Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de

Ingenieros Informáticos

TRABAJO FIN DE GRADO

Extracción de datos de la herramienta SonarQube

Autor: Sergio González Martín

Director: Ángel Lucas González Martínez

MADRID, MAYO 2019

RESUMEN EN CASTELLANO

El trabajo ha consistido en el desarrollo de una aplicación que sea capaz de extraer los datos de los proyectos de la plataforma SonarQube [1] instalada en la Escuela mediante su API [2] y guardarlos en ficheros estructurados. Estos proyectos tendrán asociada una raíz, por ejemplo Entregable1, que identificará a todas las prácticas de alumnos pertenecientes a ese entregable. Como objetivo secundario, se propuso realizar una interfaz que mostrase esta información extraída en pantalla.

El objetivo de esta aplicación es reducir el tiempo que se tarda habitualmente en darles retroalimentación a los alumnos sobre las prácticas que entregan. Además, los profesores utilizan la herramienta SonarQube para valorar el código del alumnado, gracias a la cantidad de información en forma de métricas que esta proporciona. También pueden definir los denominados *quality gate*, que son umbrales de calidad que marcan el límite para que el valor de una métrica sea erróneo o válido.

Para el desarrollo del trabajo ha sido necesario seleccionar las métricas a extraer (por ejemplo, las líneas de código, los *code smell* o los *bugs*, entre otras) y decidir en qué tipo de formato iba a almacenarse la información. Se acordó con el tutor que sería en CSV, pero que, en el futuro, esto debería de poder ampliarse para abarcar otros formatos o la inserción en una base de datos. También es importante mencionar que fue necesario crear unos modelos de datos para poder guardar en memoria la información extraída.

La aplicación ha sido desarrollada en Java [3], haciendo uso de dos librerías de uso libre: *org.json* [4] para procesar datos en formato JSON [5] y *opencsv* [6] para generar ficheros con el formato acordado. Por otro lado, se ha utilizado JavaFX [7] para desarrollar la interfaz de la aplicación.

Por tanto, la aplicación cuenta con dos partes bien diferenciadas: por un lado el *backend*, que se encarga de comunicarse con la API de SonarQube, haciendo uso del *token* que denota los permisos que tiene el usuario para acceder a cierta información. Una vez realizadas las peticiones y obtenidos los datos sobre los proyectos que se quieren analizar, se generan tres ficheros: un CSV para las métricas, otro para los *code smells* y un fichero JSON para los *quality gate*. Además, se almacenan en objetos en memoria los datos, haciendo uso de los modelos previamente mencionados.

Y por otro lado el *frontend* o interfaz, que recoge información introducida por el usuario (como la raíz de los proyectos), para, a partir de ella, realizar la consecuente llamada al *backend* y poder mostrar la información obtenida en pantalla, en forma de tabla. La interfaz cuenta con un menú de opciones que permite visualizar solo las métricas, solo los *code smells*, solo los *quality gate* o ver un listado de todos los proyectos y ver su información individual.

RESUMEN EN INGLÉS

The project consisted on developing an application which is able to extract information from the projects stored within the SonarQube platform installed in the Faculty via its API. After the extraction, this information should be stored in some kind of structured files. The SonarQube's projects are identified by a root key, for example Entregable1. As a secondary objective, the application should be able to show this information on screen on a simple and clean way.

The main objective of this application is to reduce the time it usually takes to give the students feedback on their exercises. What is more, the professors usually use SonarQube in order to rate these exercises, thanks to the great amount of information this tool provides. Also, professors are able to set up a quality gate, which represents the threshold a metric should respect in order to be considered valid.

For the successful development of the application, the student had to select which metrics to extract (number of lines of code, bugs, code smells, among others) and to decide which kind of structured file the application would use. It was agreed with the tutor that the format would be CSV for this project, but the application should be flexible enough to admit more formats or insertions in a database in the near future. Furthermore, it was needed to define some data models in order to store the extracted information in memory.

The application has been developed with Java, using some open source libraries such as: org.json for JSON manipulation and opencsv for CSV files generation. For the frontend part of the application, JavaFX was used.

Therefore, the application is made up of two parts: on the one hand the backend, which is the one in charge of communicating with SonarQube's API, putting the access token to good use. This token grants the user the permissions needed to access the information stored on the platform. When all the requests are done, three files are generated (a CSV file for metrics, another one for code smells and a JSON file for quality gates). What is more, the extracted data is stored in memory, using structures like maps and the data models previously mentioned.

On the other hand the frontend, which is in charge of gathering user input (such as the root of the projects) in order to invoke the backend. This call to the backend allows the frontend to show the information on screen, using a table format. The interface is composed of an option menu which allows the user to visualize just the metrics, the code smells, the quality gates or a list of all the projects involved in the extraction. The user is also able to select one project and visualize on screen its individual information.

Índice

1. INTRODUCCIÓN 1

2. ESTADO DEL ARTE 1

3. ARQUITECTURA DE LA PLATAFORMA SONARQUBE 3

4. DESCRIPCIÓN DEL TRABAJO 4

5. METODOLOGÍA 5

   5.1. ORGANIZACIÓN DEL TRABAJO 5

   5.2. HERRAMIENTAS UTILIZADAS 6

6. MÉTRICAS DE SONARQUBE 7

   6.1. LISTA DE MÉTRICAS 7

   6.2. MÉTRICAS SELECCIONADAS 8

   6.3. INFORME DE SONARQUBE 9

7. DOCUMENTACIÓN DE LA API 10

8. *BACKEND* 13

   8.1. PROTOTIPO INICIAL 14

   8.2. CENTRAL 15

   8.3. EXTRACTOR 16

   8.4. ESCRITORCSV 18

   8.5. MODELOS DE DATOS 19

9. *FRONTEND* 20

   9.1. DISEÑO DE LA INTERFAZ 23

   9.2. APLICACIÓN 28

   9.3. CONTROLADORES 29

   9.4. MODELOS DE DATOS 30

   9.5. CONTEXTO 31

10. LÍNEAS FUTURAS 31

11. OBJETIVOS ALCANZADOS 32

12. CONCLUSIONES 33

REFERENCIAS 34

ANEXO: MANUAL DE LA APLICACIÓN 36

1. INTRODUCCIÓN

El presente documento es la memoria final del Trabajo de Fin de Grado Extracción de Datos de la herramienta SonarQube [1] En él se detalla el proceso seguido para realizar el trabajo, así como los principales problemas encontrados.

También se comentará la organización seguida con el tutor y los objetivos que se han alcanzado.

Por otro lado, se detallarán todas las tecnologías utilizadas, y para qué se han utilizado. Junto a esto se detallarán las clases implementadas, sus usos y sus respectivos diagramas.

Por último, se evaluarán qué objetivos se han cumplido y cuáles no, en caso de que los haya, así como qué tanto se ha ajustado el trabajo al plan de trabajo elaborado al principio del curso.

2. ESTADO DEL ARTE

Evaluar la calidad del código es una de las principales preocupaciones de cualquier empresa en su día a día o, al menos, debería serlo. El problema surge cuando esta no se tiene en cuenta. En ocasiones se dejan pasar fallos sobradamente conocidos y, cuando el cliente se queja o pide una nueva funcionalidad, es cuando toca enfrentarse a ellos.

La calidad del *software* se puede ver desde dos perspectivas, de acuerdo con lo que dice Mohammad Nadeem en su artículo *An Introduction to Static Code Analysis* [8], desde la del usuario y desde la del desarrollador, ambos experimentan la calidad de formas diferentes. El primero desde un punto de vista de funcionalidad del producto y el segundo desde el punto de vista estructural, de diseño.

Entonces, podemos definir la calidad del *software* como una medida cuantitativa, esto es, objetiva, que nos indica cómo se comporta nuestro programa en diferentes ámbitos mediante la medición de ciertas características o atributos [8]. Algunos ejemplos de atributos que veremos a lo largo de la memoria podrían ser los *bugs* o la complejidad del código.

Para ayudar en esta medición existen diversas herramientas como pueden ser Squale [9] o SonarQube. En el caso del presente TFG, se utiliza SonarQube por ser la que está instalada en el centro de estudios de la Facultad de Informática.

SonarQube es una herramienta muy completa que basa su visión de la calidad del *software* en siete pilares fundamentales, definidos por sus creadores (ver Figura 1): *bugs*,

estándares de programación, duplicación, pruebas unitarias, complejidad, falta o exceso de comentarios y el diseño y arquitectura. [10]

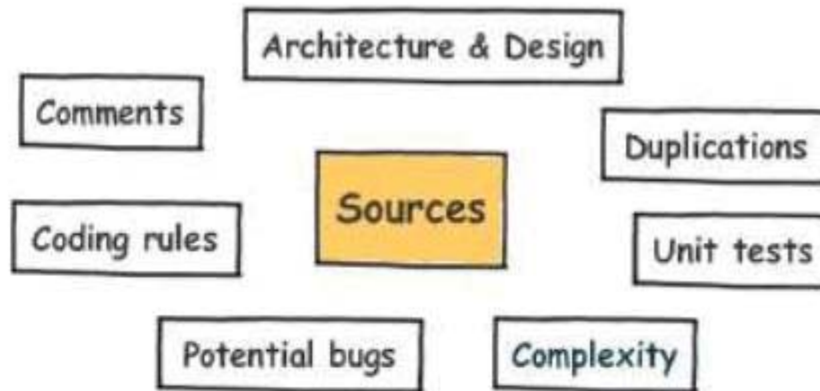


Figura 1: Los siete pilares de la calidad del software. Fuente: [11]

Como veremos en la sección de arquitectura de SonarQube, la herramienta cuenta con un servidor web sobre el que se realizarán peticiones para consumir sus servicios y obtener información. Esto puede verse ilustrado en la Figura 2 de manera genérica, en la que un Cliente solicita mediante una petición *GET* una información y recibe una respuesta en un formato de intercambio de datos.

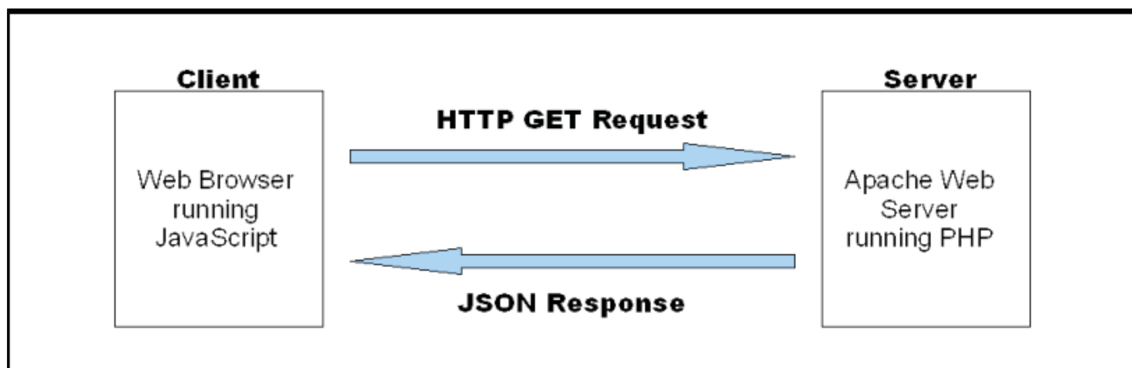


Figura 2: Petición a un servidor Web y respuesta JSON. Fuente: [12]

Actualmente los formatos más utilizados para el intercambio de información son JSON [5] y XML [13]. Ambos tienen sus ventajas y desventajas, por ejemplo JSON es muy compacto, sencillo y además compatible con JavaScript [14]. Mientras que XML es más restrictivo y tiene soporte para esquemas y espacios de nombres.

