INFORME DE CALIDAD DE PRODUCTO

Autor	David Berbil Ruiz
Número de Sprint	Sprint002
Historia de Usuario	US-DetectarUbicación
Fecha	08/11/2016

Introducción

En este documento se lleva a cabo el desarrollo del análisis de calidad interna del producto software en desarrollo, a través de la herramienta SonarQube.

Más concretamente, este análisis será desarrollado para la historia de usuario *Detectar Ubicación,* realizando el análisis de calidad correspondiente, y realizando las correspondientes mejoras y o cambio en busca de mantener la calidad del producto.

1. Aspectos Previos al Análisis

Cabe destacar, que para este análisis se ha seguida la configuración por defecto que ofrece SonarQube, teniendo los siguientes parámetros de Qualitiy Gate y Qualitiy Profile.

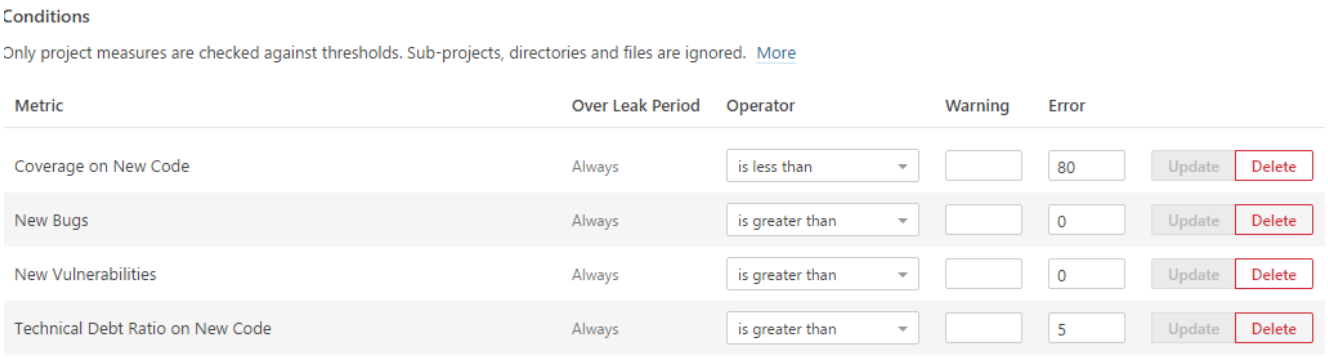


Imagen 1.1 Parámetros Quality Gate

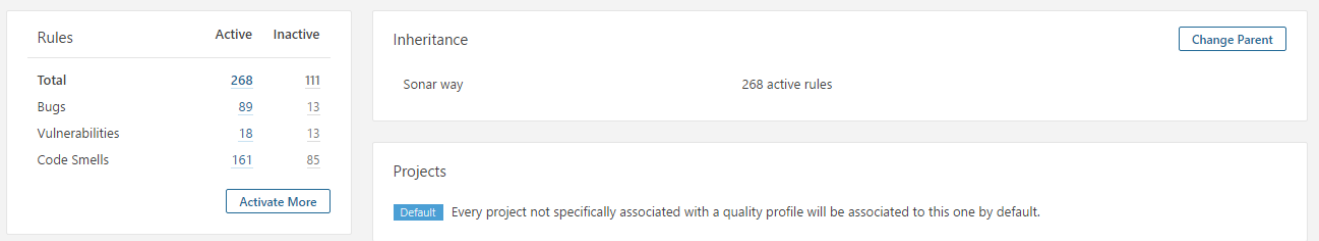


Imagen 1.2 Parámetros Quality Profile

Junto a esto, entre los aspectos previos al análisis destacar que dicho análisis contempla sólo el código de la aplicación, excluyendo también los test generados, por lo que se ha seguido la ruta ./app/src/main en el correspondiente sonar.properties.

El análisis se desarrollará de la siguiente manera:

En primere lugar, estudiaremos si el Quality Gate es positivo en base al estado actual del proyecto. Si es así, se analizarán uno a uno los parámetros de calidad; en primer lugar se estudiará la calificación SQALE, dando prioridad a aquellos aspectos que tengan una calificación D o peor.fijándonos primero en la calificación SQALE.

