Bilbao / Bizkaia

Estructura *plugin* Sonar

Iñaki García Noya

22/09/2020

Índice

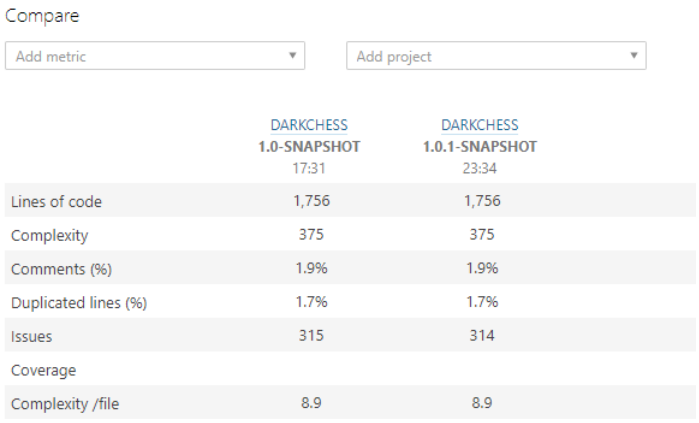
1. Introducción

En este articulo se redacta el diseño inicial y final del *plugin* a realizar en la plataforma SonarQube. Para ello, se ha realizado un estudio de las diferentes documentaciones que están disponibles en Sonar, y también de los repositorios de *plugin* ya creados por empresas externas. Entre estos, a destacar el repositorio del *plugin* CNES Report. Este repositorio tiene un comportamiento similar en algunas funcionalidades que tiene que tener el *plugin* a crear. El objetivo es implementar funcionalidades que estuvieron en versiones anteriores de SonarQube, las cuales han pasado a formar parte de la versión *premium* del programa.

1.1 Funcionalidad del *plugin*

Objetivo: crear de un *plugin* para la versión 7.9.5 o superior de SonarQube.

Sonar ha cambiado respecto a sus anteriores versiones. Muchas de sus funcionalidades que eran importantes para la herramienta, han formado parte de la parte *premium*. A pesar de ello, Sonar dispone de una API pública, de la cual podemos realizar uso de ella. Dicho esto, una de las funcionalidades a recuperar, es la de ***Compare Projects***, la cual permitía seleccionar un número de métricas de Sonar para poder comparar sus valores junto a otras versiones o proyectos (Véase Figura 1). Esta funcionalidad se quiere mejorar añadiendo más métricas para comparar, y también la posibilidad de comparar los proyectos por fechas. Además, también se quiere mejorar la usabilidad en la interfaz gráfica.


Compare

Add metric ▾ Add project ▾

	DARKCHESS 1.0-SNAPSHOT 17:31	DARKCHESS 1.0.1-SNAPSHOT 23:34
Lines of code	1,756	1,756
Complexity	375	375
Comments (%)	1.9%	1.9%
Duplicated lines (%)	1.7%	1.7%
Issues	315	314
Coverage		
Complexity /file	8.9	8.9

Figure 1: *Compare Projects* de Sonar (*deprecated)*

Por otro lado, se quiere implementar una de las funcionalidades que aporta el *plugin* CNES Report. Al realizar las comparaciones de los proyectos, se quiere descargar los análisis de los proyecto. Esta opción otorgará las siguientes posibilidades:

- Exportar el análisis de un sólo proyecto.

- Exportar el análisis de diferentes proyectos, obteniendo la *delta* (δ) aritmética de sus diferencias.

Todavía no se han especificado los diferente métodos de exportación, pero el objetivo es abarcar un rango alto de diferentes métodos, para brindar la posibilidad de obtener datos de diferentes fuentes. Además, se tendrán que mostrar estos datos de una manera gráfica, para poder comparar los valores de una manera más visual.

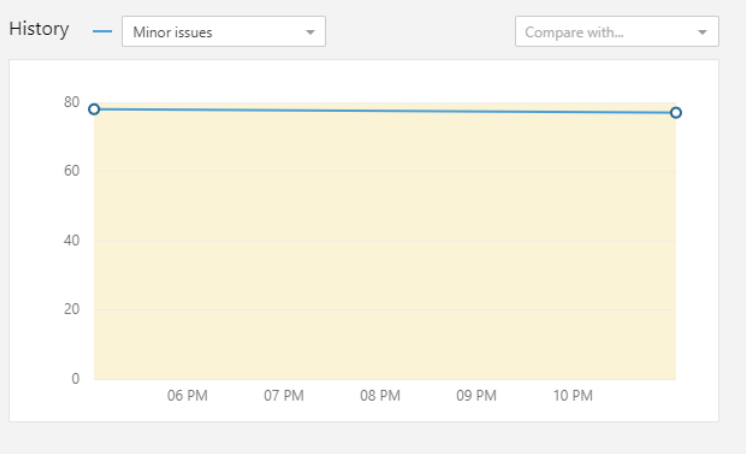


Figure 2: Historial de *Issues* en SonarQube 5.3

En esta Figura se puede ver la comparación de 1 sola métrica, pero lo idea es poder realizar gráficos de diferentes métricas.

1.2 Resultado final de funcionalidades

Debido a las complicaciones en el desarrollo del plugin que se especifican en la Sección 8, el programa está realizado para poder añadir estas funcionalidades, pero a raíz de la falta de tiempo no se ha logrado completar las diferentes funcionalidades:

- *Compare Projects* en la parte visual del plugin, pero sí en el *back-end*.
- Implementación del *Dashboard*.
- Gráficos para comparación, ya que no era un objetivo principal.
- Implementación completa del *front-end*

2. Estructura

En esta sección, se señalan las principales características de la estructura total del programa. Por ende, no se centra en explicar cada uno de los componentes que componen el sistema.