Debido a este uso tan extendido, SonarQube utiliza el formato JSON para transmitir sus datos, como veremos en las secciones posteriores.

Dado que JSON está muy extendido, existen numerosas herramientas y librerías para procesarlo. Una de las más conocidas en Java [3] es *org.json* [4], desarrollada por Sean Leary. Esta librería es de libre distribución y se puede utilizar a partir de Java 1.6. Permite la fácil manipulación de objetos en este formato e incluso su conversión a XML.

3. ARQUITECTURA DE LA PLATAFORMA SONARQUBE

De acuerdo con la Figura 3, la plataforma SonarQube está formada por los siguientes componentes:

- Un servidor (número 1) que inicia tres procesos: un servidor web que permite configurar la instancia de SonarQube, un servidor para almacenar copias de seguridad y otro servidor que procesa los análisis y se encarga de salvarlos en una base de datos.
- La base de datos de SonarQube (número 2), donde se almacenan la configuración y los análisis.
- *Plugins* (número 3) que proporcionan información adicional y que el usuario puede elegir instalar o no. En el caso del SonarQube instalado en la Facultad hay un *plugin* de *code smells* instalado, por ejemplo.
- *SonarScanners* (número 4), que permiten realizar los análisis en un ordenador local o servidor.

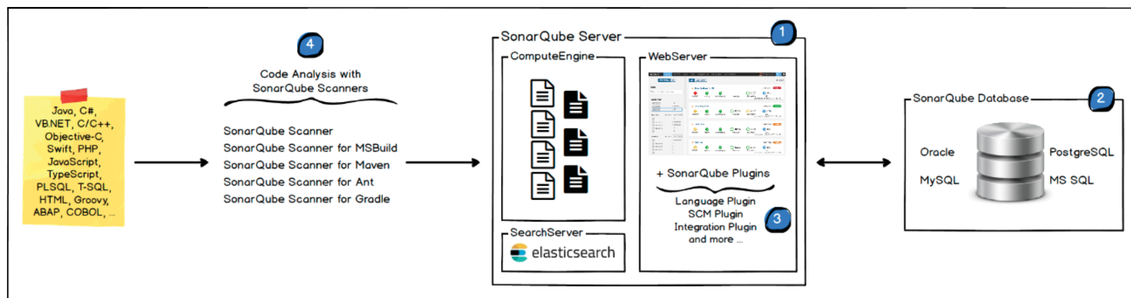


Figura 3: Arquitectura de la plataforma SonarQube [15]

Por otro lado, SonarQube proporciona también una Web API [2] que permite acceder a la información sobre los proyectos y sobre la instancia de SonarQube instalada. Sigue la arquitectura REST [16], ampliamente utilizada en la web.

4. DESCRIPCIÓN DEL TRABAJO

El trabajo consiste en extraer datos de la herramienta SonarQube. Esta es utilizada para dar una valoración del código de los alumnos, que bien pueden subir su trabajo al sistema de entrega proporcionado por el profesorado o subirlo a Moodle y desde ahí pasar a SonarQube. El hecho de contar con una aplicación que extraiga cierta información, la almacene en ficheros ordenados y permita visualizarla fácilmente, puede ser muy valioso a la hora de reducir enormemente el tiempo que se tarda en darle al alumno la retroalimentación o corrección de sus proyectos.

Para cada entrega subida y analizada, SonarQube se encarga de asignarle una *key* o identificador, que identificará al proyecto al ser única. El objetivo principal del trabajo es recolectar las métricas generadas por SonarQube, algunas provenientes de *plugins* externos como el de los *code smells*, y plasmarlas en un fichero CSV, formato que permita ser manipulado para la inclusión en base de datos, leído (puede abrirse en Excel), etc.

Estas métricas han sido seleccionadas y evaluadas individualmente para reducir su número a un conjunto de las más útiles o importantes. Aun así, esto no quiere decir que no pueda ampliarse la aplicación, ya que en cualquier momento pueden añadirse métricas nuevas fácilmente, dependiendo de las necesidades del momento.

Además, para extraer toda esta información ha sido necesario realizar un estudio y documentar la Web API de SonarQube, que posee bastantes métodos interesantes y útiles, aunque no se hayan utilizado todos.

Para realizar todo esto, ha sido necesario conocer Java y JSON principalmente, así como el funcionamiento de los servicios web mencionados. Esto implica un conocimiento del protocolo Http mínimo, por ejemplo, para evaluar los códigos de respuesta.

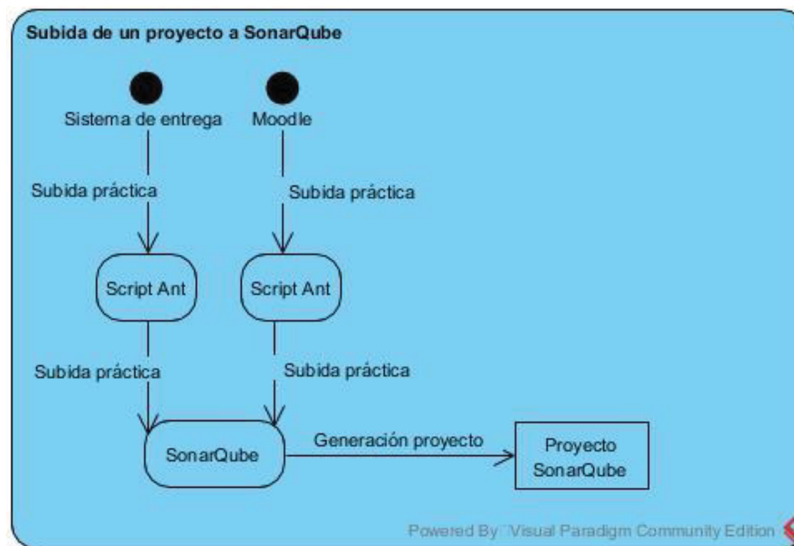


Figura 4: Subida de un proyecto a SonarQube

Es importante realizar un inciso aquí, antes de avanzar más, para evitar confundir algunos términos clave. Como se puede observar en la Figura 4, cuando se sube a SonarQube la práctica o ejercicio de un alumno, mediante cualquiera de los dos métodos mencionados, SonarQube crea un proyecto que contiene toda la información de esta práctica. Además, independientemente del método, se ejecuta un *script ant* [17] para realizar la subida al sistema de SonarQube.

Esto quiere decir que, cuando hablemos de proyecto nos estaremos refiriendo al proyecto generado por SonarQube y, cuando hablemos de práctica o ejercicio, a la entrega del alumno. Por otro lado, cada proyecto lleva asociada una raíz de acuerdo a la categoría del ejercicio, por ejemplo, una raíz podría ser Práctica1, que haría referencia a todos los ejercicios de alumnos correspondientes a dicha práctica.

5. METODOLOGÍA

Este apartado ha sido dividido en dos secciones en las que se mencionarán las herramientas o tecnologías utilizadas para el desarrollo de la aplicación y de su documentación, así como de la organización y el flujo de trabajo a lo largo del curso.

5.1. ORGANIZACIÓN DEL TRABAJO

Para el desarrollo correcto del trabajo, se han establecido algunas pautas a seguir, que han ayudado a organizar todas las tareas y objetivos a realizar. Gracias a esto, se han cumplido los plazos fijados. En este apartado se detallarán los procedimientos seguidos, tanto para organización como para el desarrollo.

Al principio del curso se estableció realizar reuniones cada dos semanas con el tutor, de modo que este fuese informado constantemente de cómo iba avanzando el proyecto. Para acompañar esto, se enviaba un informe semanal cada viernes de la semana, que contenía los avances en el proyecto.

En cuanto a la codificación de la aplicación, el desarrollo se ha basado en prototipos. Esto quiere decir que se han ido diseñando módulos de la aplicación aislados y, posteriormente, han sido integrados. Por ejemplo: primero se desarrolló la parte de consultas a la API de la aplicación, posteriormente se comunicaron las salidas de estas consultas con una clase central que las redirigiría a otra clase para un procesamiento más extenso de los datos.

Una parte importante del desarrollo del trabajo, en general, no solo para la parte de codificación, es que la documentación para la memoria final se ha ido generando a lo largo del curso, por tanto, lo primero que se hizo fue realizar un documento con las métricas a extraer, por ejemplo.

5.2. HERRAMIENTAS UTILIZADAS

En cuanto a las herramientas, no se utilizó un repositorio como tal, si no, una carpeta compartida en *OneDrive* de Microsoft [18], donde se iba colgando la documentación y el código generado periódicamente. Como apunte, decir que se ha utilizado el sistema SonarQube de la escuela principalmente, que ya dispone de prácticas de alumnos y al que sólo se puede acceder con la VPN.

Mencionar también que se ha realizado utilizando Java 8 y Eclipse [19] (cualquier versión que admita *e(fx)clipse* [20] debería valer), por lo que se utilizan expresiones lambda, *streams*, etc. Por último, el proyecto ha sido probado en Ubuntu 18 y Windows 10.

Además, se han realizado diagramas de clases para ayudar a la comprensión del problema, y velar por un buen diseño de la aplicación. Estos diagramas han sido realizados con la herramienta *Visual Paradigm* [21] en su versión gratuita. Los diseños de la interfaz mostrados en el primer subapartado del *frontend* fueron realizados con *draw.io* [22], ya que *Visual Paradigm* no permitía en su versión gratuita la creación de interfaces.

Para el desarrollo de la interfaz, se ha utilizado JavaFX [7], con su herramienta adjunta *Scene Builder* [23], como se explicará más adelante.

6. MÉTRICAS DE SONARQUBE

SonarQube proporciona una amplia gama de métricas, pero no todas son igual de útiles para el propósito descrito en apartados anteriores. Por este motivo, en este apartado se explicará detalladamente qué es cada métrica y cuáles se han escogido, así como su utilidad de cara a este proyecto. Al final del apartado, se mostrará un informe a modo de ejemplo.

6.1. LISTA DE MÉTRICAS

En este apartado se detallan las métricas que han sido consideradas más importantes y a las que se puede acceder. El método de acceso es igual para todas ellas: añadir un *queryParam* (un parámetro insertado en la URL que permite filtrar los resultados) denominado "*metricKeys*" y separar mediante comas el identificador de cada métrica. También es necesario indicar mediante otro *queryParam* llamado "*componentKey*" el proyecto del que se obtendrá la métrica.

http://sonar.fi.upm.es:9000/api/measures/component&componentKey=PROYECTO&metricKeys=MÉTRICA1,MÉTRICA2

1 2 3

Figura 5: Ejemplo de URL genérica

Como se puede observar en la Figura 5, una URL de SonarQube se compone del *endpoint* del api (1), la ruta (2) y los *queryParam* mencionados, con sus valores separados por comas (3). En la sección 7 del documento se explicarán más a fondo las rutas de la API.

