PRÁCTICA 5B: SANDRA ALEGRÍA Y SEBASTIÁN RAMÍREZ

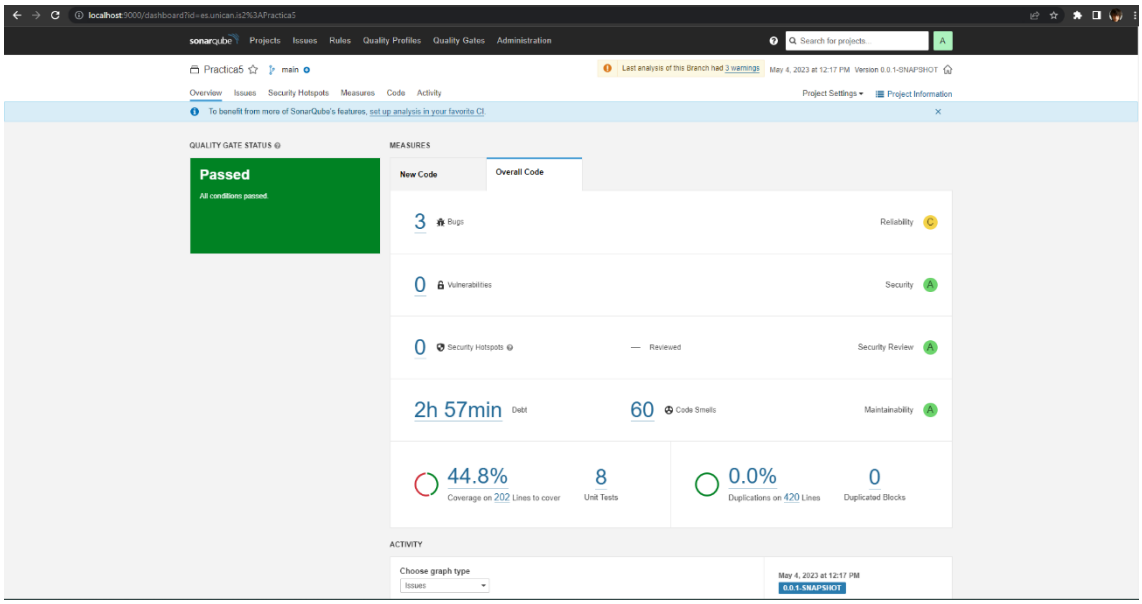
El análisis de esta práctica se ha realizado con SonarQube 9.8.

1. Análisis de la mejora obtenida en la práctica de refactorización:

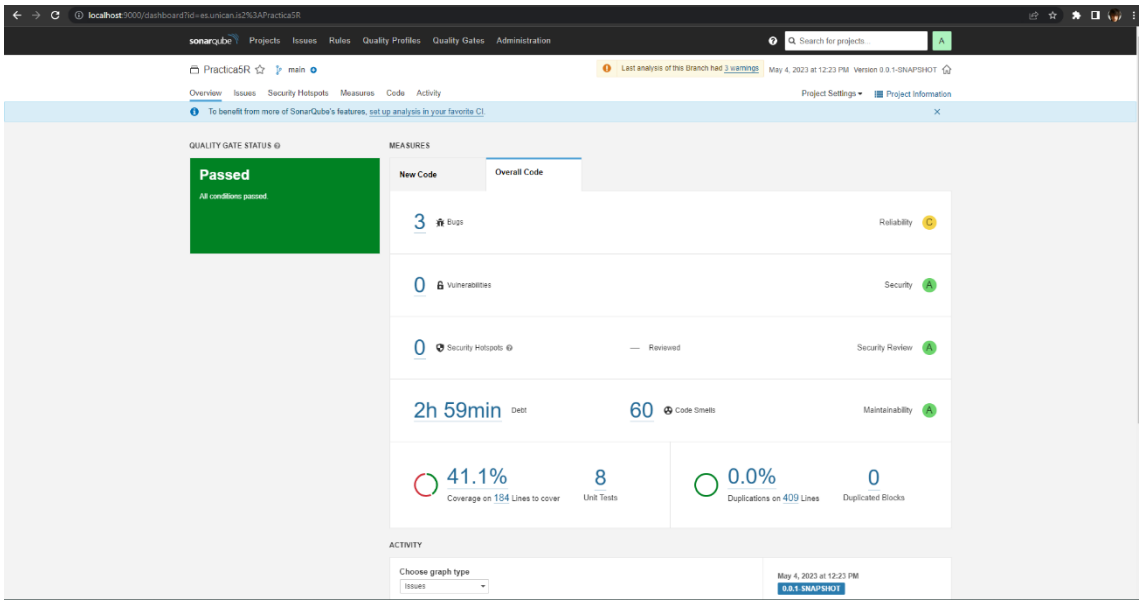
Primero vamos a realizar un análisis superficial en el que observaremos el cuadro de mando resumen de ambos proyectos y las medidas que Sonar nos ha calculado.