Para la estructura, no se dispone de mucha flexibilidad. Es necesario seguir la estructura que marca Sonar, para que pueda llegar a ser compatible con su aplicación. La estructura principal sería la siguiente:

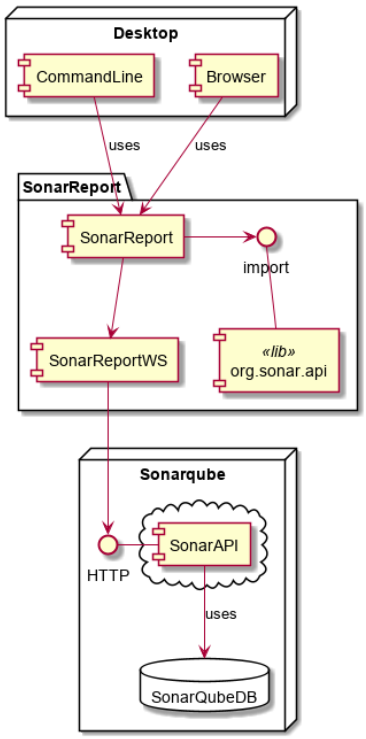


Figure 3: Diseño inicial de la estructura de la plataforma Sonar

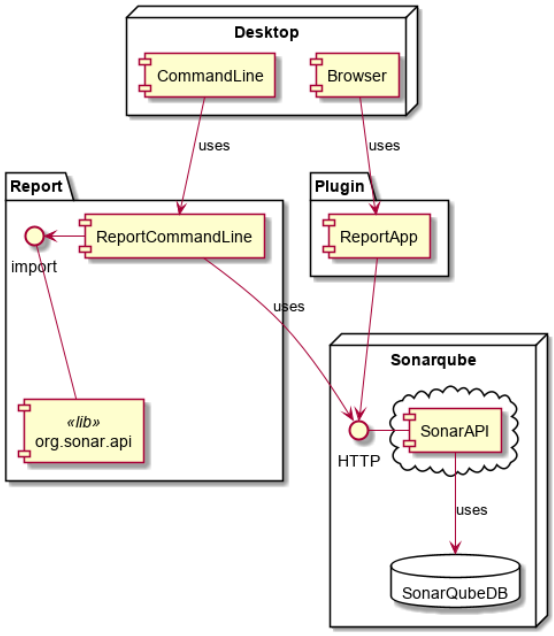


Figure 4: Diseño final de la estructura de la plataforma Sonar

2.1 Desktop

Es necesario dar la posibilidad de que Sonar se pueda ejecutar de diferentes sistemas. El objetivo es utilizar el modelo de MVC (Modelo Vista Controlador), para poder aislar el Modelo de la Vista. De esta manera, podremos hacer uso de la implementación del código desde diferentes vías de acceso. En principio, se dispondrá de dos métodos de acceso:

- ***CommandLine:*** como el proyecto tiene que ser un JAR, se podrá acceder al mismo utilizando los comandos de la terminal. De esta manera, en el caso de que no se pueda ejecutar el *plugin* desde la misma aplicación, dispondremos de esta posibilidad. A pesar de ello, está pensado de que este método sea más secundario, ya que no podría dar la posibilidad completa de seleccionar ciertas métricas, pero si que tendría la posibilidad de exportación.
- ***Browser:*** esta opción sería la principal, mediante el *plugin* de SonarQube. A pesar de que el *plugin* no esté de manera oficial en el *marketplace*, se puede añadir el JAR del proyecto de manera sencilla a la aplicación. El problema es que cada modificación acarrea que se tenga que recompilar el proyecto, y posteriormente sustituir el JAR. Esto genera que por cada cambio, se pierda entre 1-3 minutos.

2.2 Report

- **ReportCommandLine**: programa en Java, el cual tiene una arquitectura en Maven.
- ***org.sonar.api***: es el librería oficial de sonar. Dispone de un gran número de clases, las cuales hay que estudiar a fondo. Muchas de estas clases son obligatorias. Por ejemplo, para la creación de la interfaz gráfica es necesario importar 4-5 paquetes, para poder pasar la información a Sonar (ejemplo: *org.sonar.api.server.ws.WebService*).
- **SonarReportWS**: paquete necesario para gestionar las llamadas de la API. Tiene las instrucciones de las API, y dispone de los métodos que obtienen lo necesario de los JSON que obtiene como respuesta. También tiene que gestionar otros aspectos como la compatibilidad de versiones o la autenticación de la aplicación.

2.3 SonarQube

- **SonarAPI**: API creado por SonarQube. En teoría, dispone de todas las llamadas necesarias para poder implementar funcionalidades similares a la versión de pago de SonarQube. Las respuestas las realiza en JSON, y como toda API, es necesario realizar las llamadas en HTTP. Se han realizado pruebas externas para probar que la API funcione correctamente. Se ha comprobado con esta pequeña prueba que muchos de los métodos han quedado *deprecated*, pero parece que este cambio ha sido para mejorar su funcionalidad, no para restringir la versión de SonarQube gratuita. Aún así, para afirmar esta sentencia es necesario realizar muchas más pruebas.
- **SonarQubeDB**: base de datos que utiliza SonarQube, para poder gestionar los análisis que ha realizado previamente. Esta base de datos es PostgreSQL. En principio, se hará uso de la API, ya que no es una prioridad la velocidad de las operaciones del *plugin*. También hay que destacar que en el caso de utilizar la BD sería necesario aprender su estructura, lo cual alargaría la creación del *plugin*. Por otro lado, los tiempos de respuesta del plugin son relativamente cortos, de una media de 20ms.
- **SonarQube**: para que SonarQube funcione correctamente, tiene que realizar una seria de operaciones, como guardar el análisis en la Base de Datos. Para realizar el análisis, es necesario que ejecute dos tipos de procesos:

- *Sensors:* son sensores que son capaces de crear nuevos *measures*, computar sus métricas con cada recurso : método, archivo o paquete. Luego, todo esto se guarda en la BD.
- *Decorators:* cuando acaban de realizar sus operaciones los *Sensors*, se activan los *Decorators*, para poder realizar su análisis.