Para obtener los identificadores, descripciones, nombres, etc, de cada métrica, se puede realizar una consulta GET al *endpoint*: "api/metrics/search". A continuación, se muestra una lista ordenada alfabéticamente de las métricas más importantes y una pequeña descripción.

- *alert_status*. Indica si el proyecto está dentro de los límites indicados en el *QualityGate* (QG). Si no se ha definido un QG para el proyecto, SonarQube le asignará uno por defecto. Ver *quality_gate_details* para más detalles.
- *bugs*. Número de errores de codificación que existen en el código.
- *classes*. Número de clases.
- *code_smells*. Número de *code smells* detectados en el código. Un *code smell* es una característica del código que puede indicar un problema de diseño o malas prácticas, aunque en principio no sea obvio.
- *cognitive_complexity*. Indica qué tan difícil es entender el código.
- *comment_lines*. Número de líneas comentadas.

- *complexity*. Suma uno a este valor cada vez que hay un cambio en el flujo del programa.
- *coverage*. Es una mezcla de *Condition Coverage* (cantidad de condiciones que se han seguido en diferentes estructuras de control de flujo, como un if, habiendo sido evaluadas a verdadero o falso durante las pruebas unitarias) y *Line Coverage* (densidad de líneas ejecutadas durante pruebas unitarias).
- *duplicated_blocks*. Cuántos bloques de código están duplicados. Esto varía según el lenguaje, en Java, por ejemplo, se considera duplicidad el hecho de que existan diez sentencias iguales seguidas, sin tener en cuenta comentarios, indentación o nombres de variables.
- *duplicated_lines_density*. Indica que porcentaje de líneas están duplicadas de acuerdo con las medidas que establece SonarQube para cada lenguaje.
- *functions*. Número de funciones.
- *ncloc*. Número de líneas de código del programa sin contar los comentarios.
- *quality_gate_details*. Indica el valor actual de una medida, el tipo de operación (mayor que, menor que) y a partir de qué valor se considera que hay un error o un *warning*.
- *sqale_rating*. Nota que se le da al proyecto en relación a *sqale_index*, que indica el tiempo necesario para solucionar los *code smells*.
- *tests*. Número de pruebas unitarias.
- *vulnerabilities*. Problemas que pueden comprometer la seguridad de la información y la integridad de la misma.

6.2. MÉTRICAS SELECCIONADAS

En este apartado se detallan las métricas que han sido extraídas y se aporta una justificación del porqué de su importancia. Algunas que podrían ser útiles en algún momento, como la densidad de líneas duplicadas, han sido descartadas porque esa funcionalidad no se está utilizando para evaluar el código de los alumnos.

También cabe destacar que, dependiendo del proyecto, los valores de algunas métricas podrían ser más útiles o menos o incluso variar los umbrales de valores aceptados dentro del QG.

Como último apunte antes de enumerarlas alfabéticamente, más adelante se detalla cómo se ha mantenido una estructura flexible de la aplicación, de modo que en cualquier momento se puedan añadir nuevas métricas a las extraídas.

- *alert_status*, *quality_gates_details*. Permiten controlar si un proyecto entra dentro de los límites marcados por los profesores o por el propio SonarQube.
- *bugs*. Error en el código del programa.

- *code_smells*. Útil para averiguar problemas que no se vean a simple vista en el código fuente del programa, como malas prácticas. Estos problemas también pueden ser causa de futuros bugs en el futuro.
- *complexity* y *cognitive_complexity*. Permiten ver qué tan compleja es la estructura del programa y si es difícil entenderlo debido a muchos cambios en el flujo de ejecución.
- *coverage*. Indica qué porcentaje del código ha sido cubierto por las pruebas que se han ejecutado sobre él. Cuanto mayor sea el valor, más código habrá sido ejecutado y, por tanto, más útiles habrán sido las pruebas. Esto, como se ha mencionado en el apartado anterior, dependerá de las estructuras de control de flujo.
- *ncloc*. Puede ser útil para poner en contexto el resto de métricas.
- *sqale_rating*. Para ver qué tan fácil de mantener es el código.

6.3. INFORME DE SONARQUBE

En este apartado, se muestra un informe de SonarQube como ejemplo (Figura 6). Este informe ha sido extraído directamente del SonarQube de la Escuela, se han tapado los datos relativos al alumno.

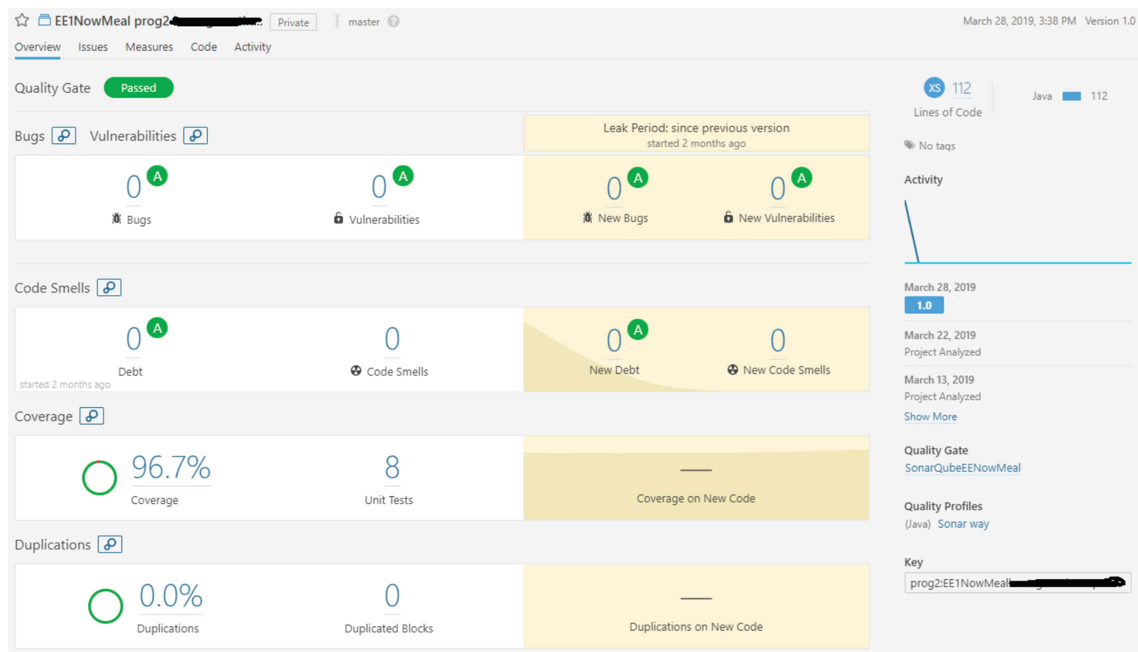


Figura 6: Informe de SonarQube

En la Figura 6 podemos observar algunas de las métricas que han sido mencionadas en los apartados 6.1 y 6.2. Entre otras, podemos ver los *bugs*, *code smells* o el nombre

del *quality gate* definido (SonarQubeEE1NowMeal). En la parte superior derecha de la Figura 6, podemos observar el número de líneas de código.

7. DOCUMENTACIÓN DE LA API

La API que proporciona los servicios web de SonarQube es bastante completa. En este apartado se comentarán los principales servicios utilizados y qué información se ha obtenido de ellos.

Antes de nada, mencionar que el *endpoint* donde descansa la API de SonarQube es http://sonar.fi.upm.es:9000/api/ en el caso de que estemos conectados a la VPN de la escuela y accedamos al SonarQube instalado en ella. Este SonarQube está en la versión 6.7.x, por lo que podría darse el caso de que haya habido algún cambio respecto a la versión más actual.

A continuación, se exponen las rutas utilizadas en la aplicación. Se encuentran enumeradas en orden de importancia. Primero se comentará como se obtienen todos los proyectos, luego como pueden filtrarse para extraer datos de uno sólo y, por último, como obtener métricas concretas. La información aparece estructurada en tablas, teniendo una columna para un ejemplo de URL y otra para el ejemplo de respuesta.

- projects/index. Permite extraer los proyectos asociados al usuario, en este caso identificado por *token*. Ver ejemplo en la Figura 7.

URL completa	Ejemplo
http://sonar.fi.upm.es:9000/api/projects/index	{ "id": "5035", "k": "org.jenkins-ci.plugins:sonar", "nm": "Jenkins Sonar Plugin", "sc": "PRJ", "qu": "TRK" }

Figura 7: Tabla de ejemplo. Extraer todos los proyectos.

- measures/component. Devuelve la información del componente indicado. Este componente se indica mediante un *queryParam* denominado *componentKey*. Además, la información deseada se indica con otro *queryParam* denominado *metricKeys*, tras el cual se indican las métricas a extraer sobre ese proyecto separado por comas. Se devuelve un objeto con información sobre el proyecto solicitado y un *JSONArray* de las métricas solicitadas. Ver ejemplo en la Figura 8.

<table>
<tr><td>URL completa</td><td>Ejemplo</td></tr>
<tr><td>http://sonar.fi.upm.es:9000/api/measures/component&componentKey=test&metricKeys=ncloc</td><td><pre>{
   "metric": "ncloc",
   "value": "114",
   "periods": [
   {
     "index": 1,
      "value": "3"
   },
   {
     "index": 2,
     "value": "-5"
    },
    {
     "index": 3,
     "value": "5"
   }
]
}</pre></td></tr>
</table>

Figura 8: Tabla de ejemplo. Extraer ncloc de un proyecto.

- qualitygates/project_status. Devuelve el *quality gate* de un proyecto, el estado (si ha habido error o no) y los umbrales para las métricas definidas en el *quality gate*. Estos umbrales pueden ser de dos tipos: de error o de *warning*. Cada una de las métricas va acompañada por la operación asociada, que puede ser “mayor que”, “menor que” o “igual que”. Mediante un *queryParam projectKey* se identifica el proyecto estudiado. Ver ejemplo en la Figura 9.

<table>
<tr><td>URL completa</td><td>Ejemplo</td></tr>
<tr><td>http://sonar.fi.upm.es:9000/api/qualitygates/project_status?projectKey=test</td><td><pre>"projectStatus": {
    "status": "ERROR",
    "conditions": [
      {
        "status": "ERROR",
        "metricKey": "new_coverage",
        "comparator": "LT",
        "periodIndex": 1,
        "errorThreshold": "85",
        "actualValue":
"82.50562381034781"
      },
      {
        "status": "ERROR",
        "metricKey":
"new_blocker_violations",
        "comparator": "GT",
        "periodIndex": 1,
        "errorThreshold": "0",
        "actualValue": "14"
      }</pre></td></tr>
</table>

Figura 9: Tabla de ejemplo. Extraer QG de un proyecto.

- issues/search?componentKeys=. Permite sacar los *issues* de un proyecto dado. En este caso nos interesan los *code smells*, pero si hubiese otros *issues* que se quieran extraer, se puede hacer mediante la misma URL de consulta. Para filtrar y extraer los code smells que no están resueltos (es decir, los problemáticos) y solo los *code smells*, se añaden los siguientes *queryParams*: *&resolved=false&types=CODE_SMELL*. Destacar que, en este caso, el profesorado había añadido un *plugin* para detectar los *code smells*, por lo que puede que otras versiones de SonarQube no dispongan de estos datos o no de esta forma. El *JSONArray* de *issues* tiene toda la información que necesitamos. Ver ejemplo en la Figura 10.

