INFORME DE CALIDAD DE PRODUCTO

Autor	Fernando Arruti Samperio
Número de Sprint	2
Historia de Usuario	Insertar distancia en la información de gasolineras
Fecha	07/11/2016

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 “Insertar distancia en la información de 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

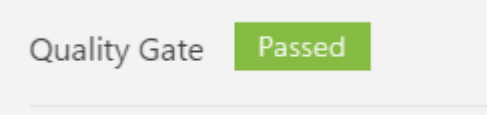
Fijándonos primero en la calificación SQALE, daremos prioridad a las medidas cuya calificación sea D o peor, ya que eso quiere decir que estamos ante un ratio de deuda técnica superior al 50%.

A la hora de analizar los bugs, se analizarán aquellos que tengan una severidad bloqueante, critica 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 analizaran y se intentaran 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

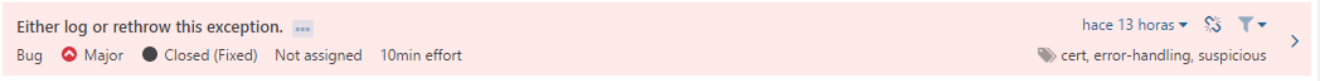


En el análisis, contemplamos que el Quality Gate ha pasado. Esto es debido a que se comparan los fallos respecto a la versión anterior, y al ser en este caso la versión 1.0 no daría problemas.

Vemos que nos aparecen primeramente un número elevado de bugs y vulnerabilidades.

Bugs:

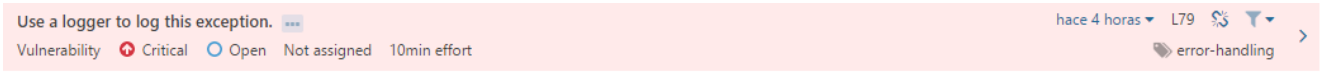
Este bug es producido al comparar un float con un double. Esto es un defecto al no tener el mismo número de decimales un tipo que el otro. Por tanto, procedemos a realizar la comparación con la librería matematica, que nos permitira limitar a un numero determinado el número de decimales.



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.

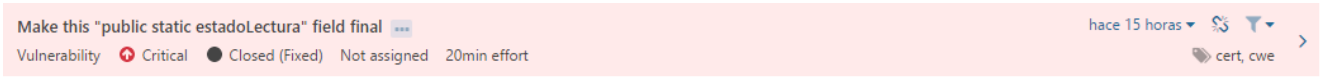
Este error se produce al no cerrar adecuadamente la lectura de los datos. Lo solucionariamos retornando el InputStream fuera del try/catch, justo antes de finalizar el método.

Vulnerabilidades:



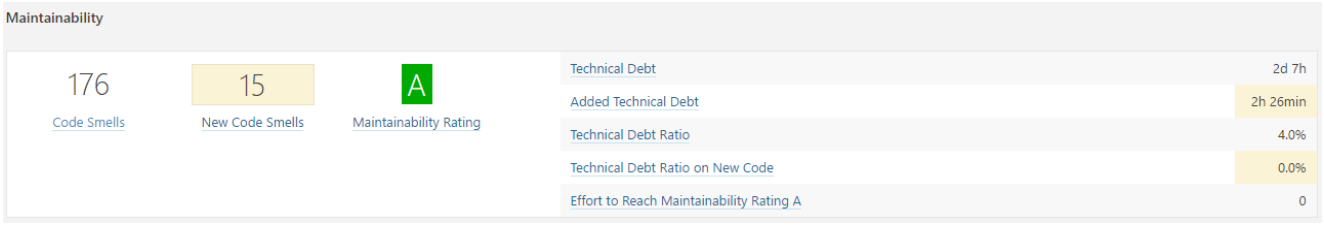
Este error se produciria al no hacer un log de la excepción. Para solucionarlo unicamente usaremos el comando log en el catch de esa excepcion para logear el error.

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:

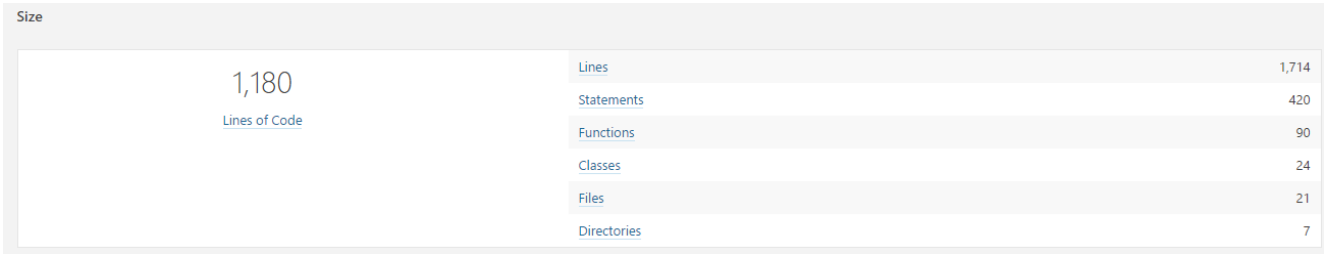


La calificacion obtenida respecto a la mantenibilidad del código es A, por lo tanto buena. La deuda técnica es de 2d7h, con un ratio del 4%.