2.4 Plugin

- **ReportApp:** aplicación principal del plugin, implementada con la librería React de JavaScript. Esta clase se comunica con otras, para poder recopilar la información necesaria para la realización del reporte. Después, procesa la información y la envia al WebService.

3. Diagrama de clases

Como se puede ver en el Diagrama de Clases de la Figura 6 y en el Diagrama de Componentes de la Figura 4, el proyecto está dividido por segmentos. Estos segmentos están compuestos por ficheros que tienen una función concreta.

- **View:** Paquete "ficticio". Se ha utilizado para realizar pruebas puntuales, en las cuales era necesario realizar el Debug. Cabe destacar que no es posible realizar un Debug del *plugin*, por lo que cuando ha sido necesario encontrar algún error en la creación del reporte, se han realizado pruebas básicas con clases auxiliares.
- **Configuration:** paquete que contiene los archivos necesarios para la compatibilidad con Sonar y la configuración seleccionada por el usuario.
  - *ExportConfiguration:* según el usuario seleccione los diferentes métodos de exportación, se habilitarán los *boolean* de esta clase.
  - *SonarQubeServer:* dispone de la configuración del SonarQube que se está configurando. Sigue el patrón.
  - *ReportConfiguration:* tiene los atributos de cada uno de los proyectos de los cuales se quiere hacer un análisis.
  - *SonarRequestList:* contiene la lista de las configuraciones de los reportes que hay que hacer, y es la clase encargada de llamar para crear los reportes. Es la primera clase a la cual hay que llamar desde la vista.
- **Factories:** contiene 3 clases que siguen el patrón de diseño Factory.
  - *ReportFactory:* se encarga de crear los exportables. Utiliza la clase ExportConfiguration y llama los respectivos métodos de exportación.
  - *ProviderFactory:* crea todas las clases de *AbstractProvider*.
  - *ReportModelFactory:* es la clase encargada de gestionar todas las llamadas a las otras clases, para crear un *Report*.