URL completa	Ejemplo
http://sonar.fi.upm.es:9000/api/issues/search?componentKeys=prog2:P1NowMeal****fi.upm.es&resolved=false&types=CODE_SMELL	[{"severity":"MAJOR","updateDate":"2019-02-26T17:12:30+0100","line":173,"author":"","rule":"squid:CommentedOutCodeLine","project":"prog2:P1NowMealjramirezfi.upm.es","effort":"5min","message":"This block of commented-out lines of code should be removed.","creationDate":"2019-02-26T17:12:30+0100","type":"CODE_SMELL","tags":["misra","unused"],"component":"prog2:P1NowMealjramirezfi.upm.es:src/gestionpedidos/GestionRepartoLocal.java","flows":[],"organization":"default-organization","textRange":{"endLine":173,"endOffset":75,"startOffset":0,"startLine":173},"debt":"5min","key":"AWkqlE9ULJ-v3ZeHr70E","hash":"77e917f10c2aea792e10196478ee4bc7","status":"OPEN"}...]

Figura 10: Tabla de ejemplo. Extraer code smells de un proyecto.

8. *BACKEND*

En este apartado se presenta una descripción detallada de cada uno de los elementos del *backend* de la aplicación, así como la funcionalidad que este implementa. En el primer subapartado se hablará del diseño del *backend* y del razonamiento seguido para crear las clases mostradas en la Figura 11.

Todas las funciones tienen documentación en formato Java dentro del código, para que sea más sencillo utilizarlas en el futuro, esto ayuda ya que en ocasiones son funciones grandes o con muchos detalles.

El *backend* es principalmente el encargado de comunicarse directamente con SonarQube, mediante la API descrita en el apartado anterior, así como de escribir y guardar los ficheros CSV con los datos extraídos. Para realizar todo esto, cuenta con una serie de clases (Figura 11), cada una con una funcionalidad específica, que se comunicarán entre sí para pasarse datos.

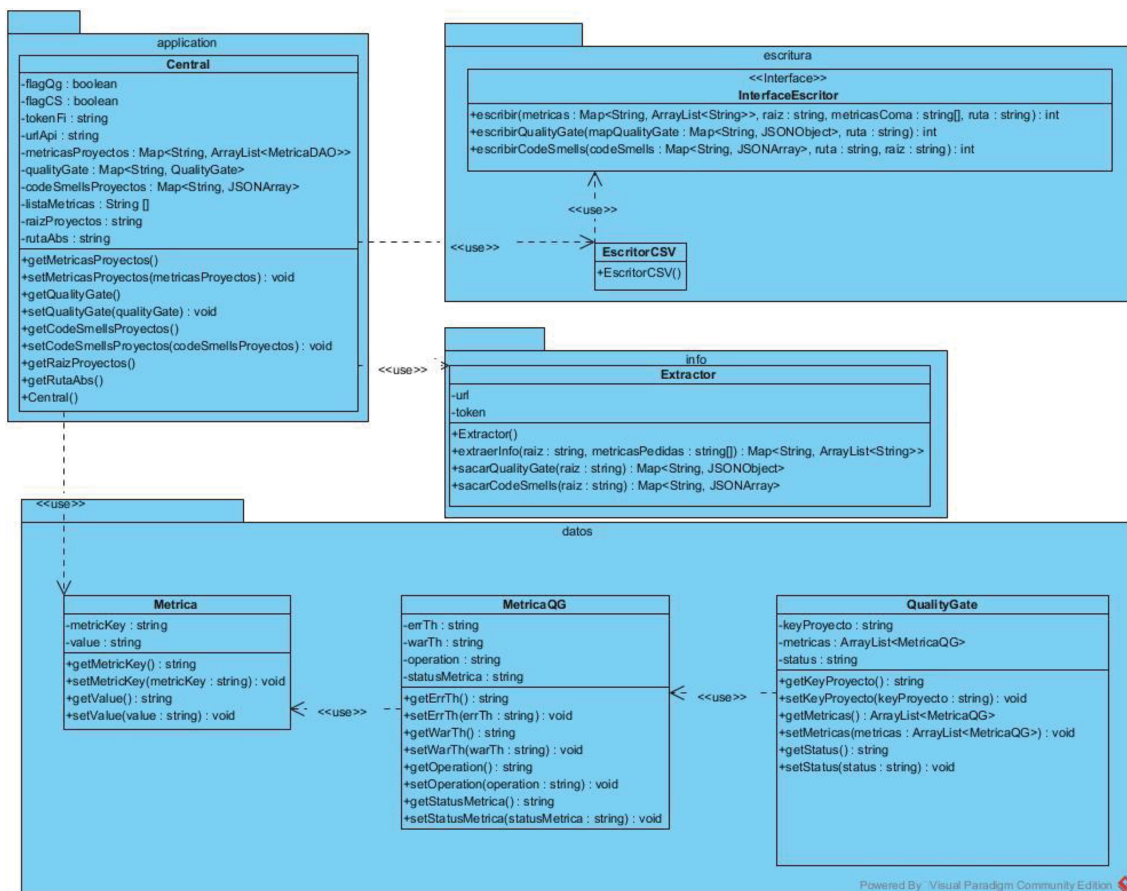


Figura 11: Diagrama de clases del backend (sólo métodos públicos)

8.1. PROTOTIPO INICIAL

Antes de comenzar la codificación de la aplicación, se desarrolló un pequeño prototipo del *backend* para probar el funcionamiento de la API y la información que era accesible.

Dicho prototipo, una versión muy primitiva del futuro *backend*, contaba con un método *main* que ejecutaba una serie de consultas, como si de una batería de pruebas se tratase. Los resultados de estas consultas a la API eran imprimidos por pantalla, lo que permitió realizar un análisis de los datos y de su estructura. Así, se comenzó a estudiar cómo se podían manejar los objetos JSON y cómo se podían realizar las peticiones, conocimientos que fueron aplicados, finalmente, a la versión final.

Podemos dividir el desarrollo del prototipo en tres iteraciones:

1. En la primera iteración, el *endpoint* del SonarQube de la escuela y el *token* utilizado para acceder a los proyectos eran variables estáticas (con esto me refiero a variables con un valor dado y que no dependían de ninguna entrada externa) incluidas en el código fuente. En esta iteración se probó como extraer un listado de los proyectos disponibles para luego imprimirlos por pantalla sin formato alguno.
2. En la segunda iteración, se extrajo la información de algunas métricas y se comenzaron a procesar los datos en formato JSON de una manera más exhaustiva y utilizando una librería (*org.json* [4]). Esto permitió una impresión en la consola más ordenada y de las secciones de los objetos que resultaban de utilidad.
3. Por último, en la tercera iteración, se pidieron los datos de la URL y el *token* por teclado, así como las métricas que se querían extraer, ya que todavía no estaban seleccionadas todas.

De estas pruebas iniciales, se obtuvieron las siguientes conclusiones:

- Era necesario dedicar una clase exclusivamente a realizar las peticiones a la API y procesar el resultado, debido a la complejidad de los datos. De este modo, esta clase debería de ser capaz de solicitar todas las métricas seleccionadas, independientemente de si estas varían, es decir, independientemente de si se solicitasen métricas diferentes. Esto permite aislar errores de contacto con la API de SonarQube.
- También, sería necesario definir modelos de datos que pudiesen almacenar la información extraída. Uno para las métricas y otro para el *quality gate*. Con esto, se evitaría tener que leer o procesar los ficheros a generar cuando se quisiese visualizar la información en una interfaz.

- Dado que uno de los requisitos de la aplicación era que esta pudiese ampliarse para escribir en varios formatos (o en una base de datos), sería necesaria una interfaz que permitiese implementar diferentes tipos de Escritores. En el caso de esta TFG se ha implementado (previo acuerdo con el tutor) el formato CSV, pero podría darse la necesidad en el futuro de generar ficheros XML o de inserción de la información en una base de datos, como ya se ha mencionado.
- Por tanto, habría que implementar una clase central que coordinase a las anteriores y realizase el intercambio de información entre ellas, como si de un *broker* se tratase.

A partir de estos cuatro puntos iniciales, se comenzó a desarrollar el *backend* de la aplicación al completo.

Antes de entrar en detalle, mencionar que para leer objetos y colecciones de objetos JSON, se utiliza la librería *org.json*. Para escribir ficheros en formato CSV se utiliza *opencsv* [6], que también permite un uso gratuito y libre.

8.2. CENTRAL

Como podemos observar en la Figura 11, la clase principal es Central, dentro del paquete *application*. Esta es la encargada de crear una instancia del EscritorCSV y otra del Extractor, así como de almacenar en memoria los datos necesarios para poder utilizarlos más adelante en la visualización. Todo esto lo realiza ejecutando su *main* y ayudándose de funciones auxiliares. Recibe su nombre por ser el pilar central de la aplicación, o al menos del *backend*, y por ser como una centralita desde donde se distribuye la información y que controla el flujo de ejecución.

La clase tiene algunos atributos útiles, como mapas donde se almacenan las métricas, los *code smells* o los *quality gates*. Por otro lado, tenemos la URL de la API de SonarQube, el *token* a utilizar (ya que son únicos y están ligados a los permisos de la cuenta), la raíz de los proyectos a estudiar y la ruta donde guardar los ficheros. Estos últimos atributos se obtendrán de la pantalla principal de la interfaz y serán enviados, cuando sea necesario, a las clases auxiliares (Extractor y EscritorCSV) que veremos más adelante.

Sin embargo, las métricas no se le solicitarán al usuario, ya que sólo se extraerán las seleccionadas para la realización de este trabajo, pudiendo añadirse más si se edita la estructura definida que las contiene, dentro de esta misma clase.

Cabe destacar dos atributos que sirven para indicar si el usuario ha solicitado la métrica *quality gates* o *code smells*, son los *flags* Qg y Cs. Esto se añadió pensando en el futuro, ya que, si se modificasen las métricas y no se quisiese obtener una de las

correspondientes a estas variables de control, no tendría sentido que se generasen, por ejemplo, ficheros vacíos para ambas. Lo que no implica que si se solicitan y no hay datos para una de ellas, no se vaya a generar el fichero. Gracias a esto, la flexibilidad de la aplicación aumenta.

Una vez se les ha dado valor a los atributos, se proceden a realizar llamadas al Extractor y al EscritorCSV, mientras a la par se van utilizando funciones auxiliares que guardarán los datos obtenidos en memoria.

8.3. EXTRACTOR

Dentro del paquete info, tenemos la clase Extractor. Esta es la encargada de, a partir del método público extraerInfo, realizar una serie de peticiones a la API de SonarQube.

Estas peticiones pueden agruparse en dos tipos, principalmente. El primero (tipo 1 en la Figura 12) consiste en pedirle a SonarQube la lista de proyectos y, tras obtenerlos, guardar en la aplicación (en memoria) únicamente las *keys* (o identificadores) de aquellos correspondientes a la raíz que queremos estudiar.

El segundo tipo (tipo 2 en la Figura 12) es, simplemente, la solicitud del valor de cada una de las métricas seleccionadas. Cabe destacar en este punto que todas las métricas se pedían a la vez, es decir, todas van insertadas en la URL de la petición. Con esto se evita que se tenga que realizar una petición por métrica (serían demasiadas peticiones). En cambio, se realiza una petición por proyecto, obteniendo todas las métricas pedidas asociadas a dicho proyecto en conjunto. Para realizar cada petición, independientemente del tipo que sea, se utiliza un método auxiliar, privado, llamado *request* que devuelve la respuesta y se encarga de leerla, así se evitó la duplicidad de código.