En primer lugar mostramos las capturas del Sonar para cada proyecto:

Proyecto sin refactorizar



Proyecto refactorizado

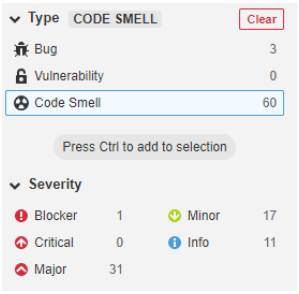


En estas capturas observamos que para el proyecto sin refactorizar (original) y el proyecto refactorizado Sonar nos calcula unas medidas bastante similares. Esto se debe a que los cambios que realizamos en la refactorización iban más enfocados en la estructuración del código, y pasamos algo más por alto las refactorizaciones centradas en el diseño, como el nombre de las

clases escrito de la manera correcta. Y Sonar realiza sus medidas centrándose más en el "estilo" y no tanto en la estructuración del código.

2. Análisis de calidad del código refactorizado:

En este apartado vamos a realizar un análisis más profundo del código refactorizado, vamos a analizar los datos que nos proporciona Sonar para identificar los problemas principales que afectan a la calidad del código.



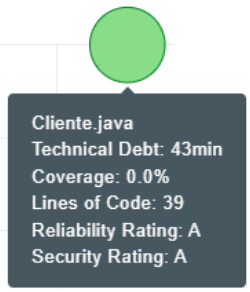
Para ello hemos empezado entrando en los Issues del proyecto y nos hemos fijado en el apartado Severity, aquí podemos observar los problemas en orden de importancia según indica el Sonar en sus métricas.

En este apartado podemos observar que tenemos un error bloqueante, cero errores críticos y 31 errores mayores. Estos errores nombrados los consideramos los más importantes.

Además hay algunos errores menores, que afectan en menor medida. Sin embargo, no podemos pasarlos por alto ya que en las siguientes observaciones realizadas podemos encontrarnos con que errores menores afectan más de lo esperado.

Para obtener más información acerca de cuales son los problemas más importantes vamos a ir al apartado Measures dentro del Sonar en el cual encontramos la gráfica siguiente:

A simple vista observamos que los colores de las clases son aceptables que hace referencia a la fiabilidad y seguridad del código. Y el tamaño hace referencia a la cantidad de líneas de código.



Sin embargo, podemos obtener información mas concreta poniendo el ratón encima de cada círculo (que hace referencia a cada clase). Y obtenemos la información que mostramos en la captura de la derecha.

