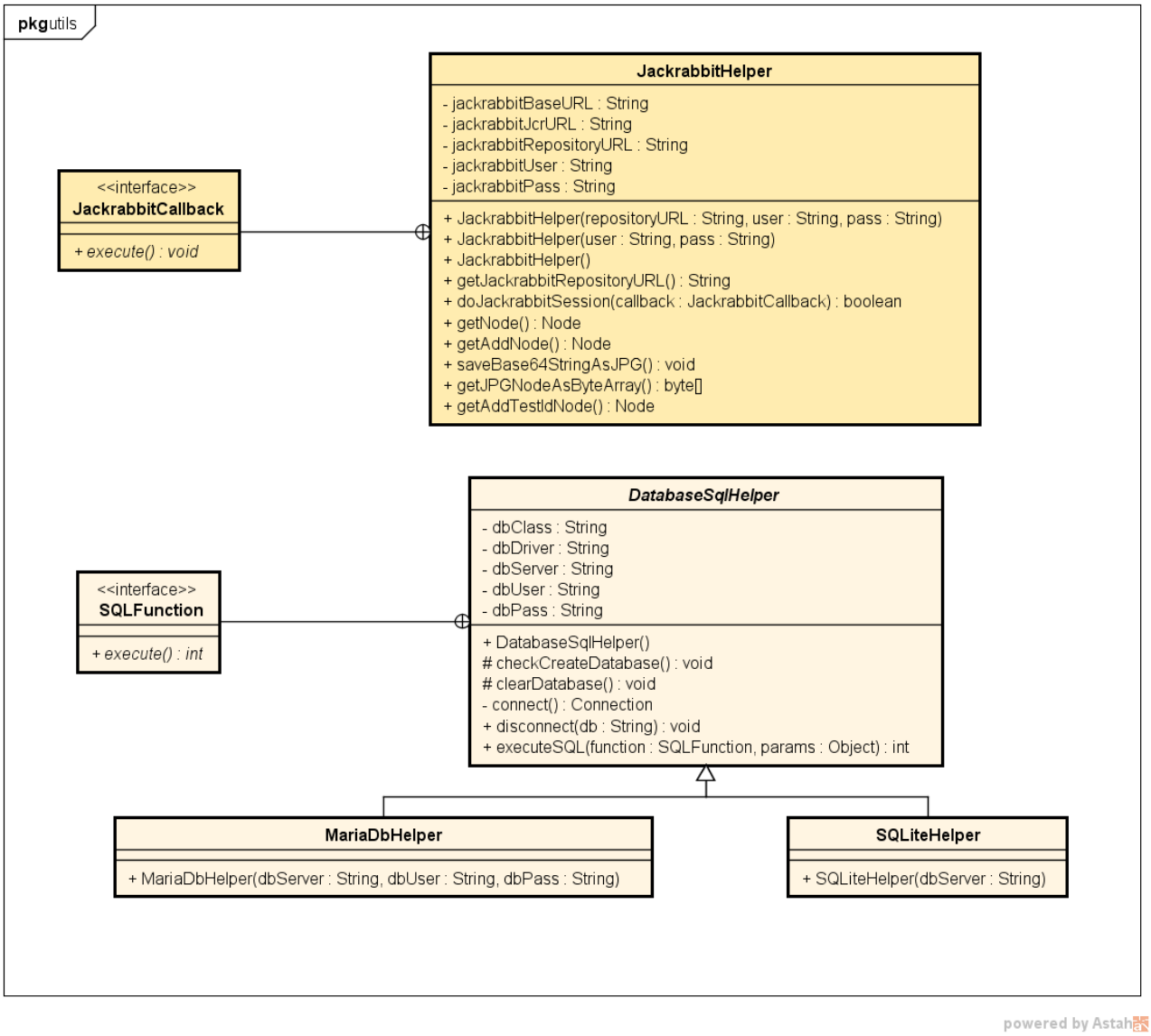

Figura 10.4: Aplicación servidor - Diagrama de clases detallado del paquete de utilidades

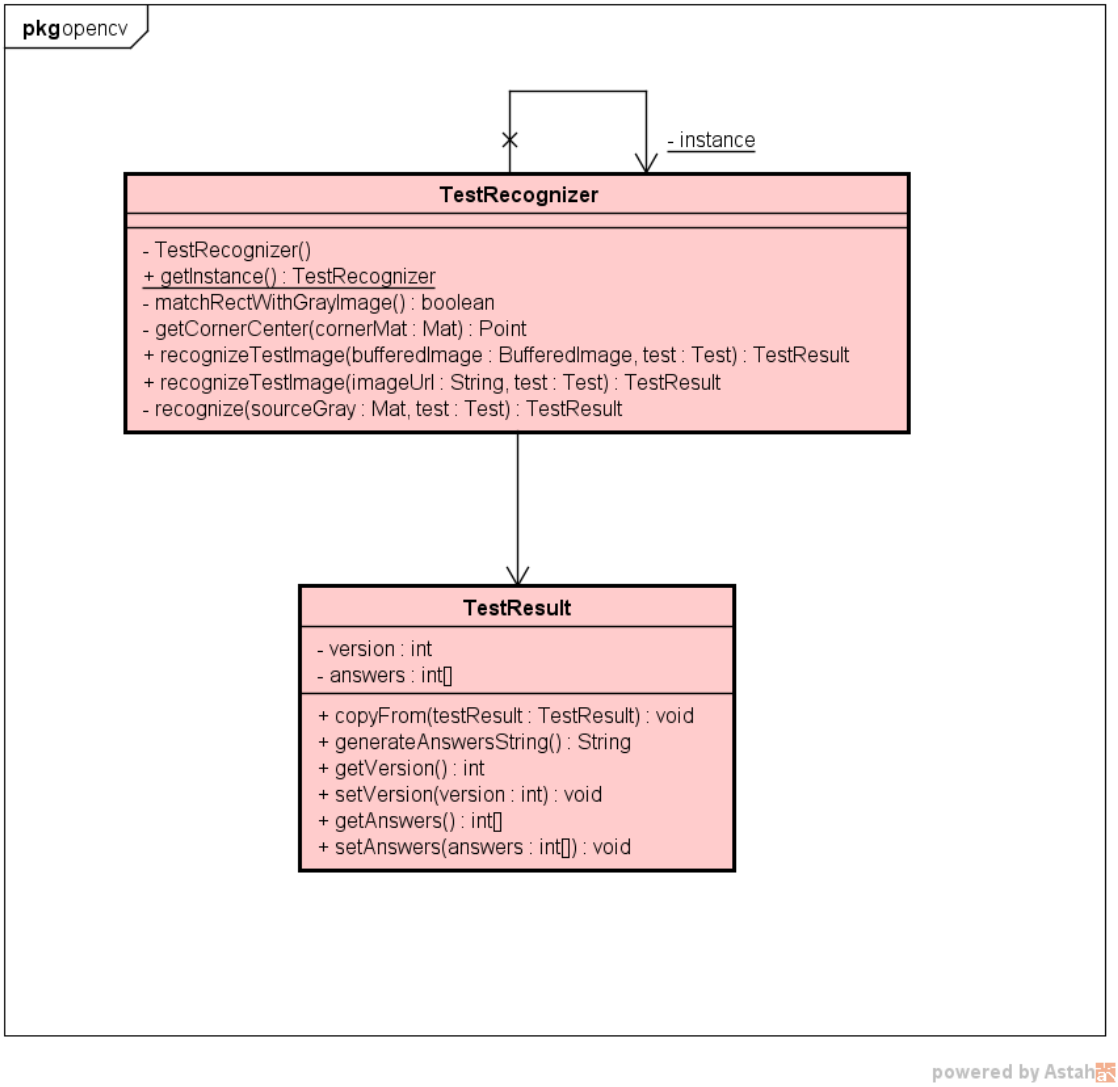

Figura 10.5: Aplicación servidor - Diagrama de clases detallado del paquete de *OpenCV*

Los componentes y las clases mostradas en el diagrama son las siguientes:

- **Paquete general**. Es el paquete básico de la aplicación y contiene el resto de paquetes y las clases principales.
  - **ServerTests**. Es la clase principal de la aplicación. Implementa el servicio *REST* de la misma - sus funciones están indicadas en secciones anteriores -, y utiliza el resto de clases para realizar todas sus funciones.
  - **ApplicationConfig**. Es una clase generada por *Netbeans* y añade los recursos y configuración básica del servidor.
  - **Application**. Realmente esta clase no pertenece al paquete general ni es utilizada activamente en la aplicación, pero se referencia porque es la clase padre de *ApplicationConfig* y la que permite la configuración básica del servidor.
  - **Test**. Clase *POJO* básica que define un test *Oficial*. Incluye además pequeños métodos para tareas comunes como recoger el número de identificación completo del examen o las respuestas para una determinada versión.

  - **ScannedTests**. Clase *POJO* básica que define un test escaneado por la aplicación móvil.

- **Paquete de utilidades**. Este paquete contiene clases centradas a implementar y facilitar el acceso a la base de datos y el respositorio.

