Universidad de Sevilla

Escuela Técnica Superior de Ingeniería Informática

Grado en Ingeniería Informática – Ingeniería del Software
Proceso de Software y Gestión 2

Curso 2020 – 2021

ANÁLISIS DEL CODIGO FUENTE Y MÉTRICAS ASOCIADAS

