INFORME DE CALIDAD DE PRODUCTO

Autor	Orlando Britto Herreros
Número de Sprint	Sprint 2
Historia de Usuario	US-169388 Actualizar listado gasolineras
Fecha	08/11/2016

Se realiza un análisis estático de código con la herramienta *SonarQube.* El código analizado corresponde con las implementaciones realizadas en el *primer sprint* y a la historia de usuario *US-169388 Actualizar listado gasolineras,* correspondiente al *segundo sprint*.
NOTA: Las comparaciones de SonarQube con versiones anteriores no son válidas, ya que solo contemplan un par de errores sencillos de la propia historia de usuario que fueron solucionados y no la evolución de la calidad del código respecto al informe anterior.

1. Resultados del análisis

La ejecución del análisis se ha llevado a cabo con la configuración por defecto de *SonarQube*. Dentro de esta configuración por defecto se está aplicando la *QualityProfile Sonar way*, la cual consta de 268 reglas de Java. En cuanto al *QualityGate* utilizado, se ejecuta *SonarQube Way,* por lo que se aplican las métricas *Coverage on New Code, New Bugs, New Vulnerabilities y Technical Debt Ratio on New Code.*

La ejecución del análisis nos deja una serie de métricas que nos indican el estado en el que se encuentra el proyecto. Como se puede observar en la *Figura 1*, tenemos un total de 152 evidencias sin resolver. Según la estimación en esfuerzo, debemos invertir unos 50 minutos para arreglar los bugs, unos 30 minutos para arreglar vulnerabilidades y unos 2 días para arreglar los code smells. Podemos decir que el estado del proyecto no es del todo malo, ya que con poco más de 1 hora de trabajo podríamos arreglar los bugs y vulnerabilidades.

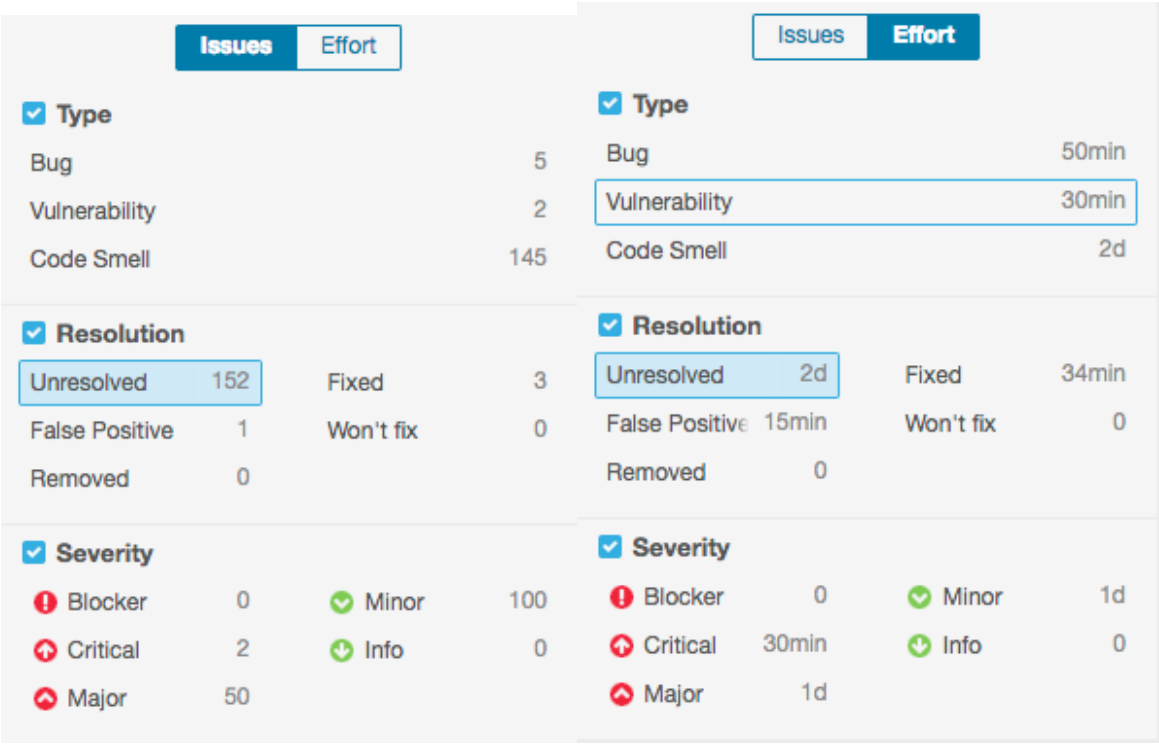

Figura 1. Número de vulnerabilidades y esfuerzo asociado.