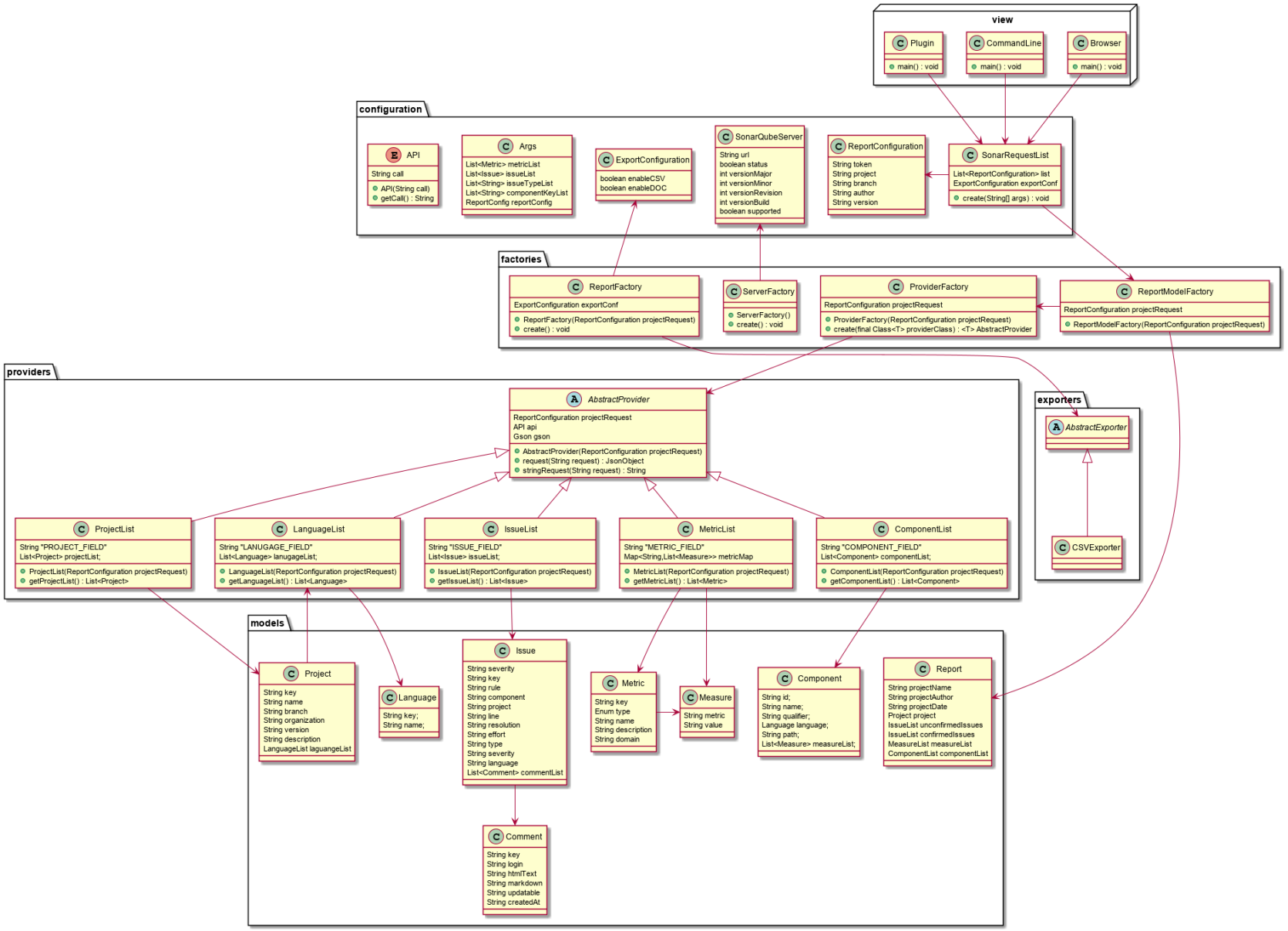


Figure 5: Diagrama de clases inicial

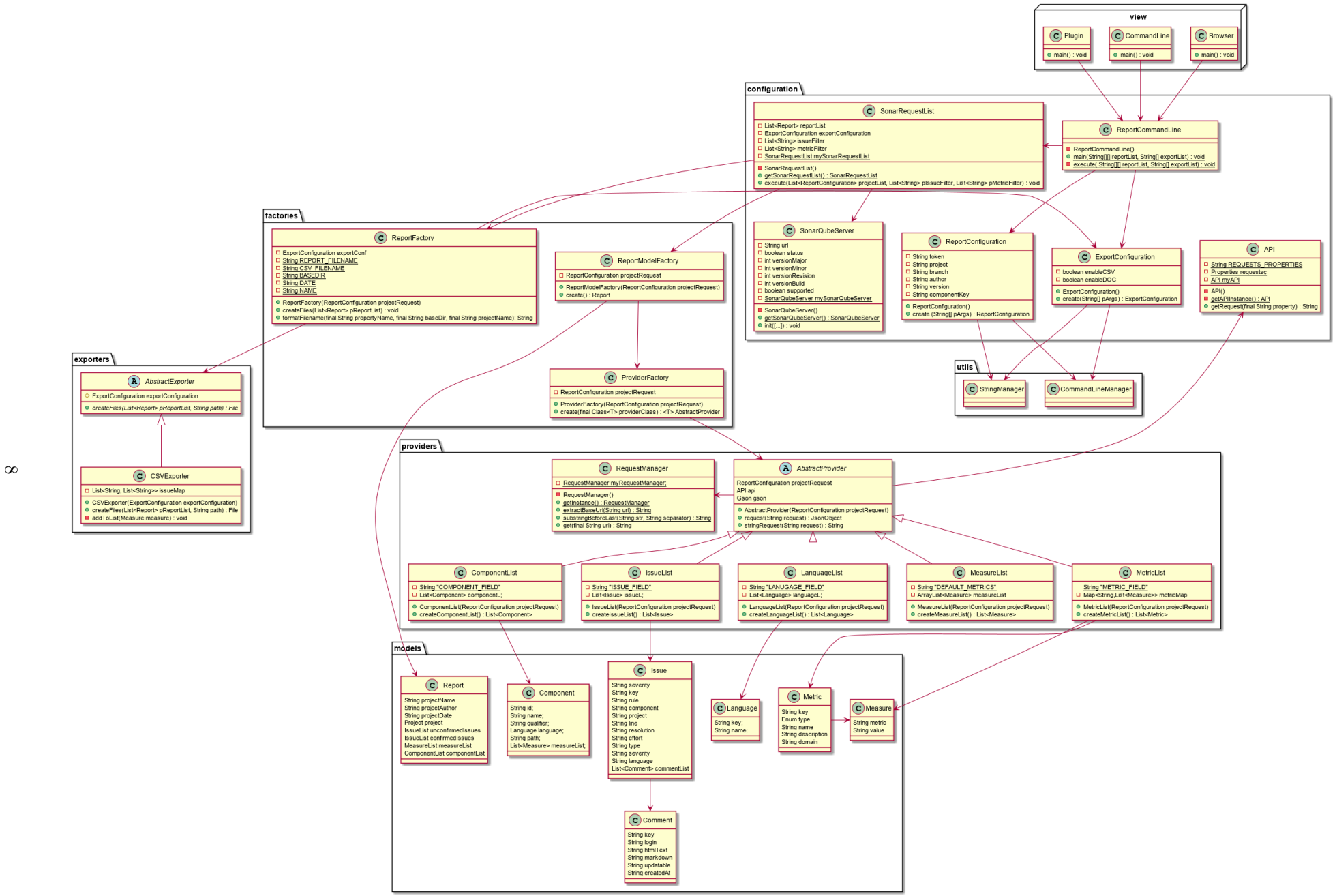


Figure 6: Diagrama de clases inicial

- **Providers:** paquete que aúna la lista de los elementos de los modelos. Se agrupan bajo una misma clase abstracta.
  - **AbstractProvider:** clase abstracta que gestiona las llamadas a la API. Para las llamadas de la API se utiliza GSON, para obtener los objetos de JSON.
- **Models:** tiene todas los objetos que forman parte del Report. Están relacionados con sus propias listas del paquete *providers*.
- **Exporters:** contiene todas las clases para los diferentes métodos de exportación.

4. Dependencias

En esta sección se enumeran las dependencias del proyecto. Es necesario el conocimiento de este apartado ya que las dependencias son muy sensibles entre ellas. Las dependencias que se definen en el pom.xml son las siguientes:

Nombre	Funcionalidad	Versión
gson	Llamadas a la API	2.8.5
sonar-ws-client sonar-ws	Librería de Sonar de WS	5.1 7.9.3
commons-csv	Exportar formato CSV	1.5
sonar-plugin-api	Librería de Sonar del plugin	7.9.3
httpclient	Comunicación con HTTP	4.5.6
commons-cli commons-lang3 poi-ooxml poi-ooxml-schemas ooxml-security commons-io maven-shade-plugin maven-eclipse-plugin frontend-maven-plugin	Maven / Apache	1.4 3.8.1 1.2 3.17 1.1 2.6 3.2.1 2.91.9.1
sonar-testing-harness	Test unitarios Sonar	7.9.3
junit-jupiter-api	JUnit (test unitarios)	5.3.1
sonar-packaging-maven-plugin	Compilador Maven para Sonar	3.5.1
jQuery	jQUery	1.11.0
node	Node JS	14.0.24
yarn	Yarn	1.22.4

Importante: el paquete native2ascii-maven-plugin indica un error que es un bug, por lo que no hay que darle importancia.

