INFORME DE CALIDAD DE PRODUCTO

Autor	Juan Manuel Lomas Fernández
Número de Sprint	Sprint 3
Historia de Usuario	US-175722 Filtrar gasolineras por precio
Fecha	24/11/2016

Se realiza un análisis estático de código con la herramienta *SonarQube.* El código analizado corresponde con las implementaciones realizadas en el *tercer sprint* y a la historia de usuario *US-175722 Filtrar gasolineras por precio,* correspondiente al *tercer 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 en este último análisis que se va a realizar. Como se puede observar en la *Figura 1*, tenemos un total de 249 evidencias sin resolver. Según la estimación en esfuerzo, debemos invertir 3 días para arreglar los code smells. Podemos decir que el estado del proyecto es muy bueno, ya que no tenemos ni bugs ni vulnerabilidades.

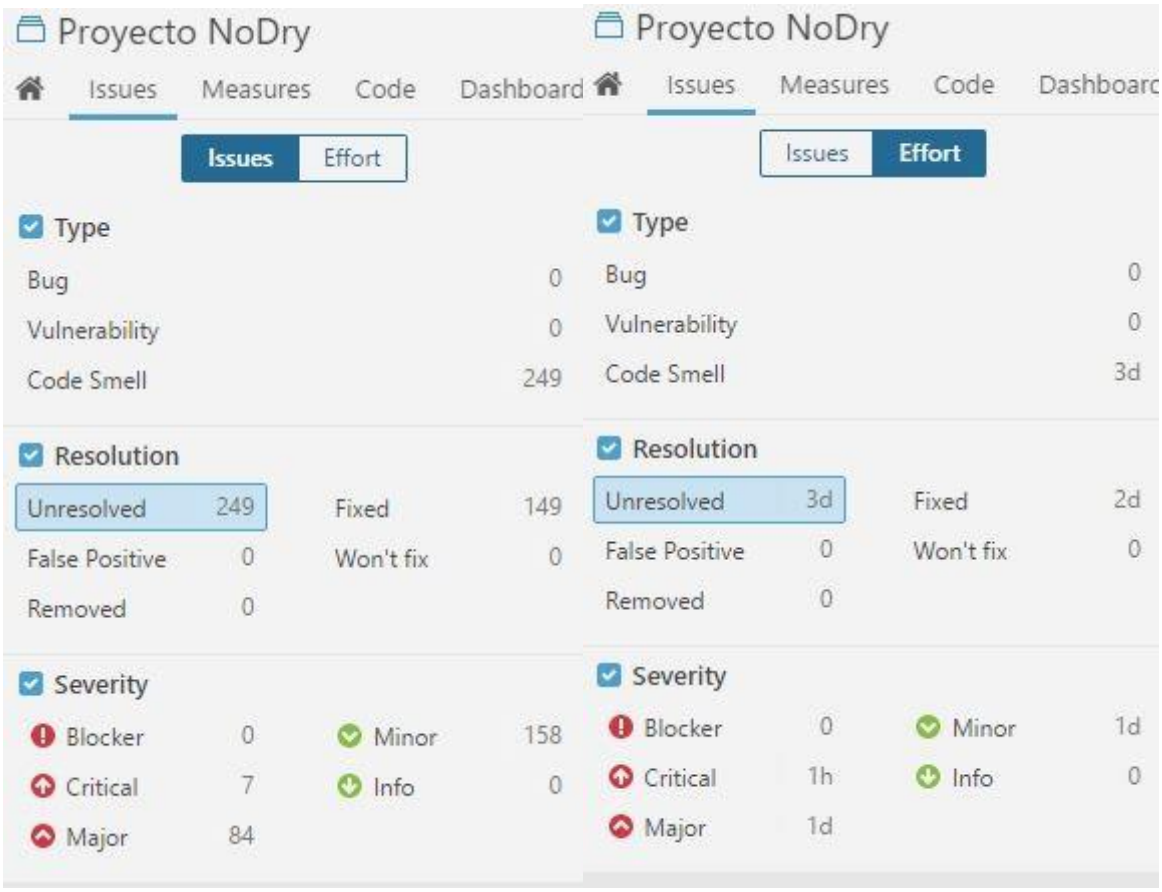

Figura 1. Número de vulnerabilidades y esfuerzo asociado.

Nos encontramos con que no tenemos errores bloqueantes, pero si 7 críticos, 84 mayores y 152 menores. En la Figura 2 se muestran algunos ejemplos de estos errores.

Figura 2. Errores.

Estos ejemplos nos indican una serie de correcciones a aplicar en el código, como renombrar expresiones regulares con los caracteres permitidos.

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.