Nos encontramos con que no tenemos errores bloqueantes, pero si críticos, mayores y menores. En la Figura 2 se muestran ejemplos de cada uno de estos tipos de errores.

Figura 2. Errores críticos, mayores y menores.

Estos ejemplos nos indican una serie de correcciones a aplicar en el código, como son utilizar el log para mostrar el diálogo de error asociado a una excepción o renombrar una variable porque está en mayúsculas sin ser una constante.

También obtenemos errores si filtramos las issues por tipo. Si filtramos para obtener las relacionadas con convenciones de código, nos encontramos con una serie de errores menores a ser resueltos.

Figura 3. Ejemplo de issues encontradas tras filtrar por el tag “convention”

Todos estos errores son menores, pero siempre es recomendable seguir las convenciones para promover la fácil reutilización del código y ayudar a terceras personas ajenas al desarrollo a entenderlo.

Si buscamos incidencias según la regla que las detecta, por ejemplo, la regla “architectural constraints” que salvaguarda los accesos directos de presentación a persistencia en arquitecturas de capas no encontramos ningún error. En un primer lugar esto se debe a que la regla no está activa en la configuración por defecto de Sonar. Procedemos a activar esta regla en nuestro QualityProfile y observamos que tampoco nos detecta ningún issue, por lo que estamos respetando las restricciones en la arquitectura de 3 capas.

Podemos también centrarnos en detectar incidencias centralizadas en un único fichero. Si aplicamos esto al fichero *CancelerTask.java* como se aprecia en la *Figura 4*, nos encontramos con que este solo posee 3 errores: 2 mayores y uno menor.

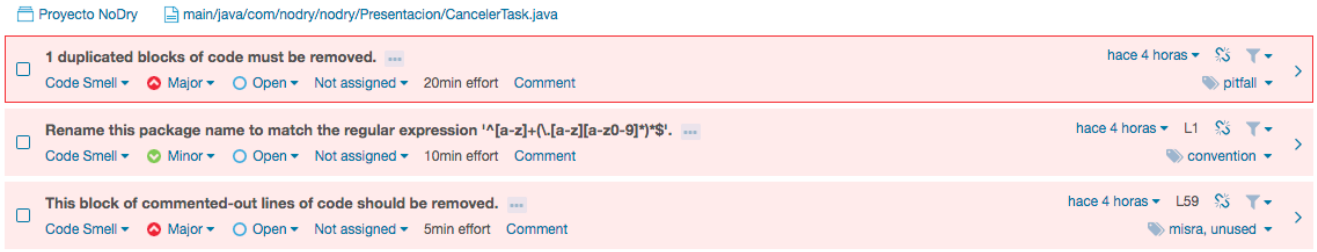

Figura 4. Issues encontradas en el fichero CancelerTask.java

Pasamos ahora a analizar las medidas de confiabilidad, seguridad y mantenibilidad del proyecto.

En un programa software, se puede definir la confiabilidad como la capacidad del programa para realizar sus funciones y operaciones correctamente sin experimentar errores. Si analizamos la confiabilidad del proyecto, como se puede observar en la *Figura 5*, encontramos un total de 6 bugs, los cuales introducen un total de 1 hora y 15 minutos de deuda técnica en el sistema. Estos errores de confiabilidad nos hacen descender la clasificación SQALE de la aplicación hasta una D, ya que entre estos 6 errores se encuentra 1 crítico. El archivo que contiene un mayor número de errores de confiabilidad es *Utils.java* con un total de 2 bugs como apreciamos en la *Figura 6*.

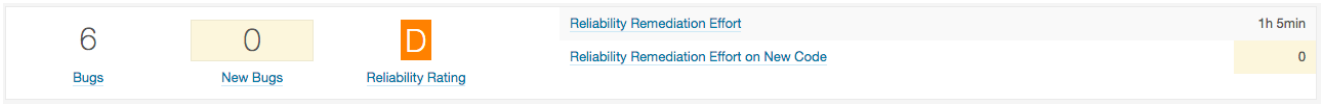

Figura 5. Resultado del análisis de confiabilidad del proyecto.