También se incluyen otros paquetes para el apartado de *build*, utilizando el paquete de módulos JavaScript llamado **Webpack**. Los ficheros a revisar son los siguientes:

- package-lock.json
- package.json
- uarn.lock
- env.js
- build.js (config.scripts)
- webpack.config.js
- webpack.config.prod.js

5. Compare Projects

Como se ha comentado antes, en la última versión de SonarQube no está disponible esta opción. Sirve para poder seleccionar de un listado de métricas, para poder realizar la comparación. El listado de métricas está compuesto por:

- LOC
- Issues
  - Blocker Issues
  - Critical Issues
  - Major Issues
  - Minor Issues
  - Info Issues
  - False Positive Issues
  - Won't Fix Issues
- Bugs
- Vulnerabilities
- Code Smells
- Security Hotspot
- Technical Debt
- Dup. blocks
  - Dup. files
  - Dup. lines
  - Dup. lines (%)

- Rating
  - Reliability Rating
  - Security Rating
  - Maintainability Rating
- Análisis por clases
  - Test por clases
  - Cubrimiento por clases

Todas estas métricas tienen que ser aplicadas en diferentes:

- Proyectos
- Versiones de proyectos
- Fechas de proyectos

6. Métodos para exportar

Aquí el listado de los diferentes métodos de exportación. Unos métodos de ejemplo serían los siguientes.

- Excel (xlsx)
- CSV
- JSON
- Documentación
- Gráficos

En este caso, solo se ha implementado la funcionalidad de CSV, pero sigue siendo extensible a todos los demás. Para ello, únicamente sería necesario añadir una nueva clase que implemente los métodos de la clase **AbstractExporter**. Al utilizar el patrón Abstract, simplemente es necesario implementar el método *create()*, para poder exportar el fichero en la extensión especificada.

7. Implementación del código

En este capítulo se redacta el funcionamiento interno del programa. El objetivo es explicar todo lo posible su funcionamiento, para que en un futuro se pueda continuar con su implementación. También se especifica como se podrían aplicar funcionalidades sencillas, y donde se deberían de realizar los cambios.

El *plugin* está dividido en 2 partes bien diferenciadas: **report** y **plugin**

- *Report: backend* del plugin que realizar el reporte por cada proyecto que tenga como parámetro, para poder crear los ficheros de exportación correspondientes. Se trata del . Se puede encontrar en la carpeta: *src/main/java/report*.
- *Plugin:* parte del *frontend* del plugin que se encarga de gestionar el plugin de SonarQube, para poder recibir los datos por parte del usuario. Se puede encontrar en la carpeta: *src/main/java/plugin* y con su respectiva documentación de código.

7.1 Report

7.1.1 ReportCommandLine

La clase principal es **ReportCommandLine** que se encuentra en la carpeta *configuration*.

Listing 1: "Clase ReportCommandLine"

```
public final class ReportCommandLine {

    private ReportCommandLine(){}

    public static void main(String[][] reportList, String[] exportList)  {
        try {
            execute(reportList, exportList);

        } catch (Exception e) {
            System.out.println("Error in main " + e);
            System.exit(-1);
        }
    }

    public static void execute( String[][] reportList, String[] exportList){

        ReportConfiguration reportConf;
        ArrayList<ReportConfiguration> reportConfList = new ArrayList<
 ReportConfiguration>();
        for (int i = 0; i < reportList.length; i++) {
            reportConf = new ReportConfiguration();
            reportConf = reportConf.create(reportList[i]);
            reportConfList.add(reportConf);
        }

        ExportConfiguration exportConf = new ExportConfiguration();
        exportConf = exportConf.create(exportList);
        SonarRequestList sonarRList = SonarRequestList.getSonarRequestList();
        sonarRList.setExportConfiguration(exportConf);

        ArrayList<String> pIssueFilter = new ArrayList<String>() {{
            add("");
            add("");
            add("");
            }
```

```
        };
        ArrayList<String> pMetricFilter = new ArrayList<String>() {{
            add("nlock");
            }
        };
        sonarRList.execute(reportConfList, pIssueFilter, pMetricFilter);
        System.out.println("Report generation: SUCCESS");
    }

}
```

Esta clase está referencia en el pom.xml como la clase principal del JAR. Como se puede ver, la variable *app.mainClass* es referenciada, la cual tiene el valor de la clase principal. Por lo tanto, en caso de querer cambiar esta clase como un *main*, sería necesario cambiar la variable.

Listing 2: "Propiedad definida en pom.xml"

```
<plugin>
    <artifactId>maven-shade-plugin</artifactId>
    <version>3.2.1</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
            <configuration>
                <transformers>
                    <transformer implementation="org.apache.maven.plugins.shade.
 resource.ManifestResourceTransformer">
                        <mainClass>${app.mainClass}</mainClass>
                    </transformer>
                </transformers>
                <shadedArtifactAttached>false</shadedArtifactAttached>
                <createDependencyReducedPom>false</createDependencyReducedPom>
                <!--<minimizeJar>true</minimizeJar>-->
            </configuration>
        </execution>
    </executions>
</plugin>
```

La clase ReportCommandLine tiene como 2 parámetros de entrada en su método *main()*.

- *reportList:* lista de listas que contiene los datos de los proyectos a realizar el reporte. En cada *array* se guardan los datos necesarios para identificar a un proyecto:
  - *ProjectKey* (clave asignada del proyecto)
  - *Branch* (rama del proyecto)
  - *Version*
  - *Token*
- *exportList:* lista que contiene los parámetros necesarios para realizar la creación de ficheros. En este caso, no se han utilizado muchos ya que solo se ha probado con el formato CSV, para no perder el tiempo del desarrollo. Los elementos de la lista son:

  - *Output* (dirección de salida de los ficheros)
  - *EnableCSV* (habilitar los ficheros en el formato CSV)

A estos 2 parámetros, sería necesario añadir otros 2:

- *metricFilter:* lista de las métricas seleccionadas para realizar el reporte.
- *issueFilter:* lista de los los parámetros para obtener las *issues*.

No se han añadido ya que no se ha implementado la selección de selección de estos parámetros en la parte *frontend* del plugin. Por consiguiente, sería necesario eliminar las listas creadas en el método *execute()* llamadas *pIssueFilter* y *pMetricFilter*, ya que se recibirían como parámetros.