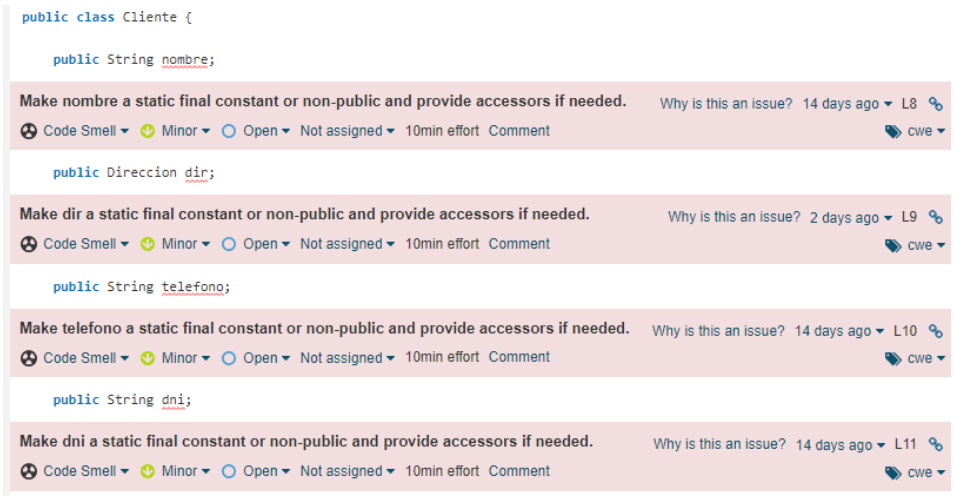
El valor que vamos a tener más en cuenta va a ser el "Technical Debt" que hacer referencia al tiempo de trabajo necesario para corregir los errores presentes dentro de la clase en cuestión.

Por ejemplo, en la captura de la derecha observamos que la clase Cliente tiene un tiempo de trabajo necesario de 43 min.

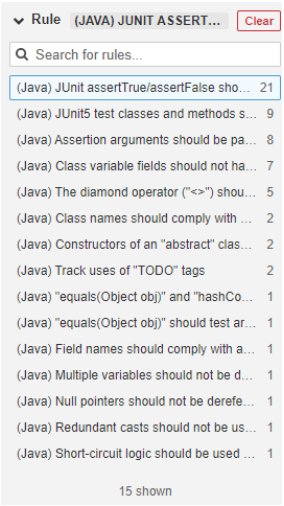
Después de observar todas las clases de la manera explicada hemos obtenido la siguiente información. Las clases con mayor tiempo de trabajo (y por tanto, con mayor cantidad de errores o errores más significativos) son: Cliente (45 min) y Dirección (30 min). El resto de las clases tienen todas un "Technical Debt" de entre siete y cero.

Cabe destacar que aunque tengan un tiempo de trabajo muy pequeño pueden tener errores muy importantes pero de rápida corrección. Un ejemplo representativo de esto es el error "Blocker" que observamos anteriormente, que se encuentra en la clase Valor que tiene un tiempo de trabajo de 5 min. Por lo que esto es únicamente orientativo y no descartamos clases únicamente por este motivo.

Dentro de la clase Cliente (45 minutos de tiempo de trabajo) hay varios errores significativos, que suman 10 min cada uno, y consideramos de los errores más importantes (a pesar de que sean errores menores, ya que afectan a la seguridad de la clase Cliente), lo mismo pasa con la clase Dirección.



En adición a lo anterior, vamos a entrar en el apartado Issues y dentro de esto vamos a Rule. Aquí observaremos que regla se incumple más.



Aquí llegamos a la conclusión de que la regla que más se incumple (un total de 21 veces) es “Junit assertTrue/asserFalse should be simplified to the corresponding dedicated assertion”. Esta regla quiere decir que tendríamos que cambiar los assertTrue/assertFalse por assertEquals. Esto podemos relacionarlo con los errores “Major” que en su gran mayoría (de 31 errores, 21 hacen referencia a esta regla).

3. <u>Plan de mejora de calidad:</u>

El Plan de mejora va a tener una extensión de aproximadamente 60 minutos de trabajo (como se indica en el enunciado).

- Correct this “&” to “&&” (Clase: Valor, Tiempo: 5 min)

Hemos elegido esta acción como la más prioritaria ya que en Issues su severidad es bloqueante y además no es muy compleja su corrección ni tiene mucho peso en cuanto a los minutos de trabajo.

- A “NullPointerException” could be thrown; “other is nullable here. (Clase: Valor, Tiempo: 10 min).