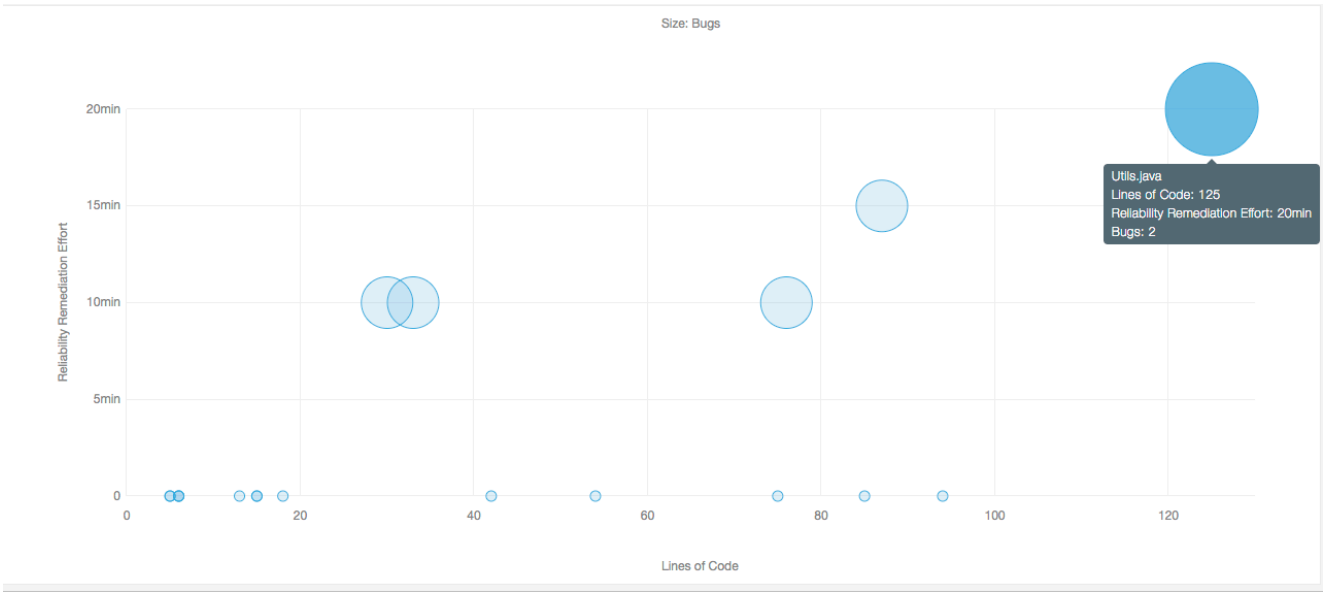

Figura 6. Bubble chart del análisis de confiabilidad del proyecto.

Si comprobamos las medidas de seguridad de la aplicación, nos encontramos con que tan solo tenemos 2 vulnerabilidades (Figura 7). A pesar de esto, obtenemos una clasificación SQALE de D, ya que una de estas dos incidencias es crítica. Estos 2 fallos de seguridad aumentan la deuda técnica de la aplicación en 30 minutos, siendo el archivo *DataFetch.java* el que contiene la vulnerabilidad crítica y que introduce mayor deuda técnica (Figura 8).

Figura 7. Resultado del análisis de seguridad del proyecto.