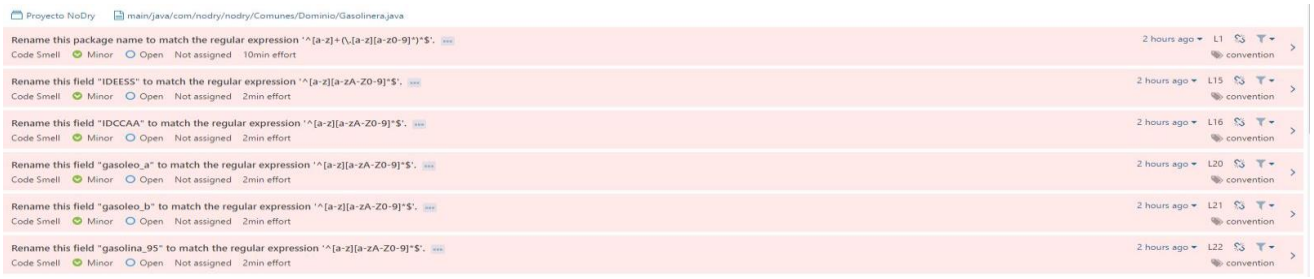

Figura 3. Ejemplo de issues encontradas tras filtrar por el tag “convention”

Todos estos errores son menores, pero siempre es recomendable seguir las convenciones para promover la fácil reutilización del código.
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 *Gasolineralocal.java y GasolineraRemoteDAO.java* como se aprecia en la *Figura 4*, nos encontramos con que poseen 2 errores menores y uno mayor, y dos menores y uno menor respectivamente.

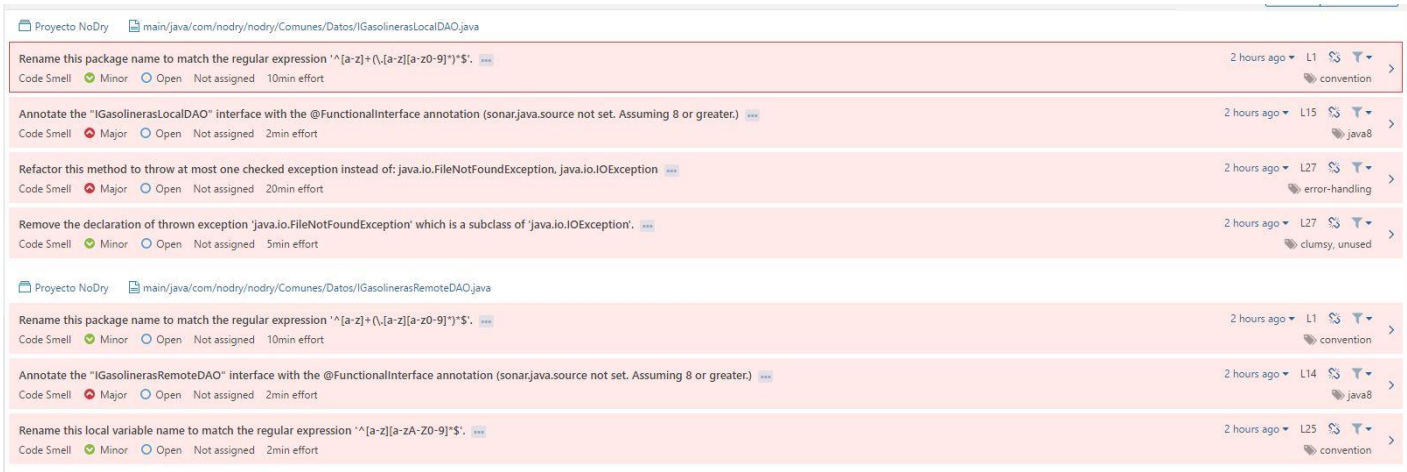

Figura 4. Issues encontradas en fichero concreto

A continuación, vamos a analizar las medidas de confiabilidad, seguridad y mantenibilidad del proyecto.

Si analizamos la confiabilidad del proyecto, como se puede observar en la *Figura 5*, encontramos un total de 0 bugs. Con esto deducimos que el proyecto está correcto ya que no tenemos errores de confiabilidad y por ello tenemos una clasificación SQALE de A.

Figura 5. Resultado del análisis de confiabilidad del proyecto.

Si comprobamos las medidas de seguridad de la aplicación, vemos que de nuevo, no tenemos ninguna vulnerabilidad y de nuevo gracias a ello tenemos una clasificación SQALE de A *(Figura 6).*

Figura 6. Resultado del análisis de seguridad del proyecto.

En cuanto a la mantenibilidad de la aplicación, obtenemos un buen resultado con una calificación SQALE de A. Como podemos ver en la *Figura 7*, tenemos 249 Code Smells, que introducen una deuda técnica de 3 días. Sin embargo, el radio de deuda técnica es del 3%, justificando así nuestra buena puntuación. Al recibir una calificación SQALE tan alta no es necesario trabajo para aumentarla, pero si deberíamos trabajar durante 3 días para dejarlo perfecto (*Figura 7*). 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 8)* como en análisis realizados con anterioridad.

Figura 7. Resultado del análisis de mantenibilidad del proyecto.