Hemos elegido esta acción ya que es un bug y además un error de severidad “Major” y consideramos que es importante repararlo.

- Use assertEquals instead (Clase: CuentaAhorroTest, Tiempo: 2 min) (x17 veces: TiempoTotal: 34 mins).

Hemos elegido esta acción ya que Sonar lo clasifica como un error de severidad “Major” y además aparece en reiteradas ocuasiones (17 para ser exactos dentro de la clase CuentaAhorroTest, como hemos indicado anteriormente). Además el tiempo individual de cada acción es bastante bajo.

- Swap these 2 arguments so they are in the correct order: expected value, actual value (Clase: CuentaAhorroTest, Tiempo: 2 min) (x8 veces : TiempoTotal : 16 mins).

Hemos elegido esta acción por un motivo similar al anterior, Sonar clasifica este error como “Major” y además consideramos que es importante reparar los test de la aplicación para poder encontrar los errores correctamente.

El Plan de mejora que hemos planteado tiene un tiempo de trabajo total de 65 minutos, que se ajusta bastante bien al tiempo pedido.

4. Resolución de incidencias de calidad:

En el plan de mejora que hemos desarrollado hay varias agrupaciones que corresponden al mismo error, por lo que la resolución es la misma en esos casos y simplifica bastante la resolución de las incidencias. Por este motivo, vamos a resolver todas las incidencias planteadas.

A continuación vamos a indicar como hemos resuelto cada incidencia:

- Correct this "&" to "&&" (Clase: Valor, Tiempo: 5 min) (Línea 43)

Este error lo hemos corregido cambiando en el return el "&" por un "&&".

- A "NullPointerException" could be thrown; "other" is nullable here. (Clase: Valor, Tiempo: 10 min) (Método: equals).

Este error lo hemos corregido introduciendo un if que compruebe que el parámetro "obj" no sea nulo.

- Use assertEquals instead (Clase: CuentaAhorroTest, Tiempo: 2 min) (x17 veces: TiempoTotal: 34 mins) (Por toda la clase)

Este error lo hemos corregido cambiando los assertTrue por assertEquals cada vez que aparecía este error. Hemos tenido en cuanta errores que aparecieron anteriores a la resolución de este para no volver a caer en los mismos fallos, por lo que hemos puesto correctamente el orden de los argumentos en los assertEquals introducidos por el cambio.

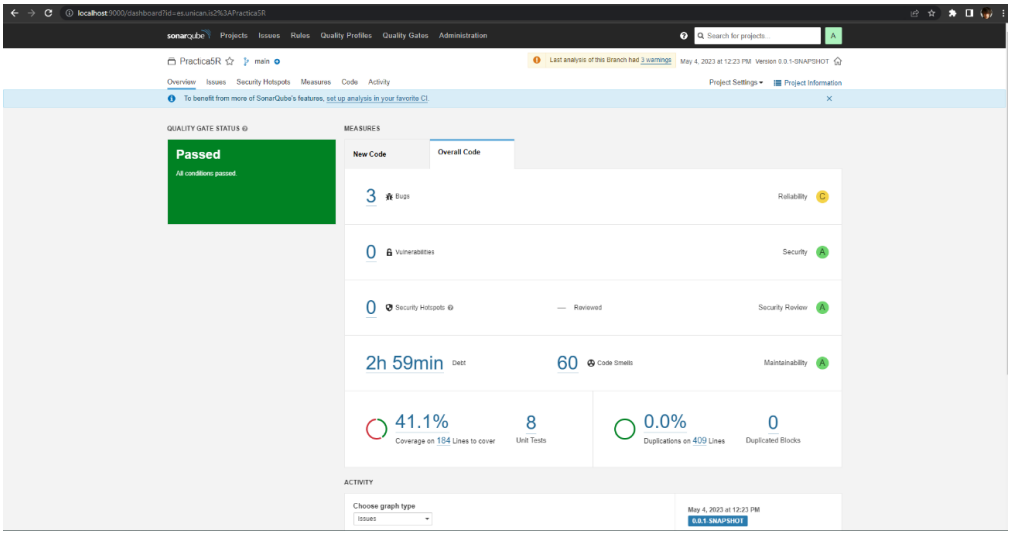
- Swap these 2 arguments so they are in the correct order: expected value, actual value (Clase: CuentaAhorroTest, Tiempo: 2 min) (x8 veces : TiempoTotal : 16 mins).