  - **JackrabbitHelper**. Implementa la creación, conexión y desconexión al repositorio, así como otorga métodos para poder realizar operaciones en el mismo. Utiliza la biblioteca de *Jackrabbit* de *Apache*. En esta clase está contenida tanto la posibilidad de usar un servidor externo como de crear y utilizar uno embebido.
  - **DatabaseSqlHelper**. De manera similar a la clase anterior, esta clase abstracta otorga métodos para inicializar, conectar y desconectar la base de datos así como funciones para realizar peticiones. Se basa en el interfaz *JDBC* y dentro de los métodos de inicialización permite crear las tablas básicas del sistema. El uso de *JDBC* permite un interfaz común para el acceso a una base de datos *SQL*, por lo que se pueden realizar las mismas operaciones mediante el mismo código en diferentes bases de datos; esto queda soportado por diferentes *drivers* que implementan *JDBC*, y que quedan definidos en las clases concretas que hereden de esta.
  - **MariaDbHelper**. Implementación concreta de la clase *DatabaseSqlHelper* que da la dirección, el nombre del controlador y los datos necesarios para utilizar un servidor externo *MariaDB* como base de datos.
  - **SQLiteHelper**. Implementación concreta de la clase *DatabaseSqlHelper* que da la dirección, el nombre del controlador y los datos necesarios para utilizar un servidor embebido *SQLite* como base de datos.
  - ***org.mariadb.jdbc.Driver***. No es una clase de la aplicación, pero se sitúa en el diagrama para referenciar el uso del controlador *JDBC* para *MariaDB*.
  - ***org.sqlite.JDBC***. No es una clase de la aplicación, pero se sitúa en el diagrama para referenciar el uso del controlador *JDBC* para *SQLite*.

- **Paquete de *OpenCV***. Contiene clases que implementan el algoritmo de reconocimiento y corrección utilizando la biblioteca *OpenCV*.

  - **TestRecognizer**. Implementa un patrón *Singleton* y utiliza la implementación de *OpenCV* en *Java* para reconocer y corregir los tests. Es una adaptación directa, con muy pocos cambios, del algoritmo utilizado en la aplicación para dispositivos móviles.
  - **TestResult**. Encapsula el resultado de una corrección, con la versión y las respuestas seleccionadas por el alumno.
  - **OpenCV**. Esta biblioteca no es parte directa de la aplicación, pero es utilizada en este paquete para ejecutar diferentes procesos sobre las imágenes de los tests para realizar su reconocimiento y corrección.

10.5.2. Patrones de diseño

En esta aplicación se utilizan varios patrones:

- **Singleton**. Utilizado en la clase *TestRecognizer*, sirve para utilizar un único reconocedor y corrector de tests, ya que se considera como una utilidad. Además, este patrón permite que se pueda acceder a él desde cualquier punto del programa.
- **Estrategia**[6]. Este patrón permite definir cómo se realiza una determinada operación, otorgando la posibilidad de construir diferentes implementaciones de la misma para casos que requieran comportamientos distintos.

[6] `https://es.wikipedia.org/wiki/Strategy_(patr%C3%B3n_de_dise%C3%B1o)`

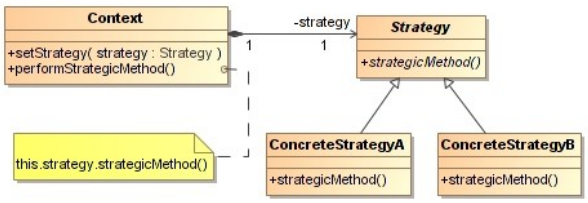

Figura 10.6: Esquema patrón estrategia

En la aplicación se utiliza una adaptación del mismo para definir el acceso a la base de datos. La clase *DatabaseSqlHelper* define la estrategia abstracta y las clases *MariaDbHelper* y *SQLiteHelper* las estrategias concretas. En este caso, la clase *contexto* que utiliza las diferentes estrategias sería la clase principal *ServerTests*.

10.5.3. Soporte de servidores

Como se ha indicado en capítulos anteriores, esta aplicación se puede desplegar utilizando servidor externos o embebidos para la base de datos y el repositorio. Para soportar esto, se utiliza una variación del patrón de diseño *Estrategia* con las clases *DatabaseSqlHelper*, *MariaDbHelper* y *SQLiteHelper*.

Por tanto, a nivel de código la diferencia de utilizar una u otra opción se reduce a elegir una estrategia concreta e instanciarla indicando la configuración del servidor en cada caso. De esta manera, se consigue el soporte y la abstracción adecuada para que los cambios a nivel de código sean mínimos en el uso de cualquiera de las dos posibilidades.

10.6. Implementación

El código de esta aplicación no es tan importante para este documento como el de la aplicación de escaneo, pero sí se pueden encontrar ciertas secciones de interés.

10.6.1. ServerTests

El primer punto destacable de la clase principal es la carga de la biblioteca nativa de *OpenCV*. Ésta se realiza en un entorno *estático*, garantizando que se ejecutará una única vez al instanciar la aplicación.

```
1 //Load OpenCV native library (only should be loaded once).
2 static {
3   try {
4     System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
5   } catch (UnsatisfiedLinkError exception) {
6     System.err.println("Catched OpenCV reloading exception.");
7   }
8 }
```

A pesar de ello y como se puede observar, se ha colocado una orden *try* para gestionar la excepción que surge al intentar cargar dos veces la biblioteca. Al ser una aplicación servidor, puede darse alguna situación en que la clase principal del mismo se reinicie o se borre y se vuelve a instanciar por algún fallo u operación de actualización del

servidor; al ocurrir esto, se volvería a ejecutar la sección *estática*, pero la biblioteca seguiría cargada en el sistema, lo que daría pie al error que es tratado.

La interfaz *REST* creada se ha realizado utilizando las anotaciones preparadas por *JAX-RS* para ello. Se muestra como ejemplo un fragmento del método *correct_test*:

```
@POST
@Path("correct_test")
@Consumes("text/plain;charset=UTF-8")
@Produces("text/plain;charset=UTF-8")
public String correctTest(String jsonString) {
```

Primeramente se indica que es un método *HTTP* de tipo *POST*[7], después el nombre con el que será accedido el método desde la red y por último se define qué tipo de datos *MIME*[8] - *Multipurpose Internet Mail Extensions* - consume y produce; en ambos caso se trabaja con texto plano con el formato de caracteres *UTF-8*[9].

10.6.2. DatabaseSqlHelper

Esta clase abstracta implementa varios métodos relativos a la gestión de una base de datos *SQL*. Al crear la clase se comprueba si la base de datos contiene las tablas básicas del sistema, creándolas si es necesario. También incluye un método para poder borrar las tablas si es necesario:

```
/**
* Drop tests system tables.
*/
protected void clearDatabase() {
  //Execute SQL function to clear database.
  executeSQL(new SQLFunction() {
    @Override
    public int execute(Connection connection, Statement statement,
        Object... params) throws SQLException {
      //Drop tables.
      statement.executeUpdate("DROP TABLE IF EXISTS tests_scanned;");
      statement.executeUpdate("DROP TABLE IF EXISTS tests;");
      connection.commit();

      return 0;
    }
  });
}
```

Para borrarlas, utiliza sentencias sencillas en *SQL* que son ejecutadas mediante los métodos que provee esta clase para dicha tarea. Éstos, simplifican la ejecución de sentencias reduciéndolas a crear una clase genérica con los comandos que se deben ejecutar, emulando el funcionamiento directo de una consola *SQL*. El código que otorga estas funciones se muestra a continuación:

```
/**
```

[7] http://www.restapitutorial.com/lessons/httpmethods.html
[8] https://en.wikipedia.org/wiki/MIME
[9] https://en.wikipedia.org/wiki/UTF-8

```
2  * Interface to ease execution of SQL functions.
3  */
4  public interface SQLFunction {
5
6    /**
7    * Callback to be executed.
8    *
9    * @param connection Active connection to database.
10   * @param statement Active statement to database.
11   * @param params Optional extra parameters.
12   * @return Integer as result.
13   * @throws SQLException
14   */
15   public int execute(Connection connection, Statement statement,
16        Object... params) throws SQLException;
17 }
18
19 /**
20 * Methods to ease execution of SQL function.
21 *
22 * @param function Function to execute.
23 * @param params optional extra parameters.
24 * @return
25 */
26 public int executeSQL(SQLFunction function, Object... params) {
27   //Connect to database.
28   Connection connection = connect();
29   Statement statement = null;
30   int finalState = -1;
31   //Check if correct connection.
32   if (function != null && connection != null) {
33     try {
34       //Execute callback and close.
35       statement = connection.createStatement();
36       connection.setAutoCommit(false);
37       finalState = function.execute(connection, statement, params);
38       statement.close();
39       connection.close();
40     } catch (SQLException ex) {
41       //Close statement.
42       if (statement != null) {
43         try {
44           statement.close();
45         } catch (SQLException ex1) {
46           System.err.println("Error closing database statement.");
47         }
48       }
49       System.err.println("Error executing statement in database.");
50     } finally {
51       disconnect(dbServer);
52     }
53   }
54   return finalState;
55 }
```

El interfaz *SQLFunction* permite crear una nueva clase que recibe una conexión activa a la base de datos,

un *Statement* para ejecutar las diferentes operaciones y unos objetos como parámetros adicionales por si fueran necesarios. Luego, el método *executeSQL* recibe una instancia de este tipo y unos parámetros opcionales que serán pasados directamente a la instancia de *SQLFunction* dada. Dentro del método, se realiza toda la configuración necesaria para realizar la conexión, se ejecuta la función dada y posteriormente se cierra la conexión.