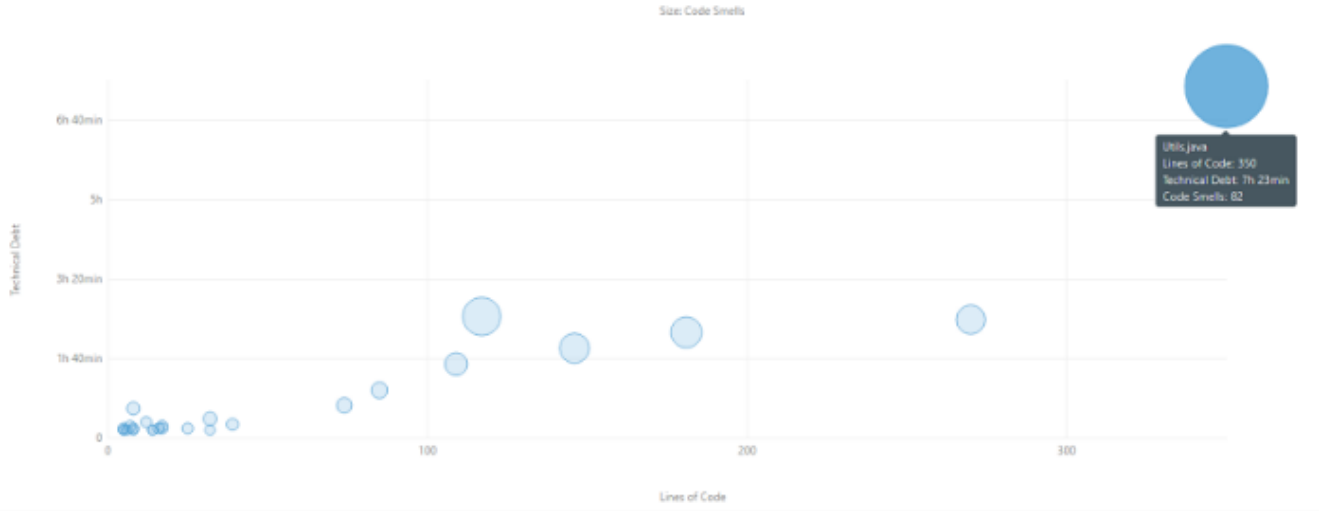

Figura 8. Bubble chart del análisis de mantenibilidad del proyecto.

Si analizamos la cantidad de código duplicado del proyecto vemos que no tenemos ni bloques, ni líneas ni ficheros duplicados, y por ello hay un 0% de líneas duplicadas. *(Figura 9)*.

Figura 9. Análisis del código duplicado del proyecto.

En cuanto a la estructura del proyecto en este punto, el proyecto consta de un total de 2673 líneas de código, divididas en 8 directorios que contienen los 26 ficheros. En total el proyecto tiene una longitud de 1602 líneas *(Figura 10).*

Figura 10. Tamaño del proyecto.

En cuanto a la complejidad que nos presenta la aplicación, tenemos una complejidad total en la aplicación de 276. Podemos calcular esta clasificación por función, obteniendo una media de 3, por fichero obteniendo una media de 10.6 o por clase obteniendo una media de 10.6 *(Figura 11).*

Figura 11. Complejidad del proyecto.

Al entrar en el análisis de documentación, vemos que el 18.9% del proyecto son comentarios. Además, el 86.2% de las librerías públicas están comentadas. Estos son unos buenos porcentajes y el proyecto está en un buen nivel en este aspecto. Sin embargo, tenemos 10 librerías publicas sin comentar *(Figura 12).*

Figura 12. Análisis de los comentarios del proyecto.

A continuación, vamos a 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 *Andrés Barrado Martín* y plasmados en el anterior informe.

La calificación SQALE del proyecto ha mejorado considerablemente de una D a una A en confiabilidad y se ha reducido el número de bugs de este tipo de 10 a 0.

La calificación SQALE del proyecto ha pasado también a una A en seguridad debido a que en esta última historia nos hemos centrado en la corrección de aquellos errores que aparecían plasmados en análisis anteriores.
También vemos que tenemos una A en la calificación en mantenibilidad, la cual ya había sido conseguida en sprints anteriores.

El tamaño del proyecto ha aumentado considerablemente debido a los cambios de arquitectura realizados en este sprint y la inclusión de nuevas funcionalidades que nos han llevado a tener 1600 líneas de código.
También vemos que una gran mejora es que no tenemos código duplicado a diferencia que en historias anteriores.
Por otro lado, el porcentaje de documentación ha aumentado

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

Después de este análisis, podemos afirmar que el proyecto está muy correcto, lo que nos lleva a tener una calificación SQALE de A.

Hemos corregido todo el código de manera que no tenemos bugs ni vulnerabilidades, por lo que no habría mejoras que realizar en este apartado.

Lo que podríamos hacer es aumentar más la documentación del código. Además, disponemos de 10 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, así como los 249 Code Smells, que introducen una deuda técnica de 3 días. Siendo muy exquisitos, podríamos decir que deberían ser resueltos los problemas mayores y menores por completo para resolver la deuda técnica.