Duplicaciones:

El número de dupliaciones es del 6%, lo que es un porcentaje relativamente elevado viendo el tamaño de nuestra aplicación.

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 ningun fichero.

Complejidad

Ilustración 3: Complejidad general

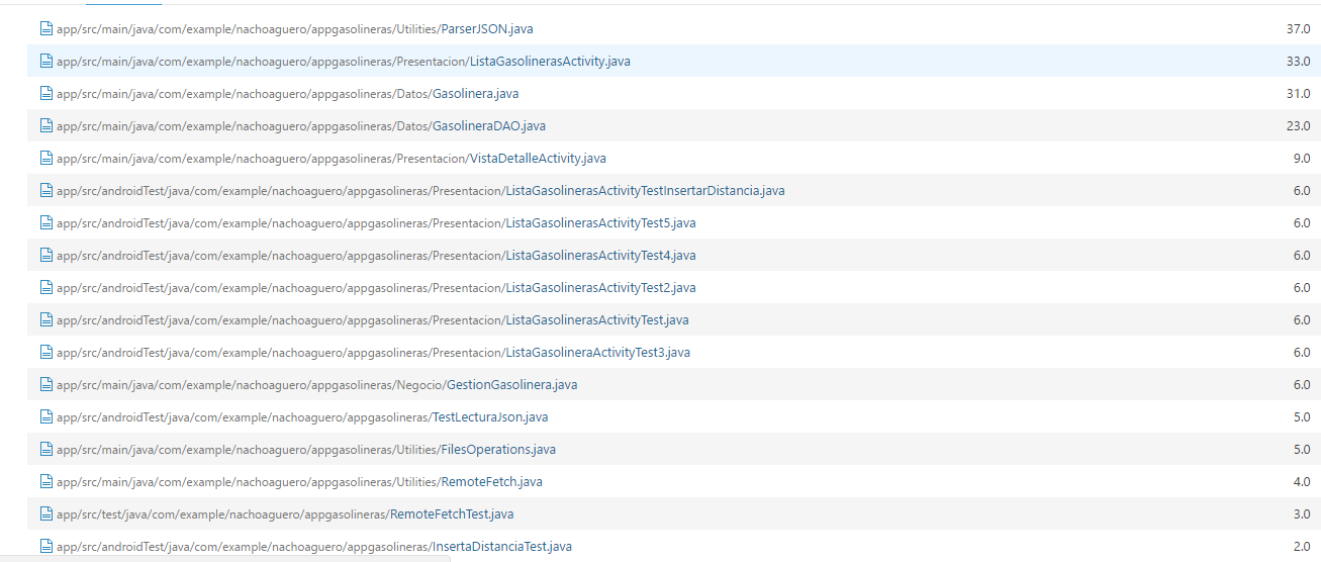
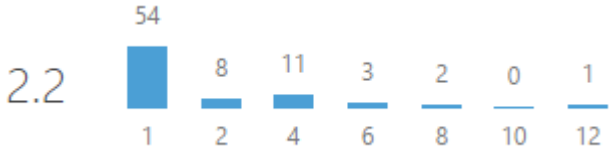


Ilustración 4: Complejidad por fichero

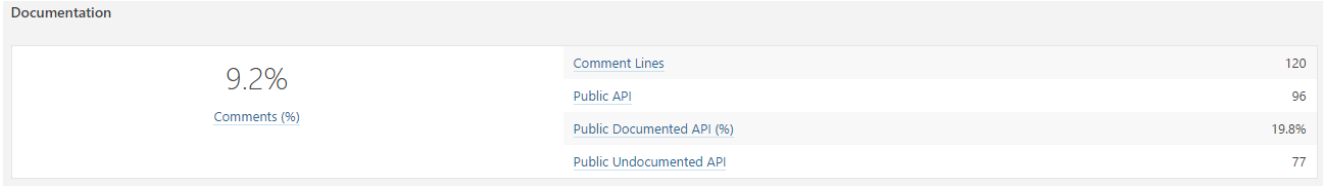
La complejidad general del proyecto vemos que es buena por lo general, aunque si entramos a ver la complejidad más en detalle, vemos que hay cuatro ficheros que tienen una complejidad excesiva, por lo que habria que modificarlos e intentar modularizarlos para repartir esta complejidad entre varios ficheros.



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 esta documentado al 9,2%, por lo que habría que mejorar y documentar más partes del codigo para intentar que este documentado al menos más de la mitad de la mitad.

PLAN DE MEJORA

Para mejorar el código de nuestro proyecto, se procedera 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 mas 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.