10.6.3. MariaDbHelper

Esta implementación concreta de la clase anterior tiene como fin soportar el acceso a una base de datos externa *MariaDB*. Para ello, indica el controlador *JDBC* a utilizar, la dirección del servidor y las credenciales de accesso.

```
1  /**
2  *
3  * Class that implements DatabaseSqlHelper to be used with MariaDB client.
4  *
5  * @author Sergio Escanciano Arribas.
6  */
7  public class MariaDbHelper extends DatabaseSqlHelper {
8
9    /**
10   * Define MariaDB database helper.
11   * @param dbServer Server URL.
12   * @param dbUser Server user.
13   * @param dbPass Server pass.
14   */
15   public MariaDbHelper(String dbServer, String dbUser, String dbPass) {
16     //Pass MariaDB data.
17     super("org.mariadb.jdbc.Driver",
18          "jdbc:mariadb://", dbServer, dbUser, dbPass);
19   }
20 }
```

10.6.4. SQLiteHelper

Esta clase implementa de manera análoga a la anterior la clase *DatabaseSqlHelper*. Ésta sirve para acceder a una base de datos embebida *SQLite*. Ésta crea la base de datos en un fichero en la dirección raíz de la aplicación, por lo que además del controlador *JDBC* esta clase especifica la ruta de dicho fichero.

```
1  /**
2  * Class that implements DatabaseSqlHelper to be used with an embedded SQLite
3  * database.
4  *
5  * @author Sergio Escanciano Arribas
6  */
7  public class SQLiteHelper extends DatabaseSqlHelper {
8
9    /**
10   * Define SQLite database helper.
11   * @param dbServer Database file name.
12   */
13   public SQLiteHelper(String dbServer) {
```

```
14      //Pass MariaDB data.
15      super("org.sqlite.JDBC", "jdbc:sqlite:", dbServer, null, null);
16    }
17  }
```

10.6.5. TestRecognizer

Esta clase implementa el algoritmo de reconocimiento y corrección de tests mostrado en la aplicación de escaneo, adaptándolo a la versión *Java* de la biblioteca *OpenCV*. El funcionamiento e incluso la escritura del código es prácticamente igual, teniendo como diferencias el nombre y situación de ciertos métodos y clases de *OpenCV* y la manera de instanciar ciertos objetos. Sobre ello se añaden además algunos pequeños procedimientos, como el dedicado a convertir imágenes desde un tipo *BufferedImage* de *Java* o el relativo a leer imágenes directamente desde una ruta - hay que tener en cuenta que en la aplicación móvil las imágenes llegaban siempre desde la previsualización de la cámara, lo cual no es el caso en esta situación -.

Por lo descrito, a lo largo de esta sección se van a mostrar diferentes fragmentos de código que difieren del algoritmo original.

El algoritmo principal queda implementado en el método privado *recognize*, que se ejecuta a partir de una matriz *Mat* de *OpenCV* en escala de grises. Los métodos públicos para acceder al algoritmo reciben una imagen de origen, bien contenida en una clase *BufferedImage* o como una ruta a un fichero de imagen. La primera opción se ha implementado de la siguientes manera.

```
1  /**
2  * Recognize test from BufferedImage.
3  *
4  * @param bufferedImage Buffered image with the test.
5  * @param test Test basic data.
6  * @return Test recognition result.
7  */
8  public TestResult recognizeTestImage(BufferedImage bufferedImage, Test test) {
9    //Load image from buffered image.
10   byte[] pixels = ((DataBufferByte)
         bufferedImage.getRaster().getDataBuffer()).getData();
11   Mat sourceColor = new Mat(
12        bufferedImage.getHeight(), bufferedImage.getWidth(),
13        CvType.CV_8UC3);
14   sourceColor.put(0, 0, pixels);
15   //Convert to grayscale.
16   Mat sourceGray = new Mat();
17   Imgproc.cvtColor(sourceColor, sourceGray, Imgproc.COLOR_BGR2GRAY);
18
19   //Pass it to recognize.
20   return recognize(sourceGray, test);
21 }
```

Este método crea una matriz en color a partir de los datos de la *BufferedImage* y después introduce todo los píxeles de la misma en la matriz. A partir de esta última, se genera otra en escala de grises y se llama al algoritmo de reconocimiento.

Por otra parte, el método que recibe una ruta utiliza la función de *OpenCV imread*[10] contenida en la clase *Imgcodecs*[11] para cargar una imagen en escala de grises directamente desde dicha dirección.

```
/**
* Recognize test directly form a URL.
*
* @param imageUrl Image URL.
* @param test Test basic data.
* @return Test recognition result.
*/
public TestResult recognizeTestImage(String imageUrl, Test test) {
  //Load image directly as grayscale.
  Mat sourceGray = Imgcodecs.imread(imageUrl, Imgcodecs.IMREAD_GRAYSCALE);

  //Pass it to recognizer.
  return recognize(sourceGray, test);
}
```

Otra diferencia es que lo que se devuelve a la clase que utilice *TestRecognizer* es un objeto de la clase *TestResult*, que como se ha indicado encapsula el resultado de la corrección.

```
//Fill result.
TestResult testResult = new TestResult();
testResult.setAnswers(answers);
testResult.setVersion(currentVersion);
return testResult;
```

Por último, a modo de ejemplo se muestra el método *getCornerCenter*, para poder apreciar las diferencias del código *Java* respecto a la implementación original en *C*.

```
/**
* Get center point of a test corner.
*
* @param cornerMat Image matrix to search the corner.
* @return Center point.
*/
private Point getCornerCenter(Mat cornerMat) {
  //Find contours (two nested contours).
  List<MatOfPoint> contours = new ArrayList<>();
  Mat hierarchy = new Mat();
  Imgproc.findContours(cornerMat, contours, hierarchy,
  Imgproc.RETR_TREE, Imgproc.CHAIN_APPROX_SIMPLE);

  //Check corners found.
  if (contours.size() > 0 && hierarchy.size().width >= 4) {
    //Calculate moments by contours.
    int index = contours.size() > 1 ? 1 : 0;
    Moments mom = null;
    //Get contours until one is correct.
    do {
```

[10] http://docs.opencv.org/2.4/modules/highgui/doc/reading_and_writing_images_and_video.html
[11] http://docs.opencv.org/3.0-beta/modules/imgcodecs/doc/imgcodecs.html

```
21          mom = Imgproc.moments(contours.get(index));
22          index++;
23        } while (mom.m00 == 0.0 && index < contours.size() - 1);
24
25        //Find centroid.
26        Point centroid = new Point(mom.m10 / mom.m00, mom.m01 / mom.m00);
27        return centroid;
28      } else {
29        //Not detected, return 'null' point.
30        return new Point(-1, -1);
31      }
32    }
```

Capítulo 11

Evolución del proyecto

El proyecto se ha gestado en un largo período de tiempo que abarca varios años, aunque no de desarrollo continuado. Durante los tiempos de actividad, siempre se ha mantenido la idea básica de realizar un sistema para permitir el reconocimiento automatizado de exámenes tipo test, pero los medios utilizados para ello han ido evolucionando hasta la solución final.

En este capítulo se realiza un breve recordatorio de algunos puntos importantes.

11.1. Planteamiento inicial

En las etapas iniciales, uno de los pilares fundamentales del proyecto era que debía ser multiplataforma en su aplicación móvil, al menos para los dos principales sistemas operativos, *Android* e *iOS*. Para ello se estudió el uso de varios *frameworks* que permitieran realizar código reutilizable:

- **Adobe Phonegap**[1]. Esta tecnología - bastante menos madura en el momento en el que pretendía utilizarse -, permite crear aplicaciones multiplataforma escritas en utilizando *HTML*, *CSS* y *JavaScript*. Ahora mismo soporta todos los sistemas operativos móviles importantes[13]. Dada la necesidad de utilizar código nativo y con características especiales en este proyecto, en ningún caso hubiera sido una opción adecuada, ya que este sistema está pensado para construir aplicaciones más sencillas.
- **RoboVM**. Esta tecnología permitía desarrollar aplicaciones para el sistema operativo *iOS* en *Java*, pudiendo reutilizar gran parte del código generado para *Android* e incluso acceder a código nativo y funciones propias de *iOS*. Además la aplicación generada finalmente funciona bajo código nativo, no como un simple intérprete de *Java*. Esta opción hubiera sido válida con la tecnología e implementación que se utiliza ahora en el proyecto, pero habría llevado a perder el enfoque en perfeccionar el sistema.

Poco después, y tras analizar la complejidad del proyecto, se decidió construir la aplicación móvil únicamente en *Java* y para el sistema operativo *Android*. En cuanto al reconocimiento del código *QR*, desde el primer momento se determinó utilizar la biblioteca *ZXing*, ya que era la más extendida y la que permitía más flexibilidad en su uso.

La aplicación de generación y visualización es la que más ha mantenido su diseño inicial. Ya en esa época se realizó el proyecto básico en *Netbeans* y se utilizaba *Java Swing* para la interfaz gráfica.

En cuanto al servidor, en este punto se tenía la idea de utilizar *Java*, *Netbeans* y un servidor *GlassFish*, pero la arquitectura y diseño básico todavía no se había planteado completamente.

[1] `http://phonegap.com/`

11.2. Utilización de *Android Studio* y métodos de reconocimiento anteriores

Una vez creada una versión muy básica del generador y la plantilla - ésta todavía muy lejos de la iteración final del diseño -, se empezó a investigar y trabajar en el que llegaría a ser el algoritmo de reconocimiento y corrección.

