INFORME DE CALIDAD DE PRODUCTO

Autor	Fernando Arruti Samperio
Número de Sprint	3
Historia de Usuario	Ordenar gasolineras
Fecha	22/11/2016

INTRODUCCIÓN

El objetivo de la actividad es realizar un análisis de calidad interna de producto software mediante la herramienta SonarQube.

Este análisis será aplicado al proyecto “AppGasolineras”, sobre la historia de usuario “Ordenar gasolineras”, en el cual se aplicarán las medidas convenientes para mejorar y/o maximizar la calidad del proyecto.

ANÁLISIS

Para el primer análisis, se procederá a realizar el escaneo haciendo uso de la configuración por defecto del Quality Gate y del Quality Profile.

Metric	Over Leak Period	Operator	Warning	Error
Coverage on New Code	Always	is less than		80.0%
New Bugs	Always	is greater than		0
New Vulnerabilities	Always	is greater than		0
Technical Debt Ratio on New Code	Always	is greater than		5.0%

Projects

Every project not specifically associated to a quality gate will be associated to this one by default.

Ilustración 1: Quality Gate

Sonar way

Rules	Active	Inactive
Total	268	111
Bugs	89	13
Vulnerabilities	18	13
Code Smells	161	85

Ilustración 2: Quality Profile

Fijándonos primero en la calificación SQALE, daremos prioridad a las medidas cuya calificación sea D o peor, ya que eso quiere decir que estamos ante un ratio de deuda técnica superior al 50%.

A la hora de analizar los bugs, se analizarán aquellos que tengan una severidad bloqueante, crítica o mayor. Haremos lo mismo con los “Code Smells”, analizando los bugs que tengan esa misma severidad.

En el caso de que existan duplicaciones de código, se analizarán y se intentarán eliminar.

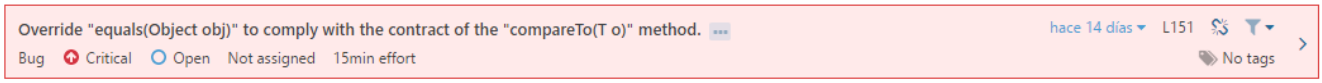
Una vez revisado esto, se procederá a la corrección de los detalles referidos a la complejidad del código, la documentación y a los bugs, vulnerabilidades y “Code Smell” poco severos.

RESULTADOS DEL ANÁLISIS

En el análisis, contemplamos que el Quality Gate no ha pasado. Esto es debido a que se comparan los fallos respecto a la versión anterior, y en este caso al haber añadido nuevas funcionalidades a nuestro código, se han añadido defectos a este.

Vemos que nos aparecen primeramente cinco bugs y seis vulnerabilidades.

Bugs:



Este bug es producido al no seguir la regla de hacer un uso adecuado del método compareTo. Este bug será ignorado al no poder ser solucionado de una manera adecuada en nuestro código, ya que al aplicar la regla con el cambio aportado por el sonar, el funcionamiento no es el esperado. Es por esto, por lo que la regla se eliminara del conjunto de reglas aplicadas al hacer uso del sonar.

Este error se produciria al no hacer un log de la excepción. Para solucionarlo unicamente usaremos el comando log en el catch de esa excepcion para logear el error.

Este error se produce al no cerrar adecuadamente la lectura de los datos. Lo solucionariamos cerrando el InputStream justo antes de finalizar el método.

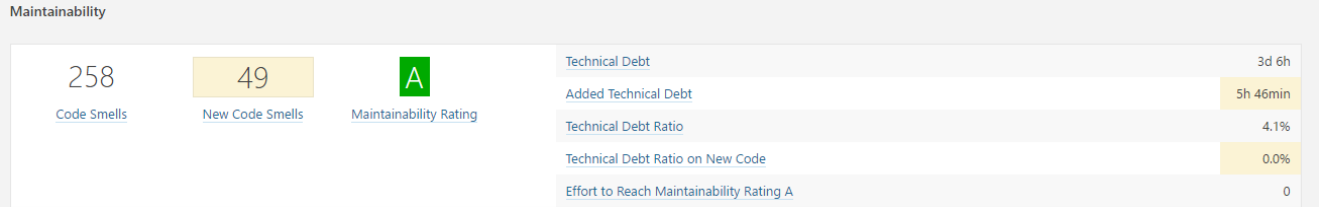
Vulnerabilidades:

Este error se produciria al no hacer un log de la excepción. Para solucionarlo unicamente usaremos el comando log en el catch de esa excepcion para logear el error.