Este error lo hemos corregido cambiando el orden de los dos argumentos que le pasamos al assertEquals.

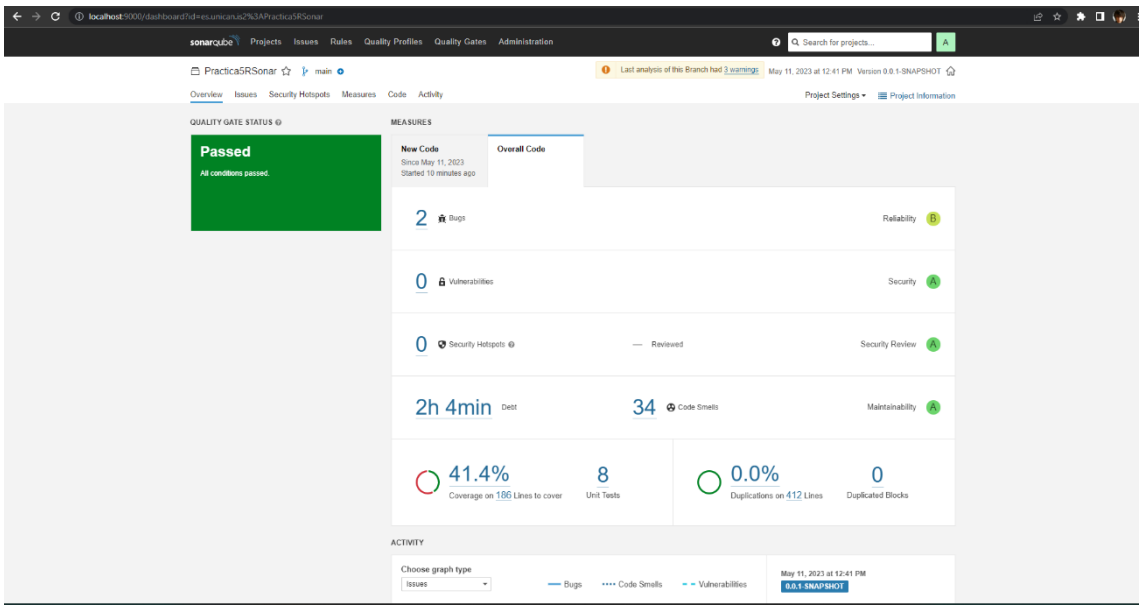
5. Análisis de la mejora de calidad obtenida:

Hemos introducido el código con la resolución de las incidencias en el Sonar y vamos a analizar los resultados.

Podemos observar una clara mejora realizando primero un análisis superficial. Recordemos como eran las métricas anteriormente para el proyecto refactorizado:



Y ahora observamos una captura de las métricas después de resolver los problemas del plan de mejora:

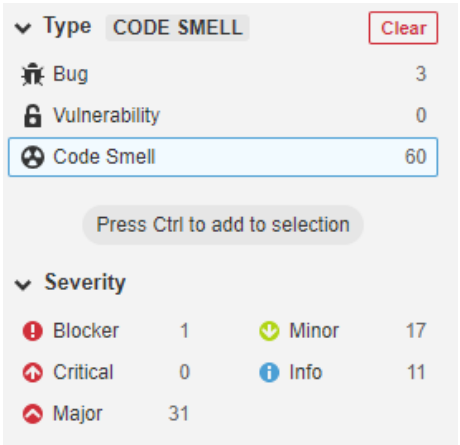


Hemos observado una clara mejora a simple vista. Lo más significativo es el número de “Code Smells” que ha bajado de 60 a 34 (casi a la mitad). Esto ha afectado positivamente a el “Debt” que también se ha visto reducido acorde al plan de mejora que hemos planteado. Cabe destacar que hemos reducido el número de “Bugs” y por ello hemos incrementado la métrica “Reliability” de C a B, que es claramente mejor.