En ese momento el entorno de desarrollo *Android Studio* empezaba a tener sus primeras versiones estables. Dado que se estaba realizando una aplicación para *Android*, parecía una buena oportunidad utilizarlo. Éste trae varias mejoras respecto a su principal *competidor*, *Eclipse*. Entre ellas se pueden destacar el uso de *Gradle*[2] para gestionar las dependencias del proyecto - lo que simplifica mucho el uso y mantenimiento de librerías externas de *Android* -, una mejor integración con las características de las últimas versiones de *Android* y un rendimiento superior.

Con este entorno de desarrollo se empezó a construir la aplicación para dispositivos móviles. En un primer acercamiento, se tomaban fotos que eran procesadas por el algoritmo. Pronto se comprobó que intentar capturar *a la primera* una foto adecuada del examen era una tarea complicada, así que se ideó un sistema para intentar descartar las capturas no válidos. Para ello se investigó en el campo de lo que es denominado como *NRIQA - No-Reference Image Quality Assessment* -. Este tipo de técnicas pretenden evaluar la calidad de una imagen en base a diferentes características y descriptores; se llegó a realizar una primera implementación en el proyecto en base al documento *Real-time No-Reference Image Quality Assessment based on Filter Learning*[12].

Lo indicado iba acompañado del uso de un filtro *Gaussiano*[3] y una binarización básica. Tras varios ensayos y optimizaciones en la implementación, se vio que el aplicar estos algoritmos a imágenes de la clase *Bitmap* de *Android* directamente desde el código *Java* no iba a permitir en ningún caso un análisis en tiempo real, ya que el uso y manipulación de objetos *Bitmap* requiere una gran cantidad de tiempo de ejecución y memoria.

Fue en ese momento cuando se planteó pasar el algoritmo de reconocimiento y corrección a código nativo mediante el uso del *NDK*. Antes de dar el paso, se investigó sobre las implicaciones de esto, las configuraciones necesarias y las posibles bibliotecas que podrían ayudar para esta tarea. Entonces se encontró la biblioteca que definiría el posterior rumbo del proyecto: *OpenCV*.

11.3. Paso a *Eclipse* : Mejora de rendimiento con *NDK* y *OpenCV*

La biblioteca *OpenCV* es una de las más extendidas en el campo del tratamiento de imágenes. Posee implementaciones oficiales para *C*, *C++*, *Java* y *Python*, entre otros lenguajes, y tiene soporte oficial en multitud de plataformas como *Android*, *Windows*, *Mac OS*, *Linux* o *iOS*; además, tiene una gran comunidad, documentación y soporte.

Este sistema disponía de todas las características necesarias para poder abordar el proyecto, así que comenzó su integración. Las primeras instalaciones y configuraciones fueron problemáticas, porque *Android Studio* no tiene una buena integración del *NDK*. Para poder trabajar adecuadamente en código nativo se creó el proyecto de nuevo en el entorno de desarrollo *Eclipse*, añadiendo el soporte para *NDK* y *OpenCV*, y manteniendo el código escrito en *Android Studio*.

Según se fue avanzando el algoritmo de reconocimiento hasta la implementación final, se concluyó que el análisis de la calidad de la imagen tomada no era necesario, ya que necesitaba un tiempo de procesamiento que no daba ninguna ventaja. Utilizando las técnicas de reconocimiento presentadas, era posible descartar rápidamente los fotogramas no válidos sin aplicar otro proceso por encima. Por este motivo, el código realizado para las tareas de *NRIQA* terminó siendo desechado, y la técnica de binarización y desenfoque *Gaussiano* adaptadas para ser utilizadas con *OpenCV*.

[2] `http://gradle.org/`

[3] `https://en.wikipedia.org/wiki/Gaussian_filter`

Capítulo 12

Conclusiones

12.1. Valoración

Llegado el final de este documento, no es sencillo redactar unos párrafos que recojan todo lo aprendido y desarrollado a lo largo del proyecto. Ha sido un trabajo con un gran número de componentes, que ha requerido realizar tareas de diferentes ramas, teniendo todas ellas, eso sí, relación directa o indirecta con la informática.

Desde la concepción inicial, se tuvo claro que el proyecto precisaba bastante investigación para resolver los problemas, principalmente los relacionados con el reconocimiento y corrección de los exámenes. Como se indicó en la introducción, una de las motivaciones originales era construir un sistema para corregir exámenes de tipo test de manera sencilla y sin el requerimiento de un hardware especial, ya que no existen muchos ejemplos de proyectos que hayan realizado algo similar.

Analizando el sistema entregado finalmente, se han presentado las siguientes características principales:

- Generación de exámenes tipo test con las siguientes posibilidades:
  - Información administrativa básica.
  - Hasta 8 versiones.
  - Hasta 50 preguntas.
  - Hasta 8 respuestas por pregunta.
  - Posibilidad de generar test de tipo *Oficial* o *Autocorrección*.
- Posibilidad de almacenar los test de tipo *Oficial* en un servidor para poder ser corregidos y estudiados posteriormente.
- Posibilidad de corregir los exámenes de tipo *Autocorrección* al momento desde una aplicación móvil.

Haciendo balance con los objetivos propuestos en un inicio, y valorando la evolución del proyecto, considero que en general se han cumplido.

Desde un punto de vista cultural y técnico, el diseño e implementación del algoritmo de reconocimiento y corrección es lo que más me ha aportado. Además, se debe destacar la necesidad de trabajar en código nativo y con la biblioteca *OpenCV*, que me ha proporcionado conocimientos para crear desde cero una aplicación *Android* que integre el entorno *NDK*.

Uno de los principales valores aportados por el desarrollo del proyecto, ha sido construir un sistema compuesto de varias aplicaciones conectadas entre sí, intentando conseguir una arquitectura adecuada, escalable y que requiera un mantenimiento lo más sencillo posible.

Finalmente, teniendo en cuenta lo realizado, los fallos y las características descartadas, pienso que se ha efectuado un trabajo competente, produciendo un sistema base funcional, que puede ser utilizado en un entorno académico controlado y servir de base para la realización de un sistema más complejo y completo.

12.2. Trabajo futuro

En este proyecto se han implementado un gran número de características que se han descrito en esta memoria. Como en todo trabajo - especialmente en los relacionados con el software -, ciertos componentes han cambiado, algunos se han eliminado y otros han aparecido nuevos a lo largo del desarrollo; en cualquier caso, siempre hay que trazar un punto donde el proyecto se da por finalizado. En este capítulo se indican algunas características que finalmente no han sido implementadas, pero para las que el sistema ha quedado preparado, dando pie a posibles trabajos futuros de cara a mejorar y ampliar las aplicaciones:

- **Escaneado con diferentes resoluciones**. El algoritmo diseñado trabaja con capturas de una resolución de *480x640*. Todas las medidas están situadas en variables para poder ser modificadas, pero actualmente sus valores están calculados en base a dicha resolución. Se ha comprobado que el índice de éxito y el rendimiento utilizando estos datos da buenos resultados, pero de cara a incluir nuevos dispositivos o características se podría modificar el algoritmo para que, dada una resolución, recalculara las medidas y ciertos valores - como los índices de aceptación de lo que es una mancha válida, por ejemplo - para poder reconocer y corregir los exámenes.

- **Implementación de otra aplicación servidor**. En el sistema actual el servidor utilizado está construido para la ocasión. De cara a poder implementar este proyecto dentro de un ecosistema diferente y más complejo, las tareas que ejecuta aquí el servidor se podrían reimplementar para utilizar un sistema diferente, como por ejemplo *Moodle*[1].

- **Conversión de aplicación de escritorio en aplicación web**. El software que se utiliza para visualizar y generar los tests es una aplicación *Java* de escritorio. Como está diseñada ahora cumple su función, pero dada la evolución del desarrollo de software, lo más común actualmente habría sido diseñar el generador y visualizador como una aplicación web que realmente fuera algo similar a un componente de tipo *Vista* de la aplicación servidor.

- **Cifrado de la información**. Los datos enviados entre el servidor y las aplicaciones con interacción con el usuario no están cifrados, se envían tal cual se componen. Dado que la seguridad no es el objeto principal de este trabajo, y que al trabajar en un sistema cerrado y controlado no se tiene el mismo riesgo que trabajando en internet o en un sistema más grande, se prefirió centrar los esfuerzos del proyecto en mejorar otras características. No obstante de cara a una distribución mayor de las aplicaciones o a una integración en un sistema más grande, se deberían implementar políticas de seguridad y añadir métodos criptográficos.

- **Soporte de tests multi-página**. Los tests diseñador permiten definir hasta 50 preguntas con hasta 8 respuestas por pregunta y en hasta 8 versiones diferentes. Se consideró un número de posibilidades suficiente como para cubrir los exámenes tipo test básicos, pero de cara a dar un mayor número de posibilidades y crear tests más complejos, se podría dar soporte a tests con más de una página.