Deberemos usar una excepcion propia en vez de una generica para solucionar este error.

Para solucionar este error, deberíamos de hacer un atributo de nuestra clase final.

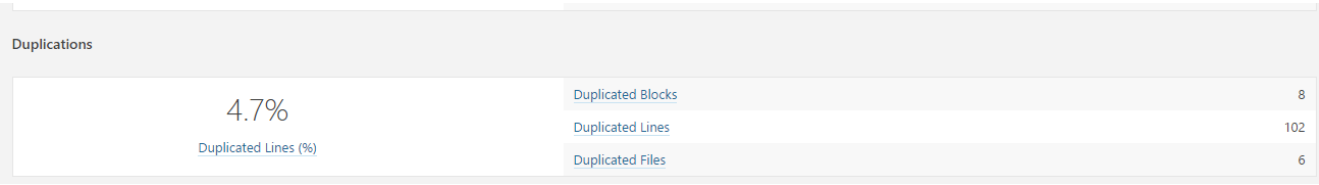
Mantenibilidad:



La calificación obtenida respecto a la mantenibilidad del código es A, por lo tanto buena. La deuda técnica es de 3d6h, con un ratio técnico de deuda del 4.1%.

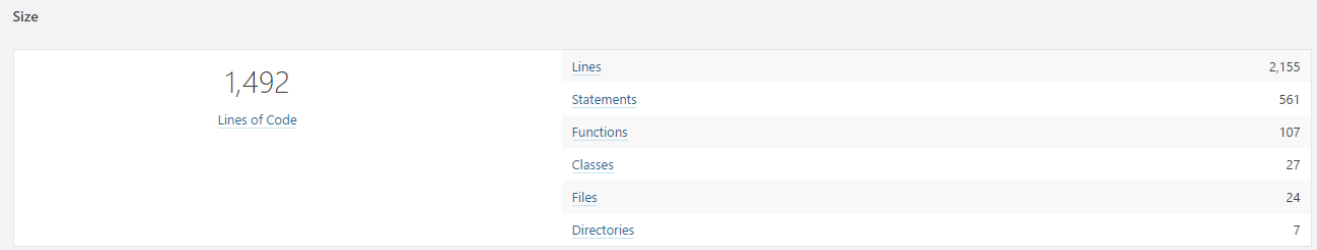
Respecto a la anterior versión, la deuda tecnica ha aumentado en 5h y 46 minutos.

Duplicaciones:



El número de duplicaciones es del 4,7%, lo que es un porcentaje aceptable, ya que no supera el 5%, lo que podria considerarse elevado.

Tamaño



Respecto al tamaño del proyecto, vemos que es el adecuado ya que el numero de clases y ficheros es muy similar, por lo que no estamos cargando en exceso a ningún fichero. La aplicación contiene 1492 lineas de codigo, 107 funciones y 27 clases.