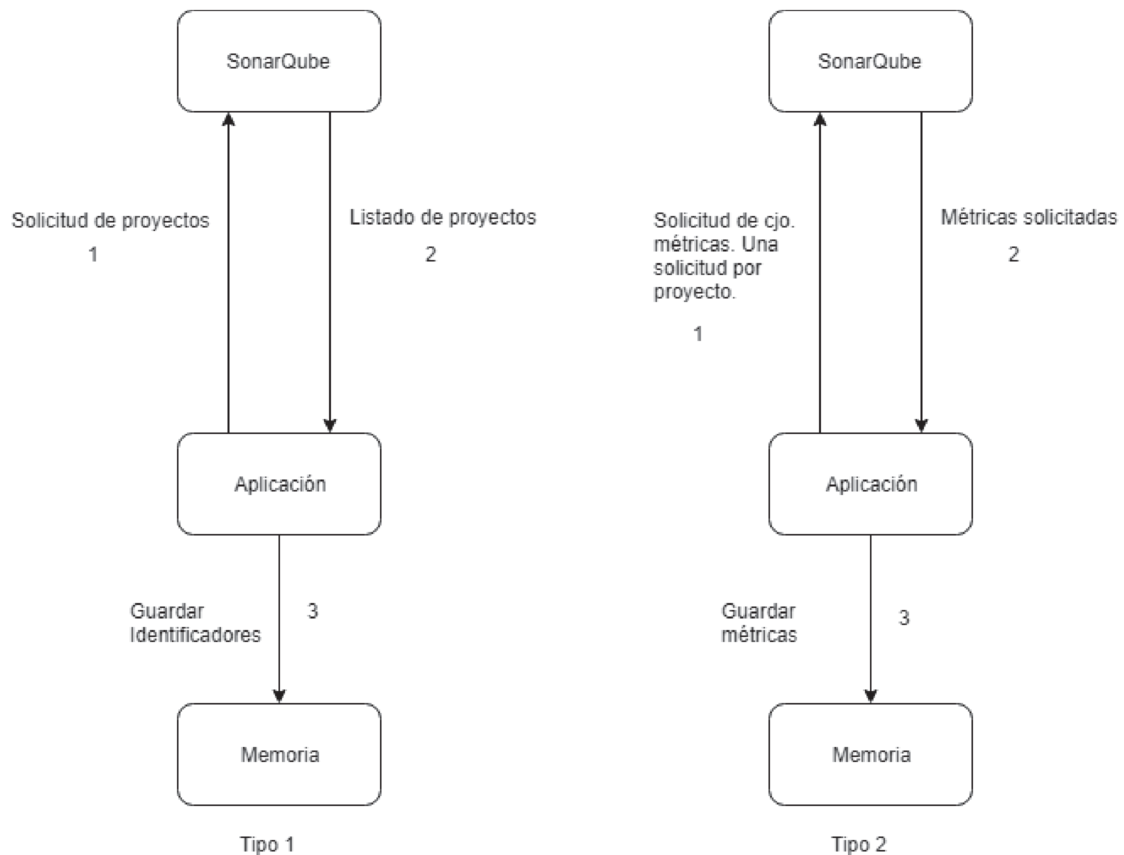


Figura 12: Tipos de peticiones.

Uno de los principales problemas afrontados al realizar la codificación de esta clase, fue que las métricas obtenidas no venían siempre en el mismo orden, a pesar de que se solicitasen las mismas y en el mismo orden. Por esto, fue necesario ordenarlas previamente a realizar las peticiones.

El método auxiliar para ordenar simplemente implementa un *comparator* que se le pasa como argumento al método *sort* de la clase *Collections* de Java. Además, este método auxiliar también recibe la clave del campo por el que queremos ordenar. En este caso siempre se le pasa el nombre de la métrica, de modo que salen ordenadas alfabéticamente, pero en cualquier momento se podrían ordenar por otros campos, aumentando así la flexibilidad.

El Extractor también cuenta con métodos exclusivos para obtener los *code smells* y los *quality gates*, por lo mencionado en el apartado anterior de la flexibilidad, ya que son métricas más complejas que el resto, como veremos en el siguiente apartado.

8.4. ESCRITORCSV

Dentro del paquete escritura, podemos encontrar una interfaz y su implementación. La interfaz se añadió para poder darles flexibilidad a los escritores, de este modo, se pueden implementar clases que, por ejemplo, escriban ficheros en un formato distinto al CSV o en una base de datos.

El EscritorCSV implementa esta interfaz basándose en un CSV delimitado por ';', antes de seleccionar el delimitador hubo que observar que caracteres utilizaban las métricas.

Los métodos principales permiten generar tres ficheros:

- Un fichero con las métricas del proyecto, denominado "Raíz.csv", siendo la raíz la que el usuario ha decidido introducir en la aplicación para obtener la información de los proyectos asociados a ella. En la Figura 13 se puede observar un ejemplo del fichero mencionado, en el que habría una línea por proyecto.

raíz	key_proyecto	fich_QG	alert_status	…
ClubLector	prog2:ClubLector****.com	C:\...	ERROR	…

Figura 13: Fragmento de ejemplo de un fichero de métricas

- Un fichero con los *code smells,* denominado "Raíz_*CODE_SMELLS*.csv", que contiene exclusivamente esta métrica. Se decidió hacer un fichero exclusivo debido a que cada uno de los *code smell* se desglosa en bastante información de utilidad, de modo que un fichero de métricas y, además, que contuviese toda esta información, sería confuso y poco manejable. Dentro de un *code smell* podemos encontrar el fichero de la práctica del alumno afectado, el mensaje de advertencia, la línea o el tiempo estimado para resolver el problema. No es necesario que venga referenciado en el archivo de métricas, ya que solo hay uno para la raíz dada, igual que el mencionado archivo. Mencionar que los *code smells* no pueden extraerse directamente, por lo que es necesario obtener primero los *issues* y luego filtrar estos por la categoría *code smells* y que estén abiertos (no solucionados). En la Figura 14 se puede observar un ejemplo del fichero mencionado, en el que habría una línea por *code smell*.

raíz	key_proyecto	Mensaje	Línea	…
ClubLector	prog2:ClubLector****.com	*Rename the…*	65	…

Figura 14: Fragmento de ejemplo de un fichero de code smells

- Un último fichero en formato JSON que contiene los datos a bajo nivel del *quality gate* de cada proyecto. Se denomina "*Key* del proyecto_QG.json". Se decidió que los *quality gates* estuviesen en un fichero a parte debido a su complejidad, ya que además, no todos los proyectos se tienen por qué regir por el mismo. Por si fuese poco, un *quality gate* contiene mucha información, para cada una de las métricas valoradas se da el valor de advertencia, de error o el estado de la métrica. Viene referenciado en el archivo de métricas. En la Figura 15 se puede observar un ejemplo del fichero mencionado.


```
"conditions":[{"comparator":"GT","metricKey":"new_vulnerabilities","errorThr
{"comparator":"GT","metricKey":"new_bugs","errorThreshold":"0","actualValue"
{"comparator":"GT","metricKey":"new_sqale_debt_ratio","errorThreshold":"5","
{"comparator":"GT","metricKey":"cognitive_complexity","errorThreshold":"45",
{"comparator":"GT","metricKey":"test_failures","errorThreshold":"5","actualV
{"comparator":"GT","metricKey":"test_errors","errorThreshold":"4","actualVal
"status":"WARN"}
```


Figura 15: Fragmento de ejemplo de un fichero de quality gate

8.5. MODELOS DE DATOS

Dentro del paquete de datos, se encuentran los modelos que han servido para almacenar los datos extraídos de SonarQube en memoria. Con esto se evita que sea necesario leer o iterar sobre los ficheros generados para poder mostrar esta información en pantalla al usuario de la aplicación.

Si bien hay objetos que modelan las métricas de manera genérica, pudiéndose adaptar a cualquiera de ellas (Metrica en la Figura 11), fue necesario realizar una extensión de la clase para poder modelar la información de los *quality gate* de cada proyecto (MetricaQG en la Figura 11).

Estos objetos específicos contienen información como los umbrales de error y de advertencia. Además, cualquier objeto de tipo *QualityGate* tendrá una lista conteniendo todas las métricas en forma de MetricaQG, además del estado global del proyecto, teniendo así, por fin, toda la información de cada *quality gate* guardada de manera ordenada.

Por otro lado, cabe mencionar que no fue necesario realizar un modelo específico para los *code smells*, ya que su estructura no era tan compleja como la de los *quality gate*, por esto simplemente de utiliza el JSONArray extraído de SonarQube, que es de fácil manipulación (cada elemento es un objeto JSON con el mensaje, la línea, etc.), al contrario que el objeto JSON del *quality gate*, que era muy grande.

9. *FRONTEND*

En el presente apartado se explicará el funcionamiento de la interfaz y se mostrará su diagrama de clases correspondiente. También se mostrará el diseño de la interfaz y su aspecto final.

El objetivo de la interfaz es permitir la visualización de los datos obtenidos de un modo claro y simple. Por ello, no cuenta con muchas pantallas diferentes, pero, sin embargo, las que tiene cubren las necesidades de la aplicación. Por otro lado, mencionar que el objetivo principal de la interfaz ha sido facilitar el paso de parámetros para la extracción, de modo que una posible evaluación de estos datos sea posible.

Para realizar la interfaz se utilizó JavaFX, el cual viene incluido por defecto en la distribución de Oracle de Java. Sin embargo, si lo que se tiene instalado es la *open* JDK, por ejemplo, en un sistema Linux, puede ser necesario instalarlo a parte. Además, el tutor compartió alguna información proveniente de un master que imparte, lo cual fue útil a la hora de desarrollar, ya que, inicialmente, el alumno no conocía esta tecnología.

En las Figuras 16 y 17 pueden observarse las clases que componen la interfaz, y que pasaremos a explicar en detalle en los siguientes subapartados. Por otro lado, para implementarla, se ha intentado mantener la independencia y la división entre la lógica de negocio de la aplicación y el aspecto de la interfaz.

Respondiendo al motivo anteriormente mencionado (modularidad e independencia), surgen las clases que se pueden ver en las Figuras 16 y 17. Existen unos modelos de datos que permiten almacenar el contenido de las tablas a visualizar en memoria y que son independientes de las clases que los manejan (controladores). Por otro lado, los archivos que contienen el diseño de la interfaz se mantienen también independientes de la lógica que se encarga de manejar las variables contenidas en ellos.