En relación a esto, se evaluará la criticidad de los bugs y vulnerabilidades resultantes, prestando mayor atención a aquellos que presenten una calificación “Blocker”, “Critical” y “Mayor” principalmente.

Después procederemos con los “Code Smell”, dando el mismo orden de importancia que con los bugs según su severidad.

En caso de existir duplicaciones de código, se revisarán a continuación.

Por último, se estudiará la complejidad y el tamaño de ficheros, clases, y funciones, prestando atención a aquellas que tengan una complejidad excesiva.

2. Resultados del Análisis

Comenzando el análisis, el primer aspecto a analizar es el Quality Gate del proyecto. Este Quality Gate nos da una primera visión de la calidad de los datos en base a los parámetros de QG definidos, en este caso los que usa por defecto SonarQube.

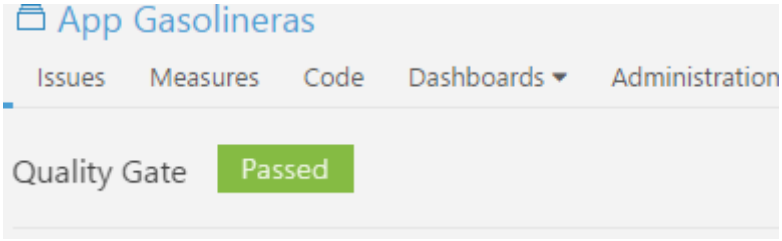


Imagen 2.1 Resultado Quality Gate

Dado que el proyecto tiene un Quality Gate "aprobado", podemos considerar que la primera visión de la calidad es buena, manteníendose en una buena línea respecto a versiones anteriores. Una vez hecho esto, deberemos analizar los distintos parámetros que marcan la calidad del proyecto.

2.1 Bugs y Vulnerabilities

En este aspecto del proyecto, obtenemos los siguientes resultados:

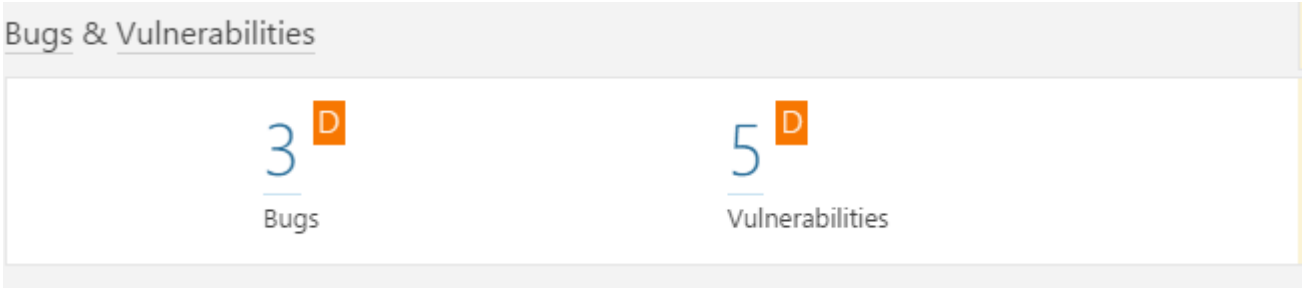


Imagen 2.1.1 Bugs y Vulnerabilities 1

En ambos apartados se obtiene una calificación "D". Esto que indica que existe al menos un aspecto crítico a corregir. Por tanto, este es uno importantes a tratar en el plan de mejora.

Si analizamos este resultado en profundidad, podemos observar que los tres bugs que contiene el proyecto podrán solucionarse en un tiempo ranozable, mientras que las vulnerabilidades, pese a ser también pocas, nos llevará el doble de tiempo en ponerles solución respecto a los bugs, pero en un tiempo razonable también.