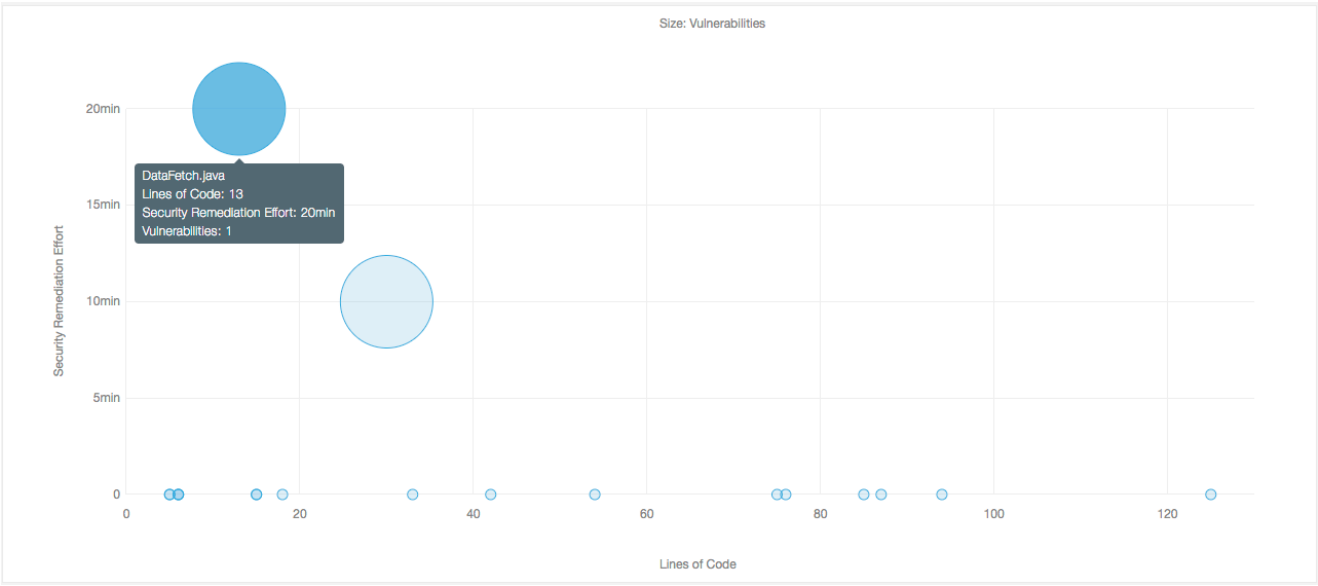

Figura 8. Bubble chart del análisis de seguridad del proyecto.

En cuanto a la mantenibilidad de la aplicación, obtenemos un buen resultado con una calificación SQALE de A. Detectamos 145 Code Smells que introducen una deuda técnica de unos 2 días, pero el radio de deuda técnica no supera el 5%, justificando así nuestra buena puntuación. Al recibir una calificación SQALE tan alta no es necesario invertir ni un minuto de trabajo para aumentar esta, pero si deberíamos trabajar durante 2 días para dejarlo perfecto (Figura 9). En el bubble chart asociado a esta medida observamos que la clase con mayor code smells y que por tanto incluye mayor deuda técnica es *Utils.java* (Figura 10).

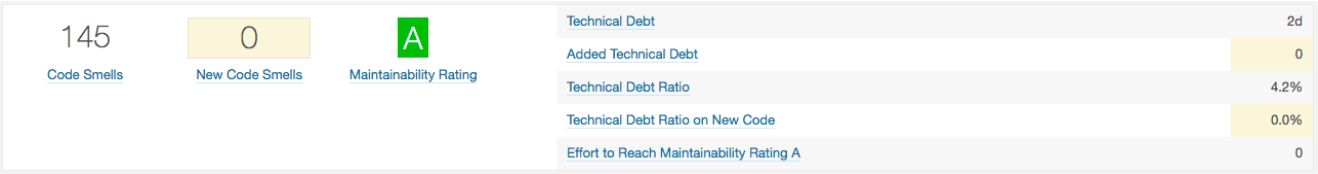

Figura 9. Resultado del análisis de mantenibilidad del proyecto.