Método *execute(reportList, exportList)*

- Objetivo: realizar los pasos necesarios para llamar a la clase principal del reporte, SonarRequestList.[1]
- Parámetros:
  - Lista de proyectos.
  - Lista de configuración de exportación.
  - A añadir filtros de métricas e *issues*.
- Salida: *void*
- Funcionamiento:
  - Se crean las instancias de ReportConfiguration. Utilizamos estos objetos durante todo el proceso de creación de reporte, para no tener listas de Strings. De esta manera, se favorece el uso de de orientación a objetos.
  - Se crea la instancia única de ExportConfiration.
  - Se añade la lista de ReportConfiration y la instancia de ExportConfiration a SonarRequestList y se llama a su método principal, *execute()*.

7.1.2 SonarRequestList

Clase principal del reporte, que se encuentra en la carpeta *configuration*.

Su función es tener las instancias de las clases principales y los patrones *factory*.

Método *execute(reportList, exportList, pIssueFilter, pMetricFilter)*

- Objetivo: generar los ficheros a partir de una lista de Report.
- Parámetros:
  - Lista de proyectos.

[1]Se diferencia la clase principal del *report* y la clase principal del reporte. La clase principal del *report* se refiere a aquella clase que se tiene que llamar externamente para realizar el apartado de reporte. Por otro lado, llamamos clase principal del reporte a aquella que se encarga de generar los reportes.

  - Lista de configuración de exportación.
  - Lista de métricas para el filtrado.
  - Lista de *issues* para el filtrado.

- Salida: *void*.
- Funcionamiento:
  - Como se ha comentado anteriormente, recibe los parámetros de ReportCommandLine. Se encargar de crear la instancia de **SonarQubeServer**, el cual contiene la configuración del Sonar. Es importante ya que gestiona el control de compatibilidades del plugin. En un futuro, sería interesante disponer de diferentes archivos con llamadas a la API, para poder tener un plugin adaptable a diferentes versiones.
  - Se ejecuta por cada ReportConfiguration (archivo de configuración de cada proyecto), un método de **ReportModelFactory**. Este crea el Report (reporte) por cada ReportConfiguration.
  - Se almacena esta lista de Report para a continuación generar los ficheros con **ReportFactory**.

7.1.3 ReportModelFactory

Clase que implementa el patrón *Factory*, la cual crea todas las listas de modelos (paquete *provider* en el proyecto). Para ello, llama a cada una de las Listas (NombreModeloList, ejemplo "LanguageList"), mediante su patrón Factory: **ProviderFactory**.

Por consiguiente en el caso de querer añadir más listas de modelos, es necesario instanciarlo en esta clase, mediante el método ***create()***. Mediante este método, se consiguien todas las listas para poder añadir a la clase Report.

Método *create()*

- Objetivo: llamar a todos los Provider para añadir las listas al Report.
- Parámetros: *void*.
- Salida: instancia de Report.
- Funcionamiento:
  - Se crean todos los Provider mediante el método *create()* de ProviderFactory.
  - Por cada uno de los Provider, se llama a su respectivo *createNombreModeloList()* (ejemplo, *createMeasureList()*).
  - Se devuelve la instancia de Report, previamente añadiéndose todas las listas.

7.1.4 AbstractProvider

Patrón *abstract* para crear diferentes clases, que proporcionan las listas de los modelos. Estas listas realizan llamadas a la API.

¿Cómo cambiar una llamada a la API? Para cambiar las llamadas de las API hay que seguir un número de pasos:

- Cambiar el fichero **request.properties**. En este fichero, como ya se ha comentado, es necesario instanciar las URL necesarias para las llamadas a la API. Tanto como si se quiere añadir una nueva como si se quiere modificar una llamada, es necesario realizar el cambio.
- Realizar cambios en el método ***createX()*** en la clase Provider afectada. Para ello, es necesario tener en cuenta el número de parámetros. Es necesario añadir al método *String.format()* el número de parámetros que tiene la URL, separado por comas.
- Después, hay que revisar que la conversión al Objeto modelo sea el adecuado.En el caso de se tengan que añadir más atributos, es necesario modificar la clase correspondiente al modelo.
- Para finalizar, hay que tener el cuenta que si el cambio se realiza sobre los ***issues*** o sobre los ***metrics***, es necesario revisar los parámetros de entrada del método execute() (ReportCommandLine) y execute() (SonarRequestList).

Se recomienda para realizar las pruebas de una llamada a la API utilizar un programa auxiliar en blanco. Este programa puede ser muy sencillo (HTML), y sirve para revisar la trazabilidad de las llamadas.

Listing 3: "Ejemplo programa para probar llamadas a la API"

```
@host = http://localhost:9000

GET {{host}}/api/authentication/validate
###
GET {{host}}/api/duplications/show?key=darkchess:src/main/java/com/c0nrad/
 darkchess/App.java http/1.1
###
GET {{host}}/api/issues/search?componentKeys=com.c0nrad.darkchess:darkchess:src\
 main\java\com\c0nrad\darkchess\datastore
###
GET {{host}}/api/projects/index?versions=true
```

8. Problemas en el desarrollo

En esta sección aparecen los errores/*bugs* que han hecho ralentizar el desarrollo de la aplicación. La mayoría del tiempo del desarrollo de la aplicación ha sido arreglando errores, ya que no se dispone de casi ninguna información. Unos pocos de estos errores se han arreglado con la ayuda del foro de Sonar. Adyacente a cada error, se indica el tiempo en horas aproximado que se ha tardado en arreglar cada error.

8.1 Instalación de Sonar.

- **Horas:** 4h.