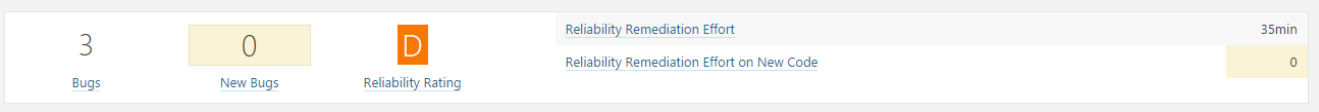


Imagen 2.1.2 Bugs

Imagen 2.1.3 Vulnerabilities

2.2 Code Smells

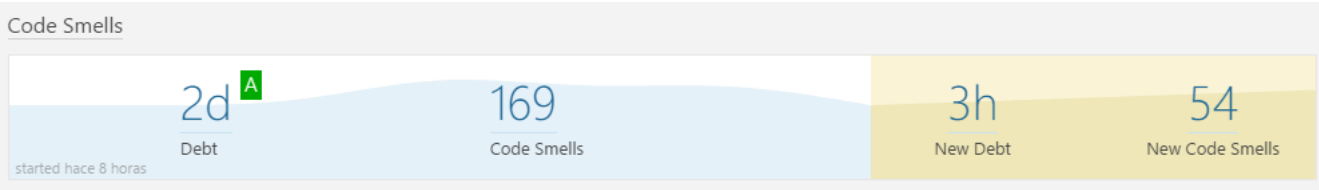


Imagen 2.2.1 Code Smells obtenido

Respecto a las medidas de mantenibilidad, se puede observar que el proyecto tiene una calificación muy positiva de “A”, lo que implica que la deuda técnica del mismo no alcanza el 0,05%. Sin embargo, esta deuda técnica alcanza los 2 días en ponerle solución (se observa que aumenta en 3 horas respecto a la versión anterior a analizar), por lo que conviene prestarle atención, ya que en su mayoría son “errores” provocados por falta de documentación o no seguir estándares de programación, de tal manera que sin ser una prioridad, se pueden seguir medidas de documentación para tener un código más claro, limpio, y que sea más fácil de seguir y comprender.

2.3 Duplicación

Siguiendo con esta tónica positiva, podemos observar que el proyecto, al igual que en versiones anteriores, no contiene duplicación en el código.

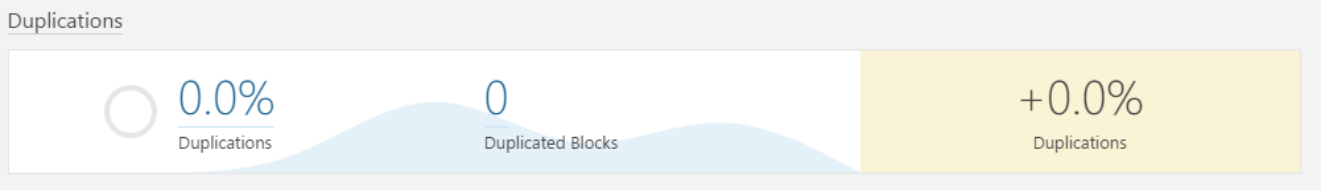


Imagen 2.3.1 Estadísticas de Duplicación

2.4 Tamaño

Respecto al tamaño que maneja el proyecto, obtenemos los siguientes resultados:

Imagen 2.4.1 Estadísticas de Tamaño

Se puede observar que se tienen 758 líneas de código, rondando el doble de líneas respecto a la versión anterior. Además, cabe destacar que este código se encuentra repartido en 13 clases albergadas en 10 ficheros, por lo que no se están cargando excesivamente estos ficheros.

2.5 Complejidad

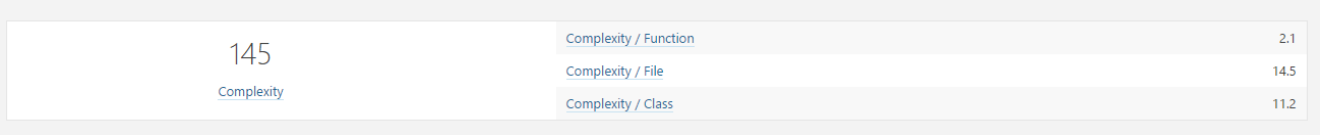
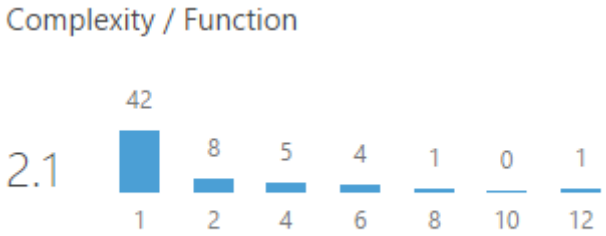