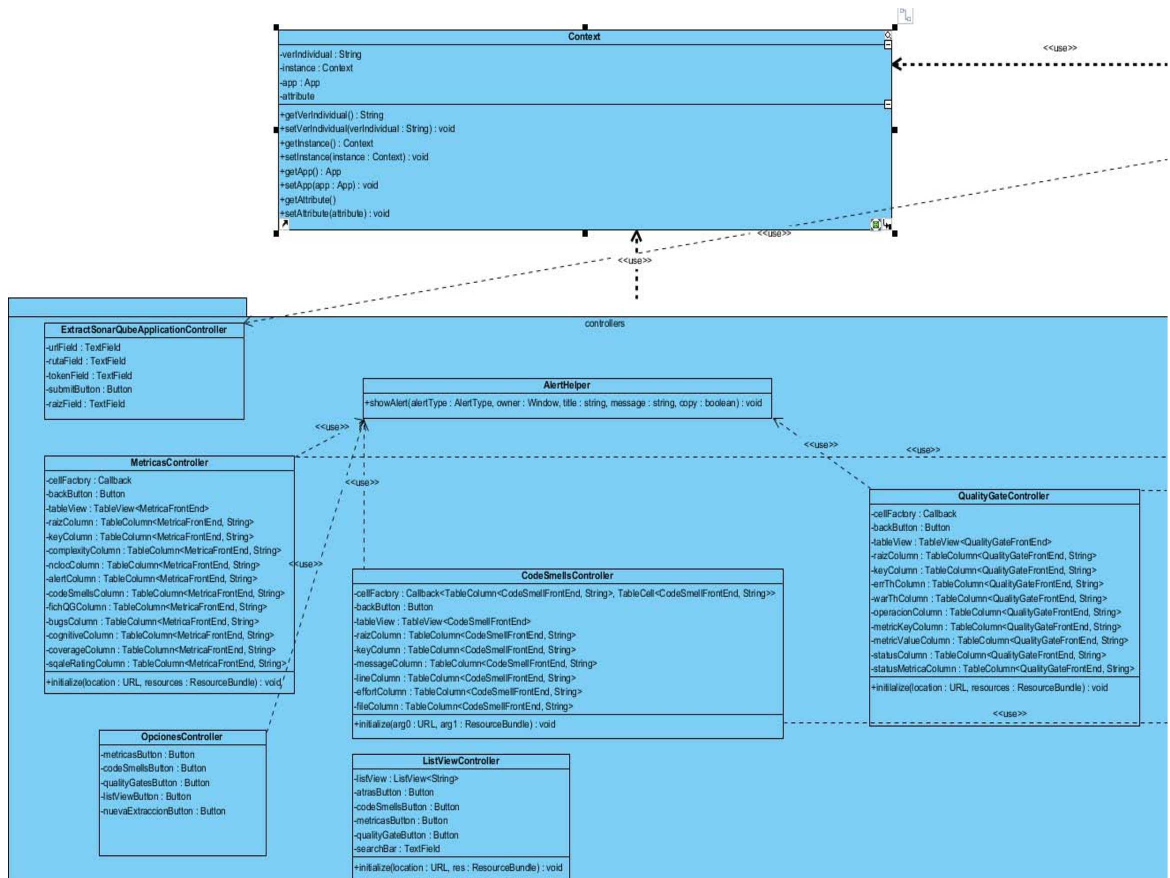


Figura 16: Diagrama de clases del frontend (sólo métodos públicos). Controladores y Contexto.

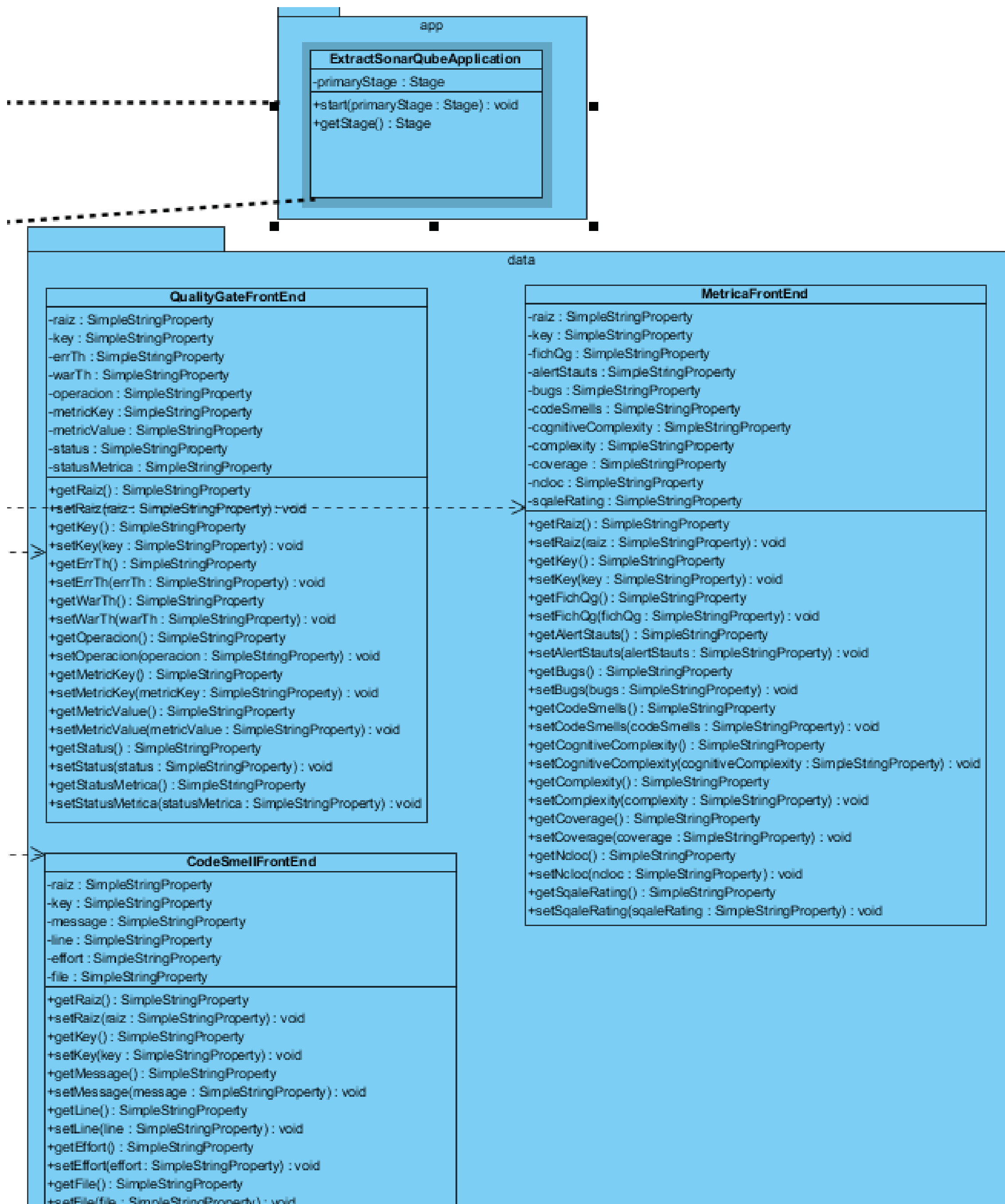


Diagrama de clases del frontend (sólo métodos públicos). Datos y app. Figura 17:

9.1. DISEÑO DE LA INTERFAZ

Los archivos correspondientes al diseño no aparecen en el diagrama de clases. Tienen su propio paquete, denominado fxml. En ellos está contenido el esquema o plantilla de cada una de las pantallas, los elementos que la componen, dónde están situados, que tamaño tienen, etc. Diagrama de clases del *frontend* (sólo métodos públicos). Datos y app.

Además, en este paquete hay dos CSS, uno para las tablas [24] y otro para la lista [25]. Estos CSS se han obtenido de fuentes externas y mejoran el aspecto de la aplicación, por ejemplo, resaltando las celdas al seleccionarlas.

Por otro lado, a la hora de escribir los ficheros fxml, se utilizó la herramienta llamada *Scene Builder*, que permite diseñar las pantallas de una manera más visual. Es de fácil acceso e instalación y solo es necesario indicarle a Eclipse la ruta del ejecutable, de modo que se puedan editar estos archivos con la herramienta directamente. A pesar de esto, muchas cosas fueron modificadas directamente dentro del propio fxml por comodidad y rapidez.

Uno de los elementos más comunes en estos archivos ha sido el *GridPane*, uno de los tipos de diseño que ofrece JavaFX, que permitía mostrar elementos ordenados por fila y columna, como si de una matriz se tratase.

A la hora de realizar el diseño se tuvo en cuenta la simpleza y la claridad y, desde el principio, se tuvo claro que el mejor modo de mostrar la información era en tablas. Estas tablas permiten realizar doble clic sobre una celda y copiar su contenido al portapapeles. Además, JavaFX permite redimensionar las columnas para adaptarlas al contenido de las celdas. A continuación se muestran algunos bocetos de las pantallas que componen la aplicación.

EXTRACCIÓN DE DATOS

URL

TOKEN

RUTA FICHEROS

RAÍZ PROYECTO

BOTÓN

Figura 18: Diseño de la pantalla principal

En la Figura 18 se puede observar la pantalla principal de la aplicación, donde el usuario introducirá los datos necesarios para poder realizar la extracción. Es una pantalla simple, con cuatro campos de texto que recogerán la entrada por teclado del usuario, un título y el botón de acción.

Key proyecto	Métrica 1	Métrica 2
Value 1	Value 2	Value 3
Value 4	Value 5	Value 6
Value 7	Value 8	Value 9
Value 10	Value 11	Value 12

VOLVER

Figura 19: Diseño de las pantallas de visualización

En la Figura 19, se observa el diseño de las pantallas que permiten visualizar los datos. Como se ha mencionado, la visualización se produce en forma de tabla, por lo que es necesario una *TableView* de JavaFX para implementarlas.

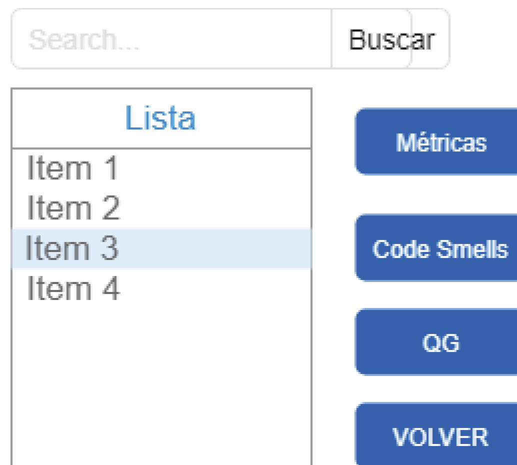


Figura 20: Diseño de la pantalla del listado de proyectos

Por último, en la Figura 20 se puede observar la lista que contendrá los identificadores de los proyectos y los botones situados en el lado derecho de la pantalla con las opciones. Esta lista requiere de una *ListView* de JavaFX para ser implementada y que se pueda buscar sobre ella. El buscador no es más que un campo de texto.

Para cerrar este apartado, se mostrarán unas capturas de la aplicación final de modo que sea posible ver el paso del diseño a la aplicación real.

Figura 21: Pantalla principal de la aplicación

La Figura 21 muestra la pantalla principal que se corresponde con la Figura 18. Como vemos se ha añadido la función de seleccionar un directorio mediante el explorador de archivos del sistema.