- **Fase de desarrollo:** Inicial.
- **Desarrollo:** Al principio, para empezar con el desarrollo de la aplicación, se propuso realizar pruebas en la plataforma para obtener más información. Además, también se tenía que realizar un estudio sobre diferentes versiones para obtener un número mayor de conclusiones. Para ello, se eligieron dos versiones:
  - *Sonarqube-5.3:* a partir de esta versión, se cambiaron muchos aspectos del programa. Muchas funciones pasaron a formar el grupo *deprecated* (obsoletos), para que la compañía pudiese beneficiarse de su versión de pago. Además, se sabía de antemano que la empresa había hecho uso de ella, por lo que había aspectos que eran más sencillos de explicar.
  - *Sonarqube-9.8.3:* es la última versión LTS recomendada por la compañía.

  Los problemas vinieron a la hora de instalar la segunda versión, ya que Maven se hacía un "lío" con las versiones de Java y de la propia Maven.
- **Solución:** seleccionar una versión de Maven estable, junto a una selección de la variable de entorno dinámica.

8.2 Versiones de POM.

- **Horas:** 30h.
- **Fase de desarrollo:** Avanzada
- **Desarrollo:** Al no tener conocimientos previos de algunas de las tecnologías utilizadas, se tardó más tiempo de lo establecido para que funcionase correctamente el ejecutable de la aplicación. En primer lugar, se hizo un estudio de la información oficial de Sonar "¿Cómo crear un plugin?". El problema es que la información base es correcta, pero corta, y además dispone de un repositorio en GitHub, con un plugin de prueba. Este repositorio ha sido de gran ayuda para poder ejecutar en modo *Debug* el plugin, para así poder aprender el flujo de eventos de la compilación del plugin. De esta manera, se podía comprobar el flujo de ejecuciones que tenía el programa.

  El problema es que la versión es antigua, ya que el código fue creado en el año 2017. Por lo cual, las versiones de las dependencias que se hacen uso en el pom.xml, no son validas. Para ello, se ha tenido que estudiar cada dependencia una a una, para poder saber si era necesaria esa dependencia o no. Aún así, había paquetes que aún con su última versión no funcionaban, por lo que se ha tenido que buscar en repositorios de GitHub de plugin oficiales de Sonar (y que estos estuvieran actualizados).

  Las dependencias que daban problemas eran **yarn** y **webpack**, además de la dependencias oficial de sonar **sonar-maven-plugin**.
- **Solución:** Seleccionar versiones de las dependencias en base al método "prueba y error".

8.3 Uso de webpack.

- **Horas:** 10h.
- **Fase de desarrollo:** Avanzada.
- **Desarrollo:** Al no haber utilizado webpack con anterioridad, se tuvo que estudiar la librería. Una vez estudiado, se tuvieron que crear los ejecutables para poder hacer el ***build*** del proyecto.
- **Solución:** Utilizar bien los *script* de creación de ficheros y suprimir un *script deprecated* que no funcionaba.

8.4 Migración de WebService a JavaScript

- **Horas:** 15h.
- **Fase de desarrollo:** Avanzada
- **Desarrollo:** Para implementar un *plugin* de SonarQube, existen diferentes tecnologías a implementar; como pueden ser Ruby-on-Rails o JavaScript. En un primer lugar, se decidió utilizar unas librerías de Java; oficiales de Sonar, para implementar un webservice. Al ser Java el lenguaje de código más utilizado, fue el seleccionado para utilizar. Una vez realizado un página web de prueba, no funcionaba. Además, se comprobó que la librería era algo limitada, por lo que no se iba a poder cumplir con el objetivo de crear un *Dashboard*
- **Solución:** utilizar JavaScript con Html para la creación de webs. El Html brinda más posibilidades para la implementación del *dashboard*, por la ayuda de las tablas y demás objetos utilizables.

8.5 Página web en blanco

- **Horas:** 25h
- **Fase de desarrollo:** Avanzada
- **Desarrollo:** Es el error que más se tardó en solucionar, debido que no se sabía la raíz del problema. Se intentó solucionar de diferentes maneras; como puede ser cambiando versión de dependencias, revisión de código o cambiando la versión de SonarQube. Aún así, todo esto fue en vano, ya que no se solucionó de ninguna manera. Para realizar estas pruebas, se utilizó el repositorio propio de GitHub, con un *commit* estable, y realizando la acción de *pull* cada vez que se probaba cada arreglo.
- **Solución:** Para que funcione el plugin en sonar, la librería de sonar-maven-plugin crea diferentes carpetas en el JAR (las cuales no se pueden ver). Una de las carpetas que crea es una llamada **api**, en la cual es donde se coloca el programa principal para realizar el reporte. Por lo cual, si no estaba en el lugar exacto, Sonar informaba de que no encontraba el fichero. Para solucionarlo, se indicó donde se encontraba la carpeta de origen y cual era su destino.

8.6 Error de url en la generación de reportes

- **Horas:** 7h
- **Fase de desarrollo:** Avanzada
- **Desarrollo:** Una vez solucionado el error anterior, Sonar no encontraba el archivo para realizar el reporte. El error que proporcionaba era el siguiente : "Errors":['msg':Unknown url sonarqube]". Para replicar este error era necesario pulsar en el botón para generar los ficheros. Buscando la solución a este error, se encontraron varios hilos similares en el foro oficial de SonarQube. A pesar de que en el foro oficial, desarrolladores propios de la compañía daban una solución, no solucionaba el problema. La solución era que se tenía que cambiar de versión de SonarQube a la última, en la cual aseguraban que se solucionaba el problema. La versión en quistión es: sonarqube-8.4.1.35646. Gran parte del tiempo para solucionar este error, aparecía el mensaje que aparece en la siguiente Figura.