Imagen 2.5.1 Estadísticas Complejidad

Respecto a la complejidad del proyecto, se observa que sin ser datos preocupantes, ha aumentado respecto a versiones anteriores. En caso de las funciones siguen siendo datos similares y aceptables, pero en caso de las clases y ficheros ha aumentado considerablemente, sin ser cifras preocupantes teniendo en cuenta los máximos valores posibles de complejidad.

En caso de las funciones, analizando los datos más en profundidad, podemos ver que la línea de complejidad sigue prácticamente desde su inicio hasta el final, una trayectoria descendente, lo que implica que la mayoría de las funciones tienen una complejidad mínima, y a medida que va aumentando esta complejidad, las funciones resultantes son menores.



Como se mencionaba anteriormente, se observa que la complejidad en cuanto a los ficheros es algo más irregular que respecto a las funciones. Si bien es cierto que por lo general también se sigue esta línea descendente de complejidad, existen 3 ficheros cómo son P*arserJSon.java, Gasolinera y ListaGasolineraActivity* tienen una complejidad que, sin ser preocupante teniendo en cuenta en valor máximo de complejidad, rompen la dinámica general de esta complejidad/fichero. Respecto a los dos primeros ficheros, dado que son pilares fundamentales en la lógica de la aplicación, resulta díficil modificarlos para reducir su complejidad, pero en el caso de la activity, se puede estudiar el intentar descomponer su funcionalidad en distintos ficheros, buscando así reducir dicha complejidad.

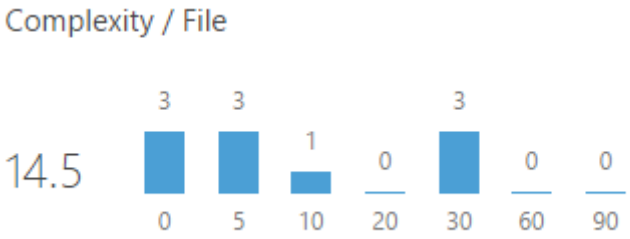


Imagen 2.5.3 Complejidad/Fichero 1

2.6 Documentación

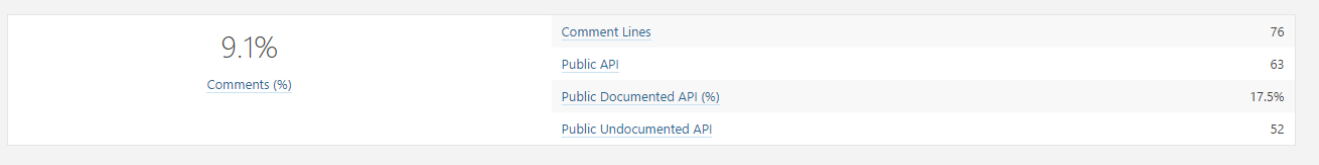


Imagen 2.6.1 Estadísticas Documentación

Un aspecto recurrente en este documento ha sido la falta de documentación del proyecto. Como se puede observar, se tienen unos porcentajes respecto a la documentación del proyecto muy bajos, lo que dificulta el trabajar con el código del mismo y su facilidad a la hora de enternderlo y ser accesible para quién intente comprender lo que sucede en él. Por tanto, sin ser un aspecto prioritario, la documentación, en base a estos datos y que afecta a otros parámetros de calidad, deberá tenerse en cuenta a la hora de querer realizar mejoras.

3. Plan de Mejora

En base a los resultados obtenidos en el análisis, en este punto se propone un plan a seguir de cara a mejorar la calidad del software.

En primer lugar, la máxima prioridad será, en la medida de lo posible, corregir los bugs y vulnerabilidades existentes en el proyecto, en vistas a mejorar la calificación SQALE.

Nos encontramos con los siguientes bugs:

En la clase Gasolinera, un fallo crítico que se arrastra desde la versión anterior, un bug prioritario en base a su estado crítico.