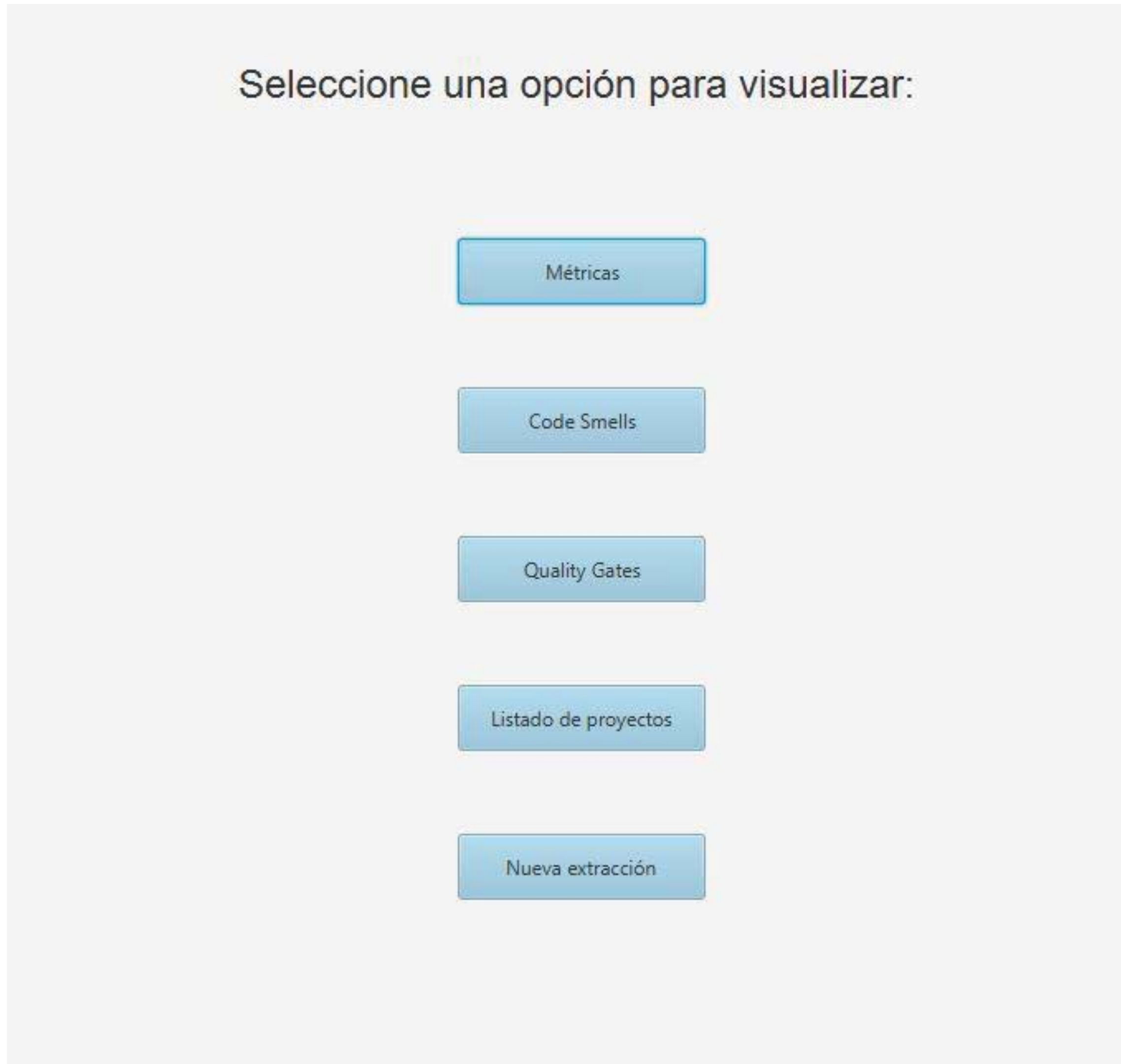
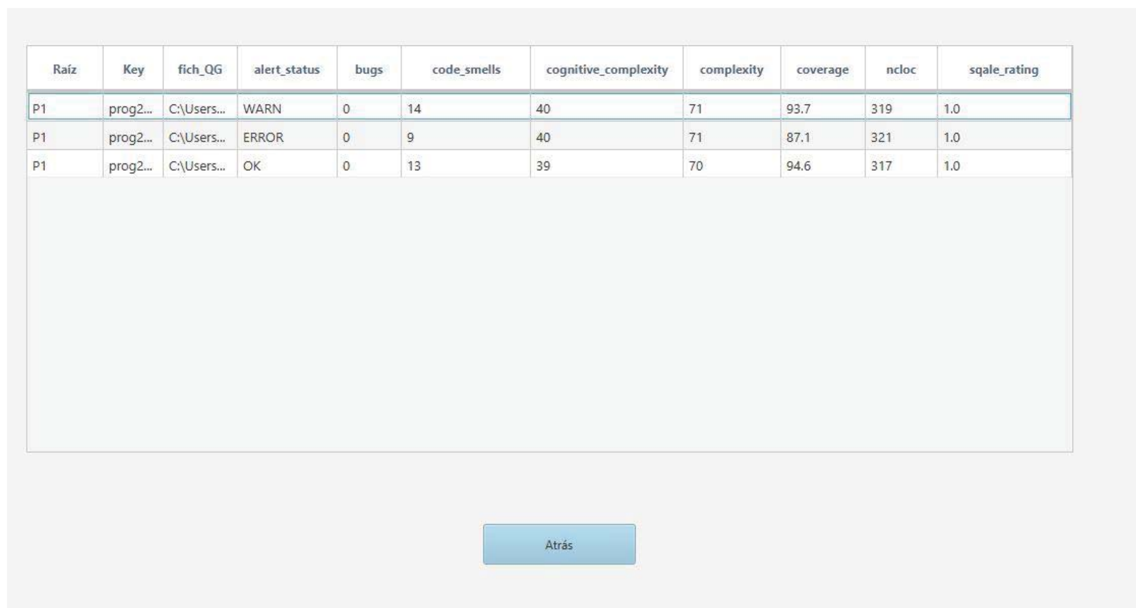


Figura 22: Pantalla de opciones

La Figura 22 muestra las opciones que tiene disponible el usuario, es la única que no tiene un diseño asociado, ya que es una pantalla realmente sencilla.



Raíz	Key	fich_QG	alert_status	bugs	code_smells	cognitive_complexity	complexity	coverage	ncloc	sqale_rating
P1	prog2...	C:\Users...	WARN	0	14	40	71	93.7	319	1.0
P1	prog2...	C:\Users...	ERROR	0	9	40	71	87.1	321	1.0
P1	prog2...	C:\Users...	OK	0	13	39	70	94.6	317	1.0

Figura 23: Pantalla de visualización de métricas

La Figura 23 muestra la pantalla que contiene la tabla de métricas, en el caso de los *quality gates* y de los *code smells* la pantalla es idéntica. Se corresponde con la Figura 19.

Figura 24: Pantalla del listado de proyectos

La Figura 24 muestra el listado de los proyectos asociados a la raíz dada. Se corresponde con el diseño de la Figura 20. El mayor cambio es que se ha añadido el icono de la lupa al lado del buscador.

9.2. APLICACIÓN

La clase principal de la interfaz, y la que se ejecuta al lanzar el programa, es *ExtractSonarQubeApplication*. Esta clase inicializa la vista principal de la aplicación, carga la pantalla de inicio y pone el título de la ventana.

9.3. CONTROLADORES

En el paquete de controladores, encontramos la lógica de la aplicación. En total existen seis controladores y una clase auxiliar que pasaremos a describir a continuación. Antes de continuar, mencionar que cuando se habla de atributos en estas clases, nos estamos refiriendo a atributos que utilizan la inyección fxml de JavaFX (a no ser que se especifique lo contrario), es decir, que tienen su correspondencia en un fichero fxml que contiene la parte visual de la pantalla. Por otro lado, se puede afirmar que se utiliza la programación orientada a eventos [26], es decir, que los elementos y el sistema responden ante las interacciones del usuario (eventos).

El controlador *ExtractSonarQubeApplicationController* representa el funcionamiento de la pantalla principal. Contiene los atributos que se corresponden con los campos de entrada que el usuario debe rellenar antes de iniciar la extracción de datos de SonarQube, así como con el botón de inicio. Estos campos de entrada son: la URL de la API de SonarQube, el *token* activo para acceder a los datos, la ruta donde se van a generar los ficheros y la raíz de los proyectos de los que se va a sacar información. Si alguno de los campos no está relleno, se lanzará un mensaje de error con ayuda de la clase *AlertHelper*, indicándole al usuario que problema ha habido.

Por otro lado, se permite que el usuario elija un directorio de su gusto, si no elige ninguno se utilizará una ruta a una carpeta relativa al proyecto. Gracias a la utilidad de Java *File.separator* es sencillo determinar con qué tipo de '/' debe de construirse la ruta al directorio seleccionado.

Por último, se encarga de realizar la llamada a la clase Central del *backend* y de cargar la pantalla de opciones.

En esta nueva pantalla, se ejecutará el código contenido en *OpcionesController*. En dicho controlador, se encuentran los atributos correspondientes a los siguientes botones: visualizar la lista de métricas de todos los proyectos, visualizar los *code smells* de todos los proyectos, visualizar los *quality gates* de todos los proyectos, ver un listado de las *keys* de todos los proyectos y realizar una nueva extracción. Dependiendo del botón pulsado, se cargará un archivo fxml u otro, mediante la función privada *continueToStage*.

Los controladores *MetricasController, CodeSmellsController* y *QualityGateController* cuentan con los atributos correspondientes a sus tablas, en forma de *TableView*, a las columnas de esta tabla, y al botón para ir a la pantalla de opciones de nuevo. Para rellenar la tabla se obtienen los datos de la aplicación del backend, que se encargó de almacenarlos en memoria en su momento. Una vez los tenemos, los convertimos a los modelos de datos del *frontend*, de modo que puedan

"encajar" en una *TableView*, para lo cual se crea para cada columna un *PropertyValueFactory* que buscará en el objeto correspondiente el valor del atributo (no fxml) que se corresponda con el parámetro de entrada.

Otro atributo a destacar es la *cellFactory*, que implementa un filtro que detectará cuando se ha hecho doble clic sobre la celda, apareciendo así una ventana emergente, con ayuda de la clase *AlertHelper*, que permitirá copiar el contenido de la celda. Esta funcionalidad surge debido a que algunos valores (como las rutas de los ficheros) eran muy extensos, así que para visualizar el valor entero sin tener que redimensionar la columna lo más sencillo era permitir el doble clic del usuario sobre la celda.

ListViewController implementa la funcionalidad de la lista de proyectos. El usuario podrá ver las *keys* de cada uno y además podrá filtrarlos con un buscador. Una vez se haya seleccionado uno, se podrá pulsar sobre cualquiera de las opciones para visualizar sus datos. En esta pantalla aparece una pequeña imagen de una lupa, al lado del buscador, es de libre distribución y obtenida de la página de *google design* [27].

La clase *AlertHelper* no es un controlador propiamente dicho, si no una clase auxiliar que sirve para crear las ventanas emergentes. Permite crear dos tipos de ventanas: las de error para la pantalla principal y las de doble clic en una celda, que son las únicas que permiten copiar el contenido. Esta funcionalidad se consigue colocando un campo de texto, que contiene ya el contenido de la celda seleccionada, dentro de la caja de alerta.

9.4. MODELOS DE DATOS

Se han diseñado nuevos tipos de datos para el *frontend*, manteniendo la independencia del *backend* y pensados exclusivamente para construir las tablas. Esto implica tener que modificar estos datos y los fxml de las tablas si se añaden nuevas métricas. Por otro lado, en el backend no hará falta más que añadir la métrica nueva al *array* de métricas. Con esta decisión de diseño, mantenemos separados backend de *frontend* en la medida de lo posible, y se permite una mayor flexibilidad.