Complejidad

Ilustración 3: Complejidad general

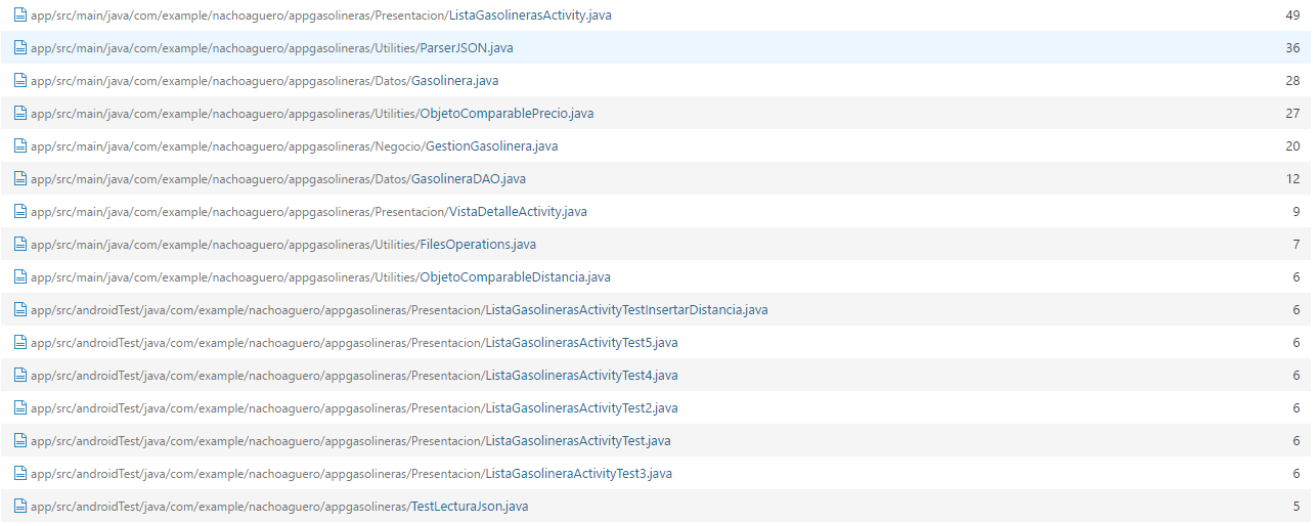
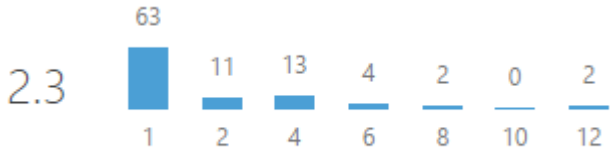


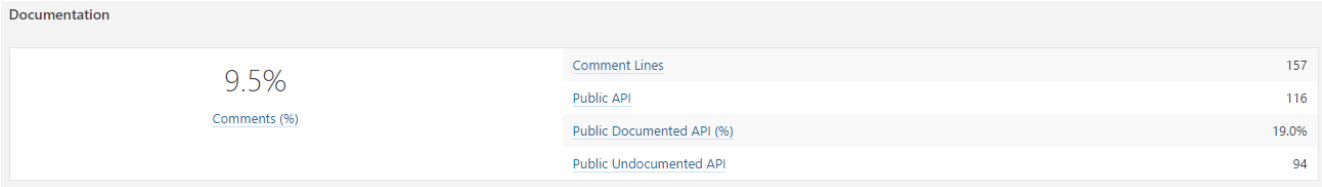
Ilustración 4: Complejidad por fichero

La complejidad general del proyecto vemos que es buena por lo general, aunque si entramos a ver la complejidad más en detalle, vemos que hay cinco ficheros que tienen una complejidad alta, por lo que habria que modificarlos e intentar modularizarlos para repartir esta complejidad entre varios ficheros.



Por funciones, vemos que hay una función que tiene una complejidad bastante alta, y el resto tienen una complejidad bastante repartida. Para solucionar esta complejidad, intentaremos modularizar esta función, e intentar repartir esta complejidad entre otras funciones. El resto de funciones al tener una complejidad inferior a diez no tendriamos porque modificarlas.

Documentacion



El proyecto viendo los resultados del análisis, vemos que solamente está documentado al 9,5%, por lo que habría que mejorar y documentar más partes del código para intentar que este documentado al menos más de la mitad.

PLAN DE MEJORA

Para mejorar el código de nuestro proyecto, se procederá a seguir los siguientes pasos, teniendo en cuenta los resultados obtenidos anteriormente.

1. Corregir bugs y vulnerabilidades con una escala D o superior.

2. Documentar las clases más complejas e importantes de nuestra aplicación.

3. Eliminar en la medida de lo posible la deuda técnica que ha mostrado el análisis durante el escaneo.

4. Intentar reducir el número de duplicaciones del código.