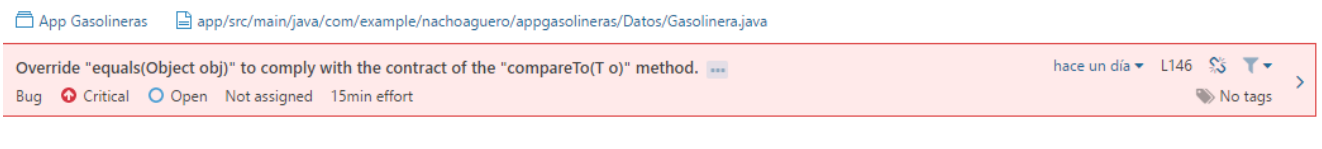


Imagen 3.1 Bug a corregir 1

En la clase FileOperations, se encuentran dos bugs en relación al tratamiento de excepciones, de estado *Major,* que se corregirán fácilmente ya que simplemente con el hecho de registrar estas exceptiones en un Log para llevar un registro de las mismas será suficiente.

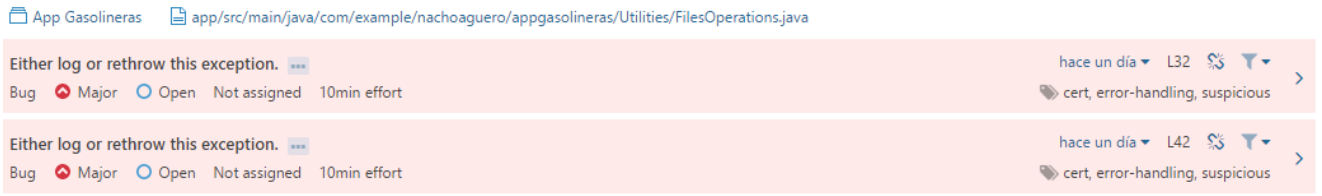


Imagen 3.2 Bugs a corregir 2 1

Respecto a las vulnerabilidades, encontramos las siguientes:

En el fichero GasolineraDAO.java, dos vulnerabilidades respecto al tratamiento de exepciones, que se solucionarán lanzando correctamente las excepciones y registrándolas en un log.

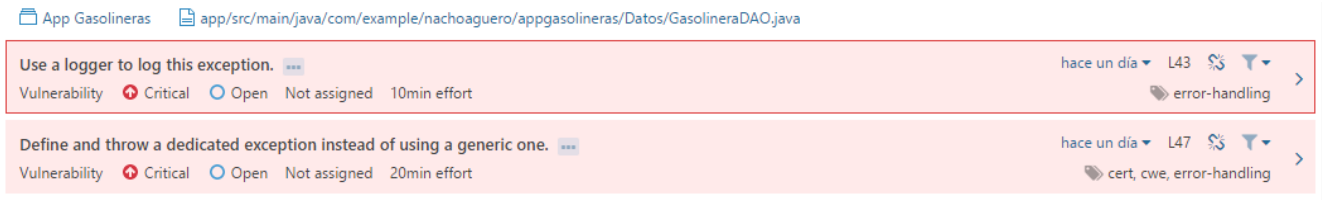


Imagen 3.3 Vulnerabilidades a corregir 1

Siguiendo en esta línea, en el fichero FileOperations.java, se encuentran dos nuevas vulnerabilidades que se solucionarán registrando las excepciones que se tratan en la clase en un Log.

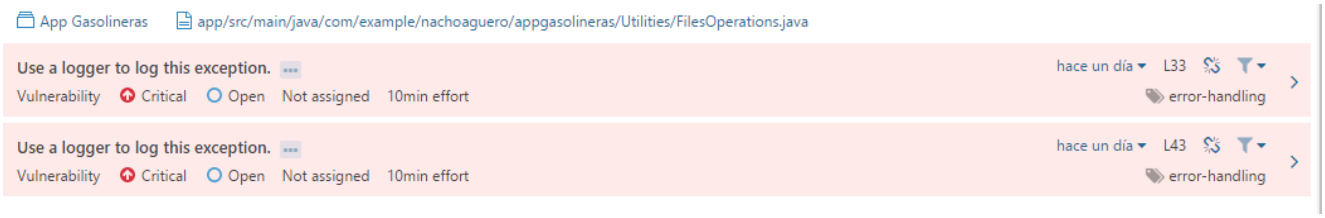


Imagen 3.4 Vulnerabilidades a corregir 2

Por último, una vulnerabilidad respecto al tratamiento incorrecto de una variable; esta incidencia se estudiará y se corregirá intentando no dificultar el comportamiento actual de la aplicación.