[1] `https://en.wikipedia.org/wiki/Moodle`

- **Soporte de preguntas con múltiples respuestas**. En la especificación descrita cada pregunta tiene una única respuesta; de cara a dar más posibilidades se podría implementar el soporte de preguntas con múltiples respuestas.
- **Soporte de diferentes respuestas por pregunta**. Todas las preguntas del test tienen el mismo número de respuestas. Siguiendo la línea de las anteriores propuestas, se podría implementar la posibilidad de que cada pregunta tuviera un número libre de respuetas - o libre hasta un máximo de 8, en base a la especificación actual -.
- **Internacionalización**. La interfaz de usuario de las aplicaciones está escrita en español. Podría implementarse el soporte para diferentes idiomas de cara a distribuir la aplicación entre usuarios no hispanoparlantes; *Android* ya incluye un soporte bastante completo, por lo que la transición sería bastante directa.
- **Aplicación móvil para varias plataformas**. Siguiendo uno de los objetivos iniciales del proyecto y de cara a tener una mayor base de usuarios, se podría desarrollar la aplicación de escaneo para otros dispositivos y sistemas operativos, como *iOS* o *Windows Phone*.

Bibliografía

[1] Android. Controlling the camera. https://developer.android.com/training/camera/cameradirect.html?hl=es.

[2] Eyal Arubas. Opencv installation on windows + netbeans + mingw, 2012. http://eyalarubas.com/opencv-installation-on-windows-netbeans-mingw.html.

[3] Inc. Change Vision. Astah reference manual, 2006-2012. http://astah.net/tutorials/astah%20professional%20referencemanual.pdf.

[4] Cameron Chapman. The simplicity of helvetica. http://www.webdesignerdepot.com/2010/01/the-simplicity-of-helvetica/, 2010.

[5] Microsoft Corporation. Tutorial for visio 2016, 2015. https://support.office.com/en-us/article/Tutorial-for-Visio-2016-c8fd9b8b-6e8c-4252-937d-a0eea0cddd94.

[6] Oracle Corporation. Información de java 8. https://www.java.com/es/download/faq/java8.xml.

[7] Oracle Corporation. Building restful web services with jax-rs. https://docs.oracle.com/cd/E19798-01/821-1841/giepu/index.html, 2010.

[8] Oracle Corporation. Chapter 13 building restful web services with jax-rs, 2010. https://docs.oracle.com/cd/E19798-01/821-1841/giepu/index.html.

[9] Oracle Corporation. Chapter 3. jax-rs application, resources and sub-resources, 2010-2016. https://jersey.java.net/documentation/latest/jaxrs-resources.html.

[10] Oracle Corporation. Designing a swing gui in netbeans ide, 2016. https://netbeans.org/kb/docs/java/quickstart-gui.html.

[11] Learn Cpp. Learn cpp, 2007-2016. http://www.learncpp.com/.

[12] Peng Ye, Jayant Kumar, Le Kang, David Doermann. *Real-time No-Reference Image Quality Assessment based on Filter Learning*. PhD thesis, Institute for Advanced Computer Studies., University of Maryland, College Park, MD, USA, 2013. http://lampsrv02.umiacs.umd.edu/pubs/Papers/pengye-13/pengye-13.pdf.

[13] Adobe Systems Inc. Platform guides, 2015. http://docs.phonegap.com/en/edge/guide_platforms_index.md.html.

[14] Craig Larman. *UML y Patrones*. Prentice Hall, 2003.

[15] David Nuescheler. Jsr 170: Content repository for javatm technology api. Technical report, Day Software, Inc, 2005. https://jcp.org/en/jsr/detail?id=170.

[16] David Nuescheler. Jsr 283: Content repository for javatm technology api version 2.0. Technical report, Day Software, Inc, 2009. https://jcp.org/en/jsr/detail?id=283.

[17] OpenCV. Canny edge detector. http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html.

[18] OpenCV. Image thresholding. http://docs.opencv.org/3.1.0/d7/d4d/tutorial_py_thresholding.html#gsc.tab=0.

[19] Bharath P. Opencv: Qr code detection and extraction. http://dsynflo.blogspot.com.es/2014/10/opencv-qr-code-detection-and-extraction.html, 2014.

[20] Luigi De Russis. Installing opencv for java, 2014-2015. https://opencv-java-tutorials.readthedocs.io/en/latest/01-installing-opencv-for-java.html.

[21] Erich GAMMA, Richard Helm, Ralph Johnson, John Vlissides. *Patrones de Diseño*. Addison Wesley, 2006.

[22] Jukka Zitting. Embedded repository, 2009. https://cwiki.apache.org/confluence/display/JCR/Embedded+Repository.

Anexo A

Manual de usuario de la aplicación de generación y visualización

A.1. Requerimientos mínimos

El software requiere lo básico necesario para la mayoría de aplicaciones desarrolladas en *Java*:

- Equipo y sistema operativo compatible con *Java 8*.
  - Como la aplicación incluye interfaz de usuario, el sistema operativo debe tener un sistema de ventanas, ya que n ose puede ejecutar únicamente como una aplicación de consola.
- Tener instalado en el sistema *Java 8*, ya sea la versión de *Oracle*[1] o el *OpenJDK*[2].

A.2. Instalación

. La aplicación se distribuye en un fichero comprimido. Dentro de él se incluye el fichero ejecutable con extensión *jar*, una carpeta *lib* que incluye las bibliotecas de terceros utilizadas y otros archivos necesarios para el funcionamiento del software. El programa se puede descomprimir para ser utilizado en la ruta que desee el usuario, pero siempre manteniendo dentro del mismo lugar todos los ficheros y carpetas incluidos en el comprimido.

La ejecución de la aplicación se puede realizar de dos formas:

- Realizando doble click en el fichero *jar* ejecutable - si la ejecución directa de programas *Java* está configurada en el sistema operativo -.
- Ejecutando la aplicación desde una consola. Para ello se debe ir a la ruta donde se encuentra el ejecutable y utilizar el comando:

```
java -jar nombre_del_ejecutable.jar
```

[1] `https://www.java.com/es/download/`
[2] `http://openjdk.java.net/`

A.3. Uso básico

A continuación se incluyen una guía para la realización de las acciones básicas de generación y visualización.

A.3.1. Configuración de ruta del servidor

Si el usuario va a generar test de tipo *Oficial*, deberá tener instalado y funcionando la aplicación servidor del sistema. Dando por hecho que se ha realizado dicha tarea, se debe indicar la ruta del servidor en el programa de visualización y generación.

Para ello se debe pulsar el botón *Archivo* de la barra de opciones y posteriormente el botón *Configuración de servidor*. En el panel que se muestra es posible introducir la dirección del mismo.

Cabe destacar que una vez desplegado el servidor, la ruta será el equipo y puerto donde está funcionando y posteriormente la ruta que se indica a continuación - esta permite acceder a la interfaz *REST* -. En el ejemplo que se muestra a continuación la ruta del equipo era *192.168.1.106* y el puerto el *8080*:

```
http://192.168.1.106:8080/TestServer/webresources/servertests
```

A.3.2. Generación de un test

Para comenzar la generación de un examen el usuario se debe situar en la pestaña *Generador* de la aplicación.

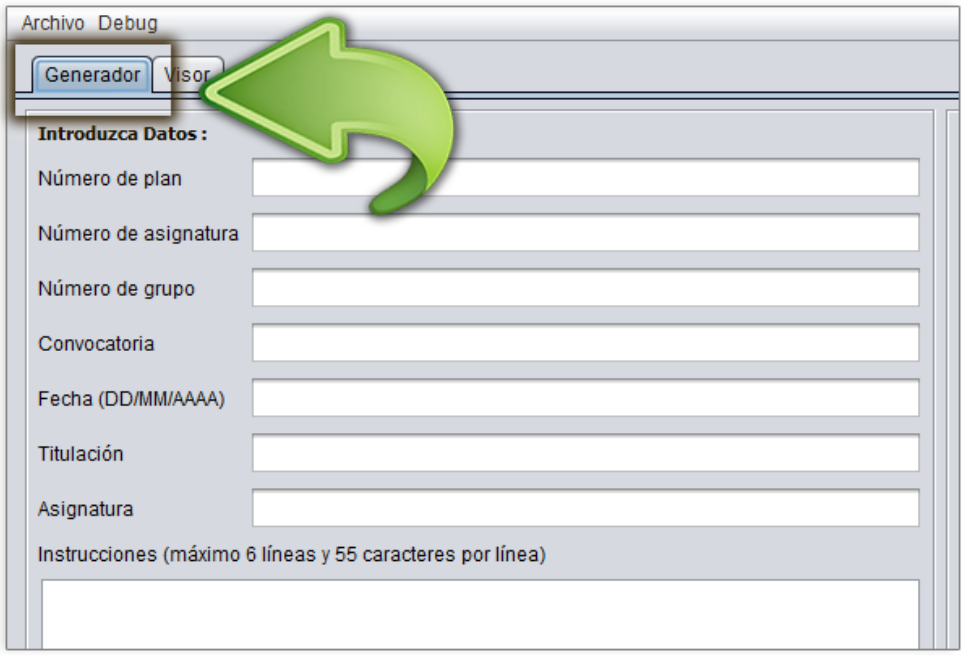

Figura A.1: Manual de la aplicación generación y visualización - Seleccionar *Generador*

Los pasos a seguir son los siguientes:

1. Primeramente se deben introducir los datos administrativos del examen. Para ello se deben rellenar los

campos situados en el *panel de información administrativa*. Cada uno de ellos tiene un límite de caracteres que evitara que el usuario introduzca más de los válidos.

Figura A.2: Manual de la aplicación generación y visualización - Panel de información administrativa

2. A continuación el usuario debe definir la configuración general del examen: el número de versiones que tendrá, el número de preguntas, el número de respuestas por pregunta y si es un test *Oficial* o de *Autocorrección*.