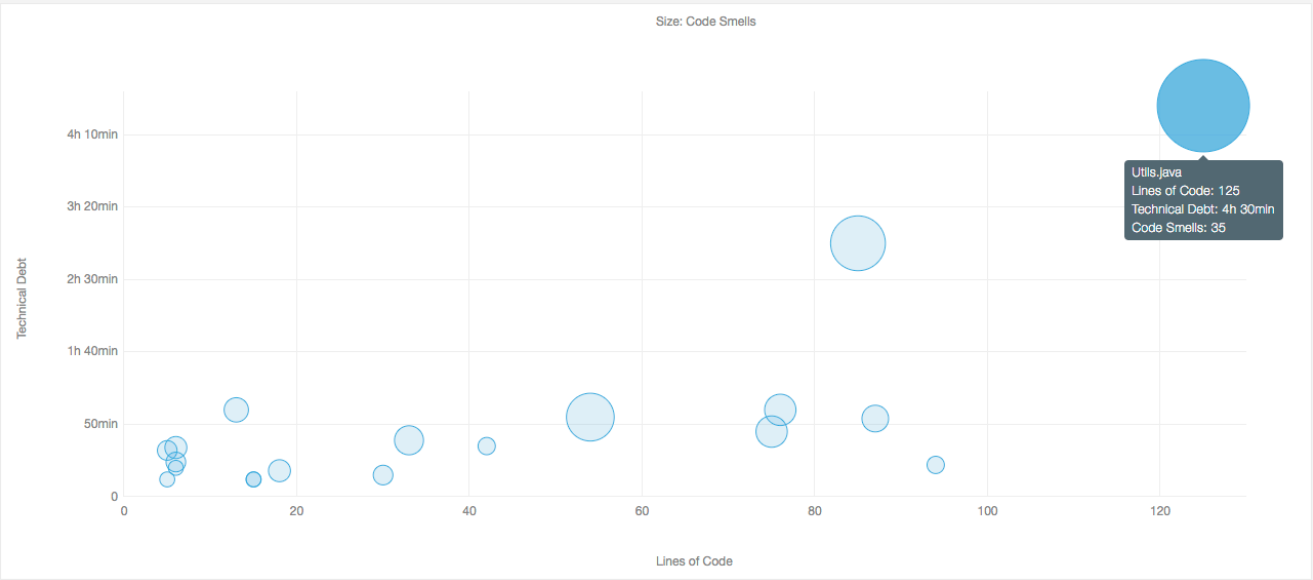

Figura 10. Bubble chart del análisis de mantenibilidad del proyecto.

Si analizamos la cantidad de código duplicado del proyecto obtenemos que tan solo un 1,7% de este son líneas duplicadas y que se encuentran en 2 ficheros distintos: *CancelerTask.java* y *GetGasolinerasTask.java* (Figura 11).

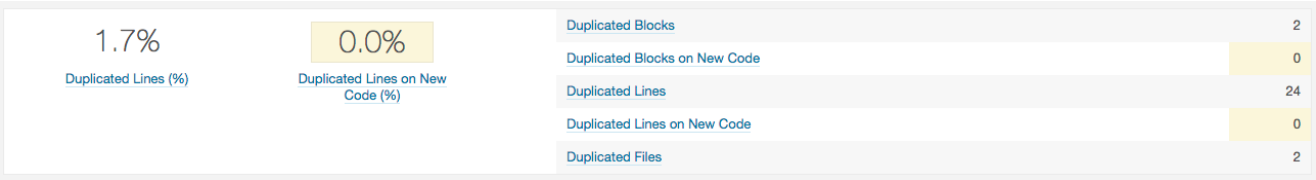

Figura 11. Análisis del código duplicado del proyecto.

Pasamos ahora a analizar la estructura del proyecto

En este punto el proyecto consta de un total de 790 líneas de código, divididas en 4 directorios que contienen los 19 ficheros. Cada fichero se corresponde con una clase Java. Utilizamos un total de 72 funciones y 349 declaraciones de variables y atributos. En total el proyecto tiene una longitud de 1407 líneas (Figura 12).

Figura 12. Tamaño del proyecto.

En cuanto a la complejidad que nos presenta la aplicación, tenemos una complejidad total en la aplicación de 112. Podemos calcular esta clasificación por función, obteniendo una media de 1.6, por fichero obteniendo una media de 5.9 o por clase obteniendo una media de 5.9 (Figura 13).
En la clasificación por función vemos como 42 funciones tienen la menor complejidad (1) y 1 función tiene la mayor (12). En la clasificación por fichero vemos como 10 ficheros tienen complejidad 0 y 1 fichero tiene una complejidad de aproximadamente 20. (Figura 14).
Estos resultados son razonables, aunque deberíamos tratar de reducir la complejidad del fichero con complejidad 20 *ParserJSON.java.*

Figura 13. Complejidad del proyecto.

Figura 14. Complejidad por función, fichero y clase.