Imagen 3.5 Vulnerabilidades a corregir 3

El siguiente punto de tratamiento será respecto a la complejidad de ciertos ficheros del proyecto. Se intentará reducir, en la medida de lo posible, la complejidad de aquellos ficheros que conlleven una mayor complejidad, en base a desacoplar estos ficheros en varios.

Tras esto, se buscará reducir la deuda técnica, trabajando principalmente en los ficheros que más deuda técnica aporten, como es en este caso el fichero PaserJSon y ListaGasolineraActivity.java, que acarrean un gran número de horas en deuda técnica.

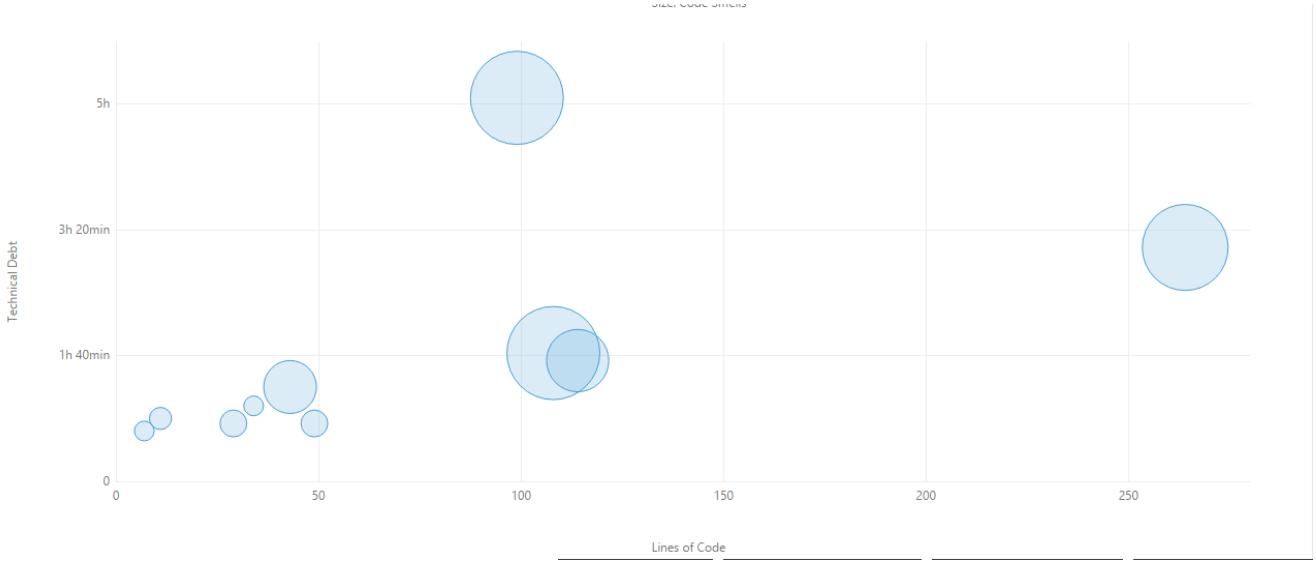


Imagen 3.6 Deuda técnica/fichero

Finalmente, se tratará de generar comentarios y documentación a lo largo de todo el proyecto para tener un código más fluido y explicado.

4. Conclusión

En este documento se ha desarrollado el proceso a seguir durante el análisis de calidad de la aplicación en su estado correspondiente a la historia de usuario *DetectarUbicación*. Para ello, se ha explicado el plan a seguir para su análisis, la configuración de la herramienta SonarQube de cara a este análisis, para comentar posteriormente los resultados obtenidos y el plan de mejora a seguir para mejorar la calidad del proyecto.