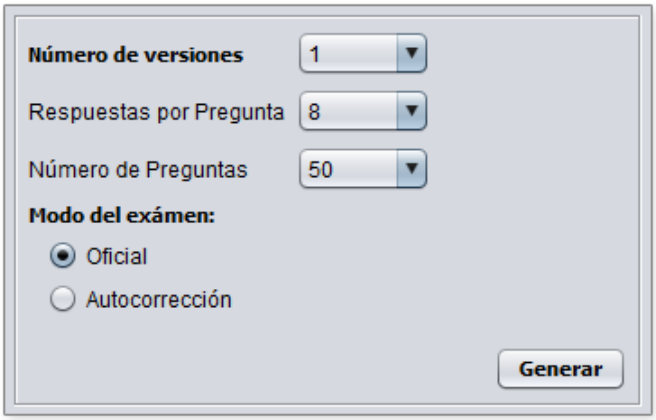

Figura A.3: Manual de la aplicación generación y visualización - Panel de configuración de test

2.1 Si el test es *Oficial*, se mostrará al usuario el *panel de datos de alumnos*, donde podrá introducir en formato *CSV* los códigos *NIA* de todos los alumnos a mano o podrá cargar un fichero con dichos datos pulsando el botón *Cargar CSV*,

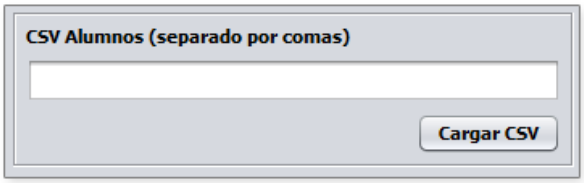

Figura A.4: Manual de la aplicación generación y visualización - Panel de datos de alumnos

3. Una vez definido el test, el usuario debe seleccionar las respuestas de cada pregunta del mismo y para todas sus versiones utilizando el *panel de preguntas y respuestas*.

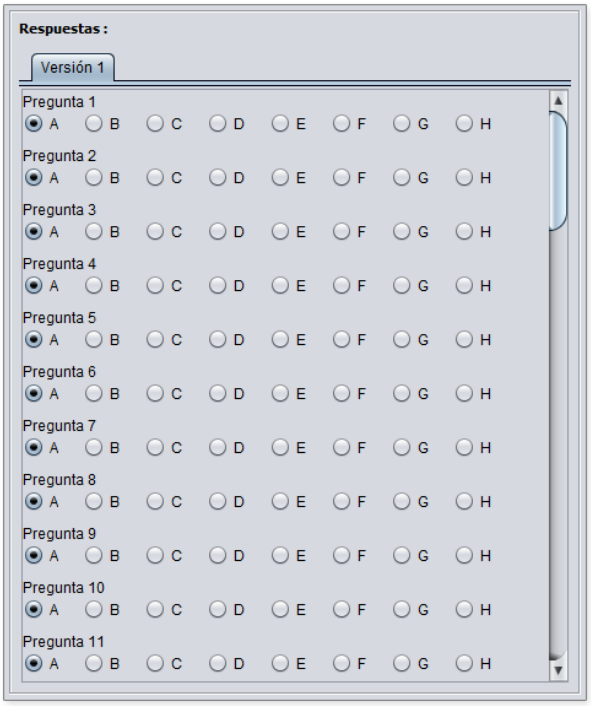

Figura A.5: Manual de la aplicación generación y visualización - Panel de preguntas y respuestas

4. Por último, cuando el usuario desee generar el examen, pulsará el botón *Generar* situado en el borde inferior derecho del *panel de configuración del test* y, en la ventana del explorador de ficheros que se muestra, elegir la carpeta donde desea guardarlo.

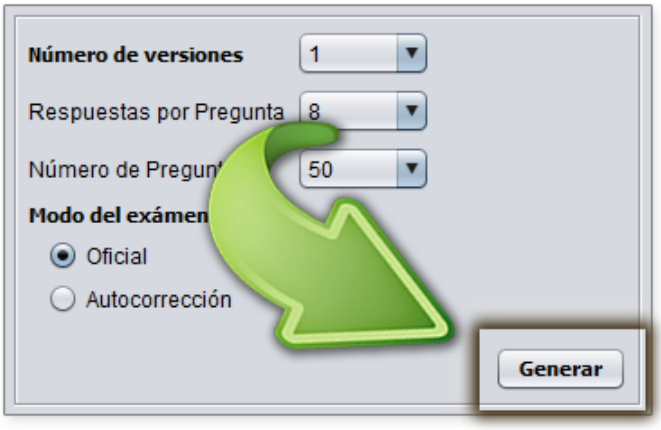

Figura A.6: Manual de la aplicación generación y visualización - Botón *Generar*

Una vez generado un test, si el usuario desea crear otro nuevo y vaciar los campos, pulsará el botón de la barra de opciones *Archivo* y posteriormente el botón *Nuevo*.

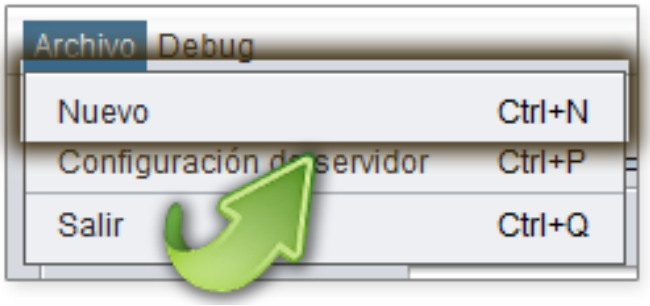

Figura A.7: Manual de la aplicación generación y visualización - Botón *Nuevo*

A.3.3. Visualización de un test

Una vez generado y escaneado algún test *Oficial*, el usuario puede visualizarlos desde esta aplicación. Para ello se debe seleccionar la pestaña *Visor*.

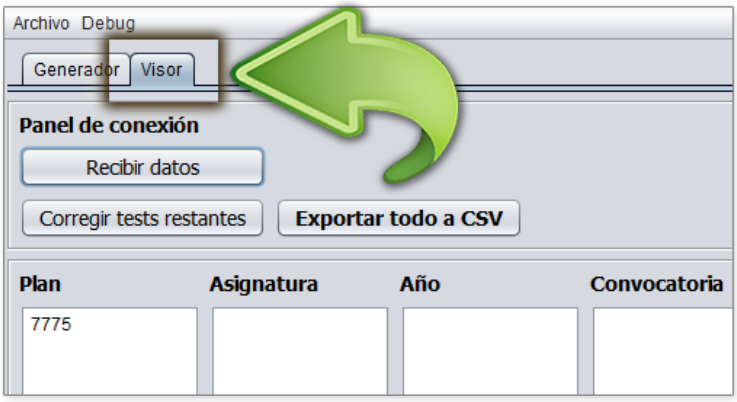

Figura A.8: Manual de la aplicación generación y visualización - Seleccionar *Visor*

Dentro de la pestaña, se deben seguir los siguientes pasos:

1. Primeramente se pulsará el botón *Recibir datos* situado en el *panel de conexión*. Se enviará entonces una petición al servidor para recibir todos los tests almacenados. Una vez termine la operación se mostrará un panel al usuario y se añadirán los planes disponibles al *panel de filtrado*.

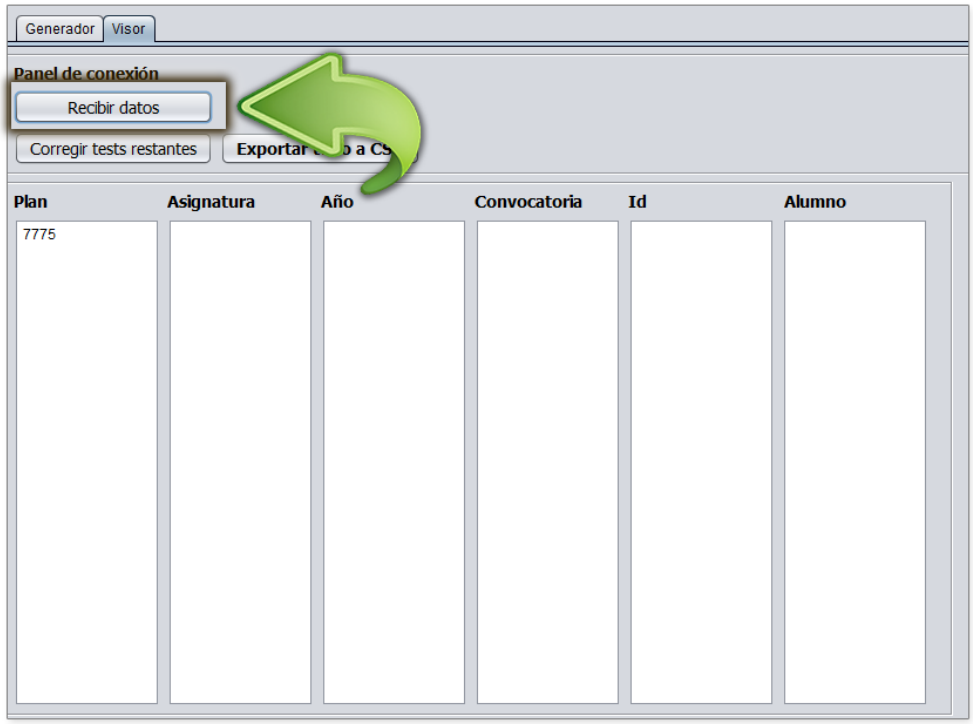

Figura A.9: Manual de la aplicación generación y visualización - Botón *Recibir datos*

2. Con los datos recibidos, el usuario deberá ir filtrando los tests disponibles utilizando los campos disponibles en panel de filtrado. Al seleccionar el examen de un alumno concreto, se mostrará el *panel de corrección* y el *panel de datos del test*. El primero indica si el examen está o no corregido, dando la opción de corregirlo si no lo estuviera. El segundo presenta los datos de generación del test.

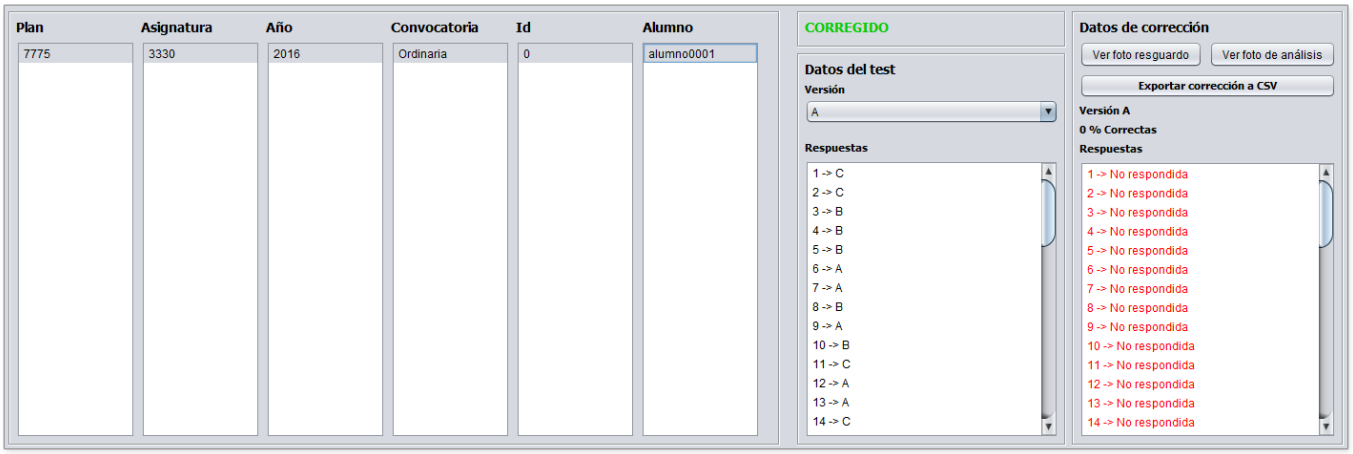

Figura A.10: Manual de la aplicación generación y visualización - Test seleccionado

A partir de aquí surgen varias posibilidades:

- Si el test no está corregido, el usuario podrá pulsar el botón *Corregir* situado en el *panel de correción*. Éste pide corregir el examen al servidor y mostrará el resultado den el *panel de datos de corrección* una vez reciba la respuesta.
- Si el test está corregido, se mostrará el *panel de datos de corrección*. En él, el usuario podrá visualizar las respuestas y la versión seleccionada por el alumno, así como el porcentaje de respuestas correctas. En este panel se muestran tres nuevas opciones.
  - El botón *Ver foto resguardo* pedirá al servidor la foto de resguardo. Una vez recibida, la guardará en el directorio raíz de la aplicación y la mostrará al usuario con el programa de visualización de imágenes por defecto del sistema operativo.
  - El botón *Ver foto de análisis* pedirá al servidor la imagen utilizada para el análisis. Una vez recibida, la guardará en el directorio raíz de la aplicación y la mostrará al usuario con el programa de visualización de imágenes por defecto del sistema operativo.
  - El botón *Exportar corrección a CSV* permitirá exportar los datos de la corrección a un fichero *CSV*. Una vez pulsado, mostrará el panel de ficheros para seleccionar el fichero a guardar.

A.3.4. Corrección de todos los tests almacenados en el servidor

Además de corregir un test concreto, el usuario puede pedir al servidor la corrección de todos los exámenes no corregidos. Situado en la pestaña *Visor* igual que en el caso anterior, se debe pulsar el botón *Corregir test restantes* situado en el *panel de conexión*. Al hacerlo, se enviará una petición al servidor; la respuesta indicará el número total de exámenes corregidos.

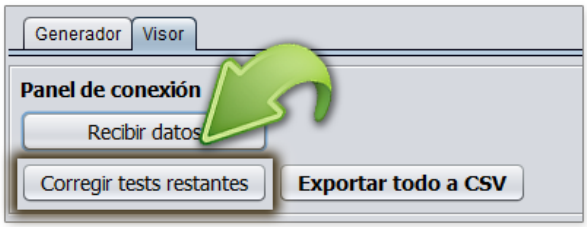

Figura A.11: Manual de la aplicación generación y visualización - Botón *Corregir tests restantes*

Si posteriormente el usuario desea visualizar las correcciones, deberá seguir los pasos indicados en la sección *Visualización de un test*.

A.3.5. Exportar todos los tests corregidos

Si el usuario desea exportar la información de todas las correcciones disponibles en un único fichero *CSV*, debe estar situado en la pestaña *Visor* y posteriormente pulsar el botón *Recibir datos* para recoger todos los exámenes.

Con los datos recibidos, se mostrará el botón *Exportar todo a CSV* en el *panel de conexión*. Tras se pulsado, se recogerán los datos de todas las correcciones y se pedirá al usuario que elija donde guardar el fichero.

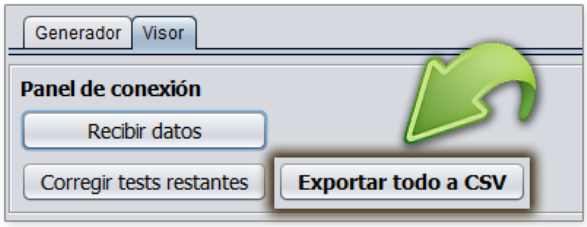

Figura A.12: Manual de la aplicación generación y visualización - Botón *Exportar todo a CSV*

Anexo B

Manual de usuario de la aplicación servidor

B.1. Requerimientos mínimos

El software requiere lo básico necesario para ejecutar una aplicación *Java* y un servidor *GlassFish*:

- Equipo con sistema operativo Windows con arquitectura x64. Dado que la aplicación está escrita en *Java* y al naturaleza multiplataforma de *OpenCV*, podrían utilizarse otros sistemas operativos, pero la distribución actual soporta sistemas *Windows x64*.
- Tener instalado en el sistema *Java 8*, ya sea la versión de *Oracle*[1] o el *OpenJDK*[2]. No obstante, en este caso para evitar problemas de compatibilidad se recomienda utilizar la versión de *Oracle*.
- Tener instalado *GlassFish* en su versión *4.1*. Debe ser exactamente esta versión porque las posteriores incluyen módulos diferentes que dificultan la ejecución de la aplicación.

B.2. Instalación

El software distribuido incluye la aplicación servidor en un fichero *WAR* y la biblioteca nativa de *OpenCV* para *Windows x64* - fichero *opencv_java310.dll* -.

Primeramente se debe copiar la biblioteca en la ruta dónde se encuentran las bibliotecas del sistema, que habitualmente será en el disco duro donde encuentre instalado el sistema operativo y a partir de ahí en la ruta "*Windows/System32*".

Después, se puede desplegar la aplicación en el servidor *GlassFish*. El usuario lo puede realizar de varias maneras, pero como línea general los pasos serían los siguientes:

1. Copiar el fichero *war* en el directorio *bin* donde se encuentra el servidor *GlassFish*.
2. Para lanzarlo en el dominio por defecto *domain1*, se deben ejecutar los siguientes comandos en una consola situada en la carpeta *bin* del servidor:

```
asadmin start-domain
asadmin deploy nombre_del_ejecutable.war
```

[1] `https://www.java.com/es/download/`
[2] `http://openjdk.java.net/`

A partir de este momento la dirección del servidor sería:

```
http://dirección_del_servidor:puerto/TestServer/webresources/servertests
```

3. Si se desea parar el servidor, se deberá ejecutar el siguiente comando:

```
asadmin stop-domain
```

Como último apunte, si se desea acceder a los ficheros relativos a la base de datos y servidor, éstos se encuentran en la carpeta *config* que se puede encontrar en la carpeta del dominio donde se ha lanzado la aplicación. Tomando como referencia el dominio por defecto *domain1*, la ruta sería la siguiente:

```
glassfish\domains\domain1\config
```

Anexo C

Manual de usuario de la aplicación de escaneado de tests

C.1. Requerimientos mínimos

Se requiere un dispositivo móvil con las siguientes características:

- Sistema operativo *Android 5.0* o superior.
- Cámara trasera.
- Opcional pero recomendable la existencia de flash en la cámara.

C.2. Instalación

La aplicación se distribuye en un fichero en formato *APK*, que es el estándar para *Android*. Simplemente se debe instalar en el dispositivo mediante cualquier de los métodos habituales: copiando el fichero al almacenamiento del dispositivo e instalándolo, utilizando comandos en el *ADB*[1] - *Android Debug Bridge* -, etc.

Cabe destacar que requiere los siguientes permisos:

- ***WRITE_EXTERNAL_STORAGE***. Permiso de escritura en el almacenamiento externo para guardar temporalmente las fotos tomadas.
- ***CAMERA***. Permiso de acceso a la cámara, componente básico de la aplicación.
- ***FLASHLIGHT***. Permiso de acceso al flash.
- ***INTERNET***. Permiso para abrir sockets de red, para el envío de los datos al servidor.
- ***ACCESS_NETWORK_STATE***. Permiso para acceder a información sobre al red, para el envío de los datos al servidor.
- ***ACCESS_WIFI_STATE***. Permiso para acceder a información relativa a redes *Wi-Fi*, para el envío de los datos al servidor.

[1] `https://developer.android.com/studio/command-line/adb.html`

C.3. Uso básico

Esta aplicación provee una interfaz de uso sencilla, ya que está enfocada únicamente en el escaneo de exámenes. Una vez que el usuario haya completado el teste que desea reconocer, debe seguir los siguientes pasos:

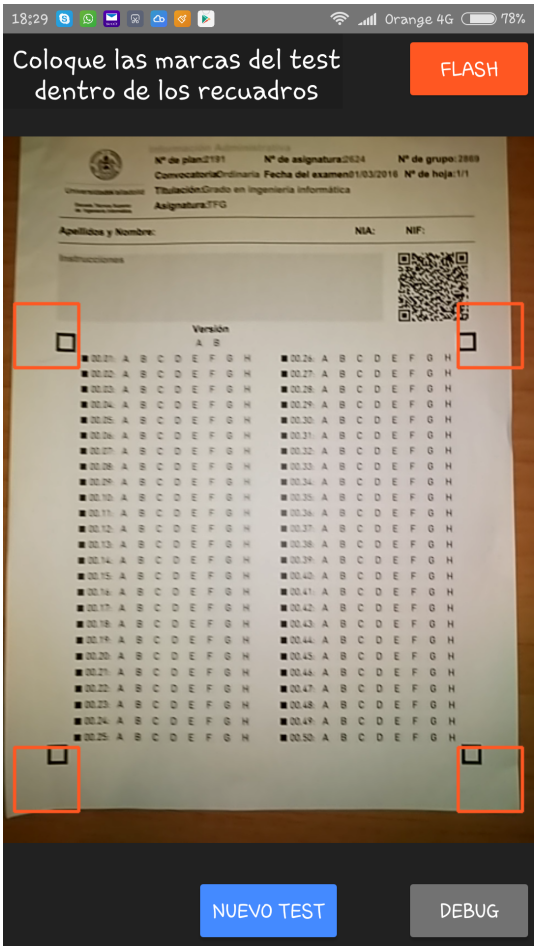

Figura C.1: Manual de la aplicación de escaneo - Pantalla principal

1. Si el test está impreso en papel, se debe en una superficie lo más plana posible y con una iluminación general lo más fuerte posible. Este último requisito no indica que deba tener una luz potente y directa apuntando al examen, simplemente señala que cuanta más luz pueda captar la cámara, más sencillo le será al programa reconocer el test. Por otra parte, si se encuentran problemas durante el reconocimiento y la luz no se puede aumentar externamente, se puede utilizar el flash de la cámara - si ésta dispone de él -.

2. El usuario colocará el móvil de tal manera que la imagen de la cámara sitúa las esquinas de la zona de test en los recuadros indicados en la pantalla de la aplicación. Esto se realizará siempre con el código QR quedando situado en al parte superior derecha de la imagen, y se recomienda colocar el dispositivo lo más paralelo posible al test para evitar distorsiones.

3. El programa empezará a analizar el test. Una vez termine, el resultado variará según el tipo de test:

   - Si es *Oficial*, se indicará por pantalla que se están enviando los datos al servidor. Una vez termine, se le mostrará al usuario y habrá terminado el reconocimiento del test.
   - si es de *Autocorrección*, se mostrará en pantalla un panel con los resultados de la corrección, la versión seleccionada y el porcentaje de respuestas acertadas.

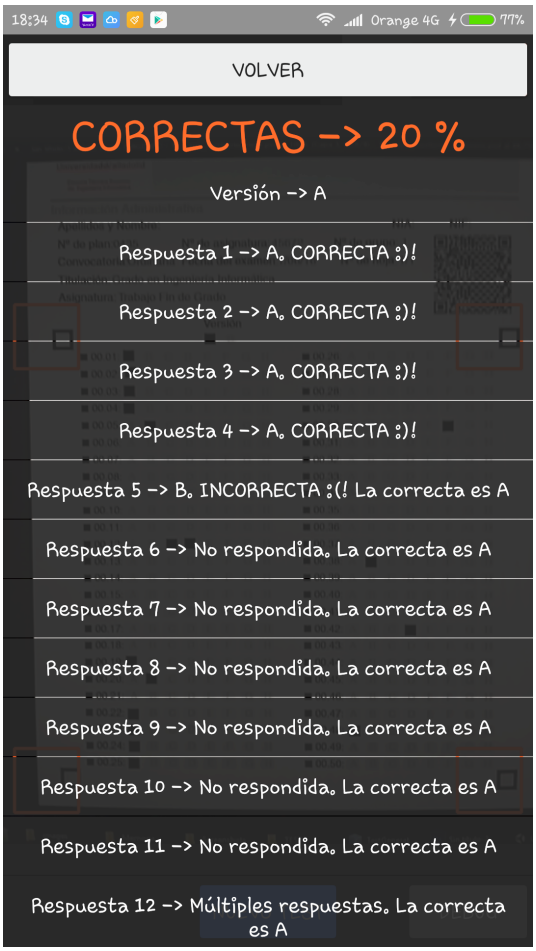

Figura C.2: Manual de la aplicación de escaneo - Resultados

Por otra parte, se incluyen los siguientes botones:

- **Nuevo Test**. Una vez reconocido un test, si se desea realizar otro escaneo se deberá pulsar este botón.
- **Flash**. Permite encender el flash. Si el dispositivo no dispone de él, este botón no se mostrará.

Anexo D

Información incluida en el soporte digital

D.1. Contenidos

En este documento se incluye en formato de lista el contenido del soporte digital entregado junto a la memoria del trabajo. El listado indica de manera jerárquica los directorios y ficheros más importantes.

- ***memoria.pdf***. Fichero *PDF* con la memoria en formato *A4*.
- ***contenido.pdf***. Fichero que contiene la información del soporte digital.
- **Aplicaciones**. Dentro de esta carpeta se encuentran las aplicaciones listas para ser instaladas y utilizadas.
  - **Aplicación de generación y visualización**. En el interior de esta carpeta se encuentra un fichero comprimido en formato *ZIP* con la aplicación de generación y visualización de tests.
  - **Aplicación de escaneado**. En el interior de esta carpeta se encuentra el fichero en formato *APK* con la aplicación de escaneado de tests. Debe ser instalado en un dispositivo *Android 5.0* o superior como indica el manual.
  - **Aplicación servidor**. En el interior de esta carpeta se encuentra la aplicación servidor en formato *WAR* y el fichero relativo a la biblioteca *OpenCV*. En el manual de esta aplicación se indica cómo se debe realizar la instalación.
- **Código fuente**. Dentro de esta carpeta se incluye una carpeta para el proyecto de cada una de las aplicaciones y una carpeta más que incluye el código fuente utilizado para crear la memoria.
  - **Aplicación de generación y visualización**. En su interior se encuentra una carpeta *TestGeneratorViewer* con el proyecto de la aplicación. Para abrirlo se debe utilizar el entorno de desarrollo *NetBeans* - concretamente se ha realizado utilizando la versión *8.0.2* -.
  - **Aplicación de escaneado**. En su interior se encuentra una carpeta *TestRecognizer* con el proyecto de la aplicación. Para abrirlo se debe utilizar el entorno de desarrollo *Eclipse* - concretamente se ha realizado utilizando la versión *4.5.0* -. Para su correcto funcionamiento se debe tener configurado e instalado el *SDK* y *NDK* de *Android*.
  - **Aplicación servidor**. En su interior se encuentra una carpeta *TestServer* con el proyecto de la aplicación. Para abrirlo se debe utilizar el entorno de desarrollo *NetBeans* - concretamente se ha realizado utilizando la versión *8.0.2* -.
  - **Memoria**. En su interior se encuentran todos los ficheros fuente LaTeXy las imágenes utilizadas para la creación de la memoria. Se ha trabajo y compilado utilizando el entorno de desarrollo *TeXstudio* y al instalación *MiKTeX*.

- **Diagramas**. En esta carpeta se incluyen las imágenes de todos los diagramas utilizados para la concepción del proyecto y su memoria.
  - **Análisis**. Incluye los diagramas del análisis del problema.
  - **Solución**. Incluye los diagramas de la solución propuesta. Al ser un número elevado, están separados en carpetas.
    - **Aplicación de generación y visualización**. Incluye los diagramas de la aplicación de generación y visualización.
    - **Aplicación de escaneado**. Incluye los diagramas de la aplicación de escaneado.
    - **Aplicación servidor**. Incluye los diagramas de la aplicación servidor.
    - **Diagramas de despliegue**. Incluye los diagramas de despliegue teniendo en cuenta las dos posibles variaciones de la aplicación servidor.
    - **Diagramas de secuencia**. Incluye los diagramas de secuencia relativos a los casos de uso detectados.
- **Manuales**. Incluye los manuales de cada aplicación en ficheros *PDF* y formato *A4*.