Los modelos creados para la interfaz son parecidos a los anteriores, pero utilizan como atributos el tipo *SingleStringProperty*, esto permite que las columnas de las tablas tengan su correspondiente atributo en el objeto que se corresponde con el tipo de los datos de la columna. JavaFX, por tanto, permite el enlace de datos fácilmente con Propiedades (*Properties*), y sería más complicado actualizar los campos de una tabla o rellenarla si no se utilizase este tipo de dato.

9.5. CONTEXTO

Esta clase tiene su propio paquete, ya que no encajaba en el resto. Es una clase auxiliar que almacena la instancia de la aplicación del *backend* activa, de modo que no se pierdan los datos que tiene en memoria y que estos puedan pasarse entre distintas pantallas. También, al seleccionar un proyecto en el listado, se guarda el proyecto seleccionado en un atributo del Contexto, de modo que, si el atributo tiene un valor asignado, se visualizarán sólo los datos de ese proyecto, y si no, se visualizarán los de todos los proyectos. En resumen, es una clase que ayuda a controlar algunos comportamientos de la aplicación.

10. LÍNEAS FUTURAS

En este apartado se comentarán características que podrían implementarse en el futuro para mejorar o expandir la aplicación actual. Algunas de ellas han sido propuestas por el tutor y otras por el alumno.

En primer lugar, y solicitado por el tutor, en el futuro los ficheros generados deberían de poder procesarse y ser insertados en una base de datos (con su correspondiente diseño). Implementar más formatos de ficheros también podría ser interesante, así como generar las clases que creen la base de datos de acuerdo con las métricas que se han seleccionado.

También se podrían añadir algunas funcionalidades a la interfaz que podrían ser de utilidad para el usuario. Por ejemplo, sugerencias a la hora de escribir en un campo de texto, de modo que la aplicación recuerde lo que se ha escrito en dicho campo anteriormente. Relacionado con los campos de texto, sería interesante que la aplicación recordase los últimos datos introducidos en la pantalla principal.

Por otro lado, y relacionado con lo anterior, podría ser útil implementar un historial de extracciones, de modo que se puedan recuperar los datos de la extracción anterior, de este modo, sería más rápido el realizar múltiples extracciones y más fluida la navegación entre ellas, al no tener que volver a la pantalla principal.

Por último, que a la hora de buscar los proyectos por raíz se pudiese introducir solo un fragmento de esta, podría ser también útil. Aunque podría haber problemas si hubiese varias raíces con el mismo nombre y sólo se quisiese obtener la información sobre una de ellas.

11. OBJETIVOS ALCANZADOS

Al principio del curso, fueron establecidos una serie de objetivos a cumplir a lo largo del desarrollo del proyecto. En este apartado, se pretende hacer una recapitulación de ellos para indicar cuáles se han cumplido.

La lista de objetivos utilizada se corresponde tanto con la presentada en la Memoria intermedia como con la del Plan de trabajo, vienen enumerados en orden de realización. En conclusión, no ha sido necesario modificar ninguno de los objetivos.

Para ello, se ha realizado una tabla (Figura 25) que cuenta con dos columnas:

- Tarea. Dónde se indica el objetivo en sí mismo.
- Estado. Indica si el objetivo se ha completado o no.

Tarea	Estado
Estudiar SonarQube	COMPLETADO
Documentarse sobre la Web API de SonarQube	COMPLETADO
Selección de la información a extraer	COMPLETADO
Selección y documentación de los servicios web a usar.	COMPLETADO
Pruebas de concepto y toma de contacto con la Web API	COMPLETADO
Diseño del sistema de recuperación de datos	COMPLETADO
Diseño del API para el acceso y gestión de los datos recuperados	COMPLETADO
Diseño de la GUI de la aplicación	COMPLETADO
Implementación del sistema de recuperación de datos	COMPLETADO
Implementación del API para el acceso y gestión de los datos recuperados	COMPLETADO
Implementación de la aplicación	COMPLETADO
Preparar la memoria	COMPLETADO

Figura 25: Tabla de objetivos del TFG

12. CONCLUSIONES

Tras realizar este proyecto, se determina que la utilidad de SonarQube, tanto en un ámbito profesional como académico, es clara. En este caso, los alumnos podrán ver reducido el tiempo de espera entre la entrega de una práctica y su corrección, que usualmente es bastante largo.

Además, gracias a las métricas que proporciona SonarQube, los profesores podrán hacer una evaluación más detallada del trabajo de los alumnos o incluso sacar estadísticas sobre datos concretos de manera muy sencilla.

La posibilidad de expandir la aplicación en el futuro, con otros TFG, también es amplia, por lo que lo anteriormente mencionado podría acentuarse más, si cabe. De cara a esta expansión o mejora, cabe mencionar que se puede modificar el *frontend* de acuerdo con las necesidades del momento mientras se utiliza el mismo backend.

REFERENCIAS

[1] SonarSource SA, «SonarQube,» SonarSource SA, 2007. [En línea]. Available: https://www.sonarqube.org/. [Último acceso: 10 2 2019].

[2] SonarSource SA, «SonarQube Web API,» [En línea]. Available: https://codeen-app.euclid-ec.org/sonar/web_api/. [Último acceso: 16 2 2019].

[3] Oracle, 3 3 2015. [En línea]. Available: https://docs.oracle.com/javase/8/docs/.

[4] S. Leary, «Github,» 13 8 2018. [En línea]. Available: https://github.com/stleary/JSON-java.

[5] D. Crockford, «JSON,» 1 12 2005. [En línea]. Available: https://www.json.org/.

[6] S. Glen, C. Scott y R. J. Andrew, «opencsv.sourceforge,» [En línea]. Available: http://opencsv.sourceforge.net/.

[7] Oracle, «openjfx,» 13 5 2012. [En línea]. Available: https://openjfx.io/.

[8] M. Nadeem, «Dzone,» 30 7 2015. [En línea]. Available: https://dzone.com/articles/why-sonarqube-1.

[9] Apache, «Squale,» [En línea]. Available: http://www.squale.org/.

[10] MILINAUDARA, «milinaudara.wordpress,» 15 1 2015. [En línea]. Available: https://milinaudara.wordpress.com/2015/01/15/an-introduction-to-sonarqube/.

[11] K. Semizhon, 23 1 2017. [En línea]. Available: https://www.slideshare.net/KateSoglaeff/how-to-improve-code-quality-for-ios-apps.

[12] S. Kosuri, «Research Gate,» 5 2006. [En línea]. Available: https://www.researchgate.net/publication/37994098_GeneJax_A_Prototype_CAD_tool_in_support_of_Genome_Refactoring.

[13] W3C, «W3 XML,» 10 11 2016. [En línea]. Available: https://www.w3.org/XML/.

[14] Mozilla, «JavaScript,» [En línea]. Available: https://developer.mozilla.org/es/docs/Web/JavaScript.

[15] SonarSource S.A, «SonarQube documentation,» [En línea]. Available: https://docs.sonarqube.org/latest/architecture/architecture-integration/.

[16] R. T. Fielding, «Architectural Styles and the Design of Network-based Software Architectures,» 2000. [En línea]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[17] Wikipedia (Apache), «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Apache_Ant.

[18] Microsoft, «OneDrive,» 18 2 2014. [En línea]. Available: https://onedrive.live.com/about/es-es/.

[19] Eclipse Foundation, [En línea]. Available: https://www.eclipse.org/.

[20] Eclipse Foundation, «e(fx)clipse,» 28 2 2019. [En línea]. Available: https://www.eclipse.org/efxclipse/index.html.

[21] Visual Paradigm, «Visual Paradigm,» 20 6 2002. [En línea]. Available: https://www.visual-paradigm.com/.

[22] JGraph Ltd, [En línea]. Available: https://about.draw.io/.

[23] Gluon, «gluonhq,» 5 6 2018. [En línea]. Available: https://gluonhq.com/products/scene-builder/.

[24] M. A. Carmelo, «Github,» 12 11 2015. [En línea]. Available: https://github.com/TutorProgramacion/TableView-JavaFX.

[25] R. Schwarze, «Stack overflow,» 27 3 2014. [En línea]. Available: https://stackoverflow.com/questions/15641478/javafx-css-styling-listview.

[26] A. Molas, «Monografias,» [En línea]. Available: https://www.monografias.com/trabajos/progeventos/progeventos.shtml.

[27] Google, «Google Design,» [En línea]. Available: https://design.google/. [Último acceso: 4 2019].

ANEXO: MANUAL DE LA APLICACIÓN

Para ejecutar la aplicación tan sólo hay que descargarse el archivo *ExtractSonar.jar*, que ha sido generado en Eclipse mediante la opción de exportar un JAR ejecutable.

Este JAR ya contiene las librerías que se han mencionado en el texto por lo que no es necesario hacer nada más.

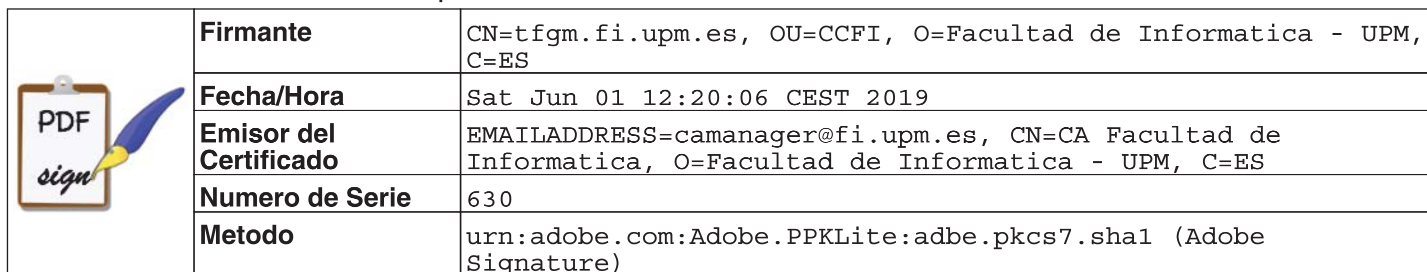
Por otro lado, el usuario debería de tener instalado en su máquina Java 8 como mínimo. Como se ha comentado en el texto, si estamos en un sistema Windows no será necesario instalar nada más, pero si estamos en un sistema Unix, por ejemplo Ubuntu 18, es posible que haya que instalar JavaFX a parte. Para esto, se instala un paquete denominado openjfx con el siguiente comando `sudo apt-get install openjfx`.

Dicho commando nos proporciona los siguientes paquetes:

```
/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/ext/jfxrt.jar
/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/jfxswt.jar
/usr/lib/jvm/java-8-openjdk-amd64/lib/ant-javafx.jar
/usr/lib/jvm/java-8-openjdk-amd64/lib/javafx-mx.jar
```

Una vez tenidas en cuenta estas consideraciones, se podrá ejecutar la aplicación correctamente. La navegación por las pantallas es simple, ya que el menu principal no tiene muchos botones. Si no se selecciona un directorio, se generará uno nuevo al lado del JAR del proyecto.

Este documento esta firmado por



Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Sat Jun 01 12:20:06 CEST 2019
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)