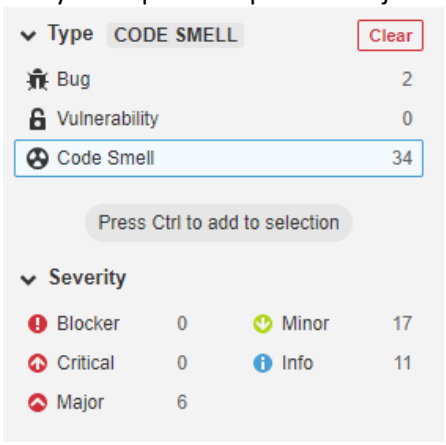
Ahora vamos a realizar un análisis un poco más profundo del nuevo proyecto (proyecto refactorizado después de aplicar el plan de mejora). Para ello vamos a profundizar más en las métricas del Sonar.

Nuevamente vamos a recordar las características del proyecto refactorizado, y lo compararemos con el nuevo.

Proyecto Refactorizado

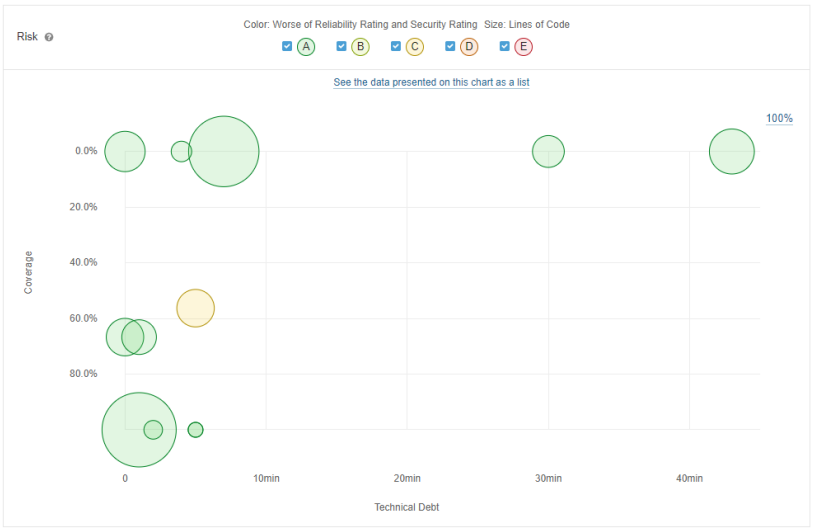


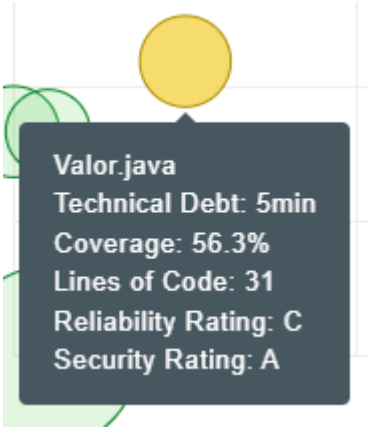
Proyecto aplicando plan de mejora



Aquí obsevamos que hemos solucionado el error “Blocker” y gran cantidad de los “Major” además de uno de los “Bugs”.

Además vamos a volver sobre la gráfica que nos proporciona el Sonar para el análisis:





Las capturas anteriores se corresponden con el proyecto refactorizado, posterior al plan de mejora. Nos fijaremos sobre todo en la clase Valor que es en la que hemos resuelto algunos de los problemas más significativos.

En las capturas siguientes vemos los mismos datos después de aplicar el plan de mejora. En estas capturas observamos que la clase Valor ha sufrido cambios favorables, como por ejemplo la métrica “Reliability Rating” que ha pasado a ser una B y no una C. Y además hemos reducido a 0 el tiempo de trabajo para esta clase.