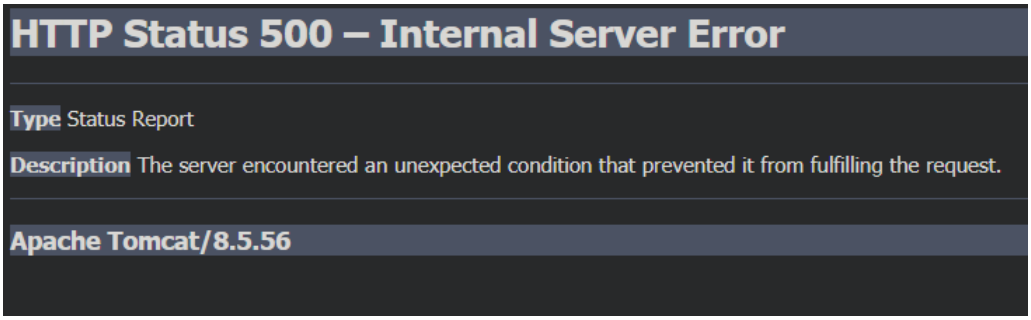


Figure 7: Error para la generación de reportes

- **Solución:** La solución fue que en un fragmento del código, al indicar la carpeta de "api", se indicaba una carpeta que no existía. Era necesario escalar otra carpeta, lo cual no tenía sentido ya que no era lo correcto. Esto puede ser porque no se exportan la misma cantidad de carpetas al Jar, por lo que el orden de las carpetas no era lo correcto.

8.7 Creación de Zip para la descarga de ficheros

- **Horas:** 8h.
- **Fase de desarrollo:** Avanzada
- **Desarrollo:** Para la descarga de archivos de la página web, era necesario hacer uso de un archivo de comprensión. Para ello, se utilizaba una librería de Java para Zip. Esta librería funcionaba, pero el Zip daba un error al abrirlo. Además, ya informaba con el formato del nombre, ya que el archivo se llamaba ***null***.
- **Solución:** Se utilizó otra librería de comprensión tras no poder solucionar el error anterior. La librería se llama ZipFolder, y después de configurarlo, funcionaba correctamente.

8.8 React

- **Horas:** 20h
- **Fase de desarrollo:** Final
- **Desarrollo:** Para la realización del plugin, se ha utilizado la librería de JavaScript React. En principio, en los documentos oficiales de Sonar, se redactaba que solo se podían seleccionar unos pocos de lenguajes para el *frontend*. Por ello, se seleccionó JavaScript y su librería React. Debido a la falta de conocimientos previos de esta librería, no se pudo avanzar mucho en su desarrollo. Además, uno de los problemas encontrados era que se necesitaban listas de estados, lo cual era un problema avanzado de programación web. Este problema combinado con la falta de conocimiento de la librería, hizo que no se pudiese concluir la tarea. Por consiguiente, no se han implementado parte de las funcionales del plugin.
- **Solución:** No se ha encontrado solución, pero se ha dejado parte del código desarrollado comentado.

9. Calidad del Código

Para finalizar, ya que se dispone de la herramienta, se ha utilizado SonarQube para el análisis del código. De esta manera, se ha mejorado la **calidad del software**, con el objetivo de que se pueda extender el programa en el futuro. El reporte indica una gran cantidad de *Bad Smells*, pero este debe a que no se ha utilizado un *Logger*, porque no era visible en la consola de ejecución del programa.

Además, también notifica de métodos que no se utilizan y fragmentos de código comentado, los cuales no se han eliminado para que se pueda continuar con la implementación en el futuro.

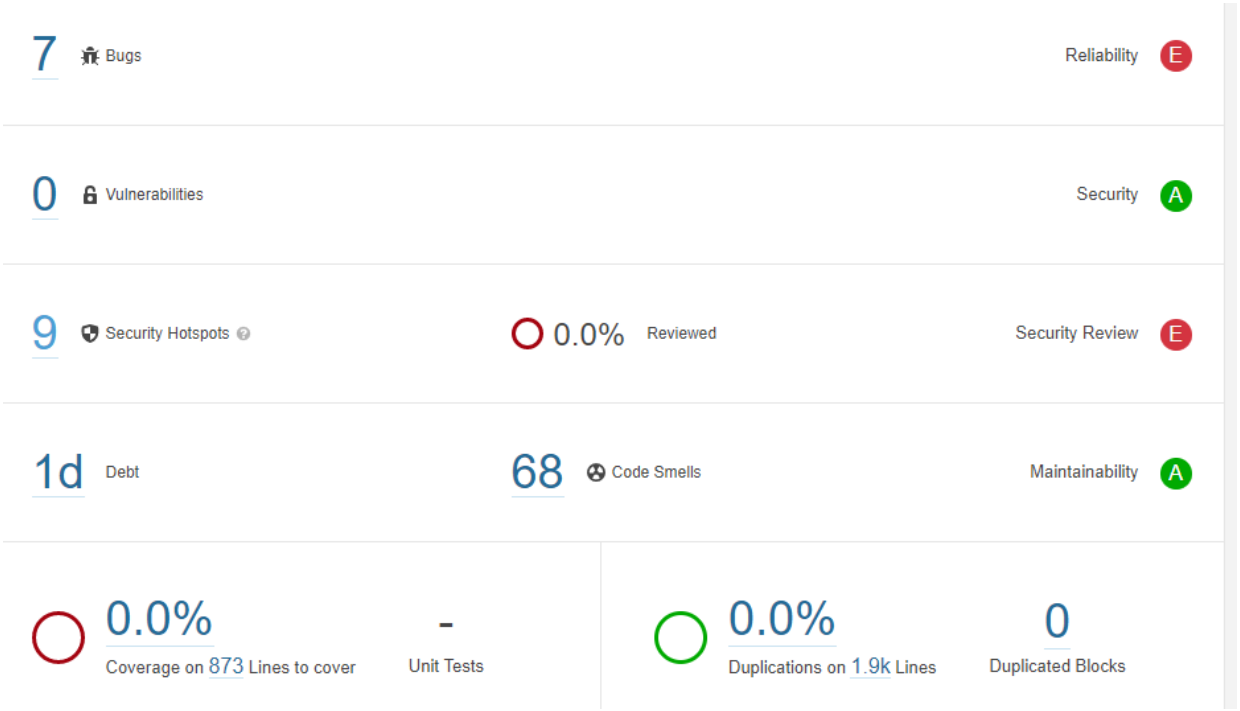


Figure 8: Análisis del código por SonarQube