En la pestaña de documentación observamos como el 24.5% del proyecto son comentarios y que un 85.2% de las librerías públicas están comentadas. Estos son unos buenos porcentajes y considero que se está trabajando bien en este aspecto, a pesar de que tenemos 9 elementos públicos que deben ser comentados. (Figura 15).

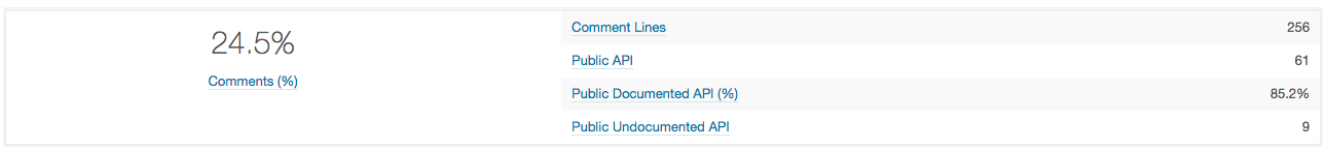

Figura 15. Análisis de los comentarios del proyecto.

Trataremos ahora de analizar la evolución del proyecto comparando los datos con el análisis realizado en la anterior historia de usuario implementada.
Lo ideal sería montar las nuevas versiones del proyecto y que este nos genere una serie de métricas automáticas, pero al realizar estos informes de manera individual y en distintos servidores no podemos hacer uso de estas. Por ello, realizaremos la comparación con los datos obtenidos por mi compañero *Manuel Álvarez Lavín* y plasmados en el anterior informe.

En el anterior análisis teníamos un total de 199 issues frente a las 152 que encontramos en esta versión del proyecto. Un buen dato que indica que estamos haciendo las cosas bien y se han corregido múltiples errores. Como consecuencia de esto la deuda técnica que era de aproximadamente 2 días y 6 horas ha sido reducida a aproximadamente 2 días.

La calificación SQALE del proyecto ha mejorado considerablemente de una E a una D en confiabilidad y se ha reducido el número de bugs de este tipo de 13 a 6.

La calificación SQALE del proyecto se mantiene en una D en seguridad a pesar de haber reducido de 7 a 2 el número de bugs. Es muy probable que estos 5 errores corregidos no fueran bloqueantes ni críticos, por lo que seguimos incumpliendo algún criterio para ascender en la escala.

Mantenemos la A de la calificación en mantenibilidad a pesar de reducir el número de code smells de 179 a 145. De nuevo debemos de estar ante correcciones no muy críticas pero que han conseguido reducir la de duda técnica de 2 días y 6 horas a 2 días.

El tamaño del proyecto ha disminuido y el porcentaje de documentación ha aumentado. Esto es muy probable que sea motivado por la no inclusión en este segundo análisis de las clases de test.

La complejidad se reduce tanto en el total del proyecto como en cada una de las categorías que nos proporciona sonar (fichero, función y clase). Al tener un mayor número de clase la complejidad ha sido distribuida en estas y se ha reducido.

2. Plan de mejora

A pesar de que actualmente obtenemos una clasificación SQALE global de D, el proyecto no muestra una mala calidad.

Mantenemos esta D en confiabilidad y seguridad tan solo por un fallo critico en cada categoría, por lo que con solucionar estos dos errores (lo cual se estima realizado en menos de 1 hora) ascenderíamos mínimo a una C en cada una de estas categorías. Obtener una calificación superior a C ya será más complicado pues tendremos que solucionar unos 50 errores mayores que nos producen el cuello de botella en la búsqueda de la B.

Si analizamos la complejidad, vemos como la clase *ParserJson.java* tiene una complejidad de aproximadamente 20. Debemos tratar de reducir esta complejidad, ya sea repartiéndolo en otras clases o reestructurando el contenido de esta.

Analizándolo desde el punto de vista de las funciones, debemos reestructurar la función que posee una complejidad de aproximadamente 12, ya que es un valor muy alto para un único método.

Disponemos de 9 APIs públicas que no han sido comentadas. Estas deben de tener una documentación asociada para facilitar el trabajo a terceros que no hayan estado implicados en su desarrollo inicial.

Sería ideal solucionar también los issues menores que se obtienen en el análisis, pero considero que al no reducir gravemente la calidad del producto es preferible invertir nuestros esfuerzos en aumentar la productividad en lugar de solucionar estos (o al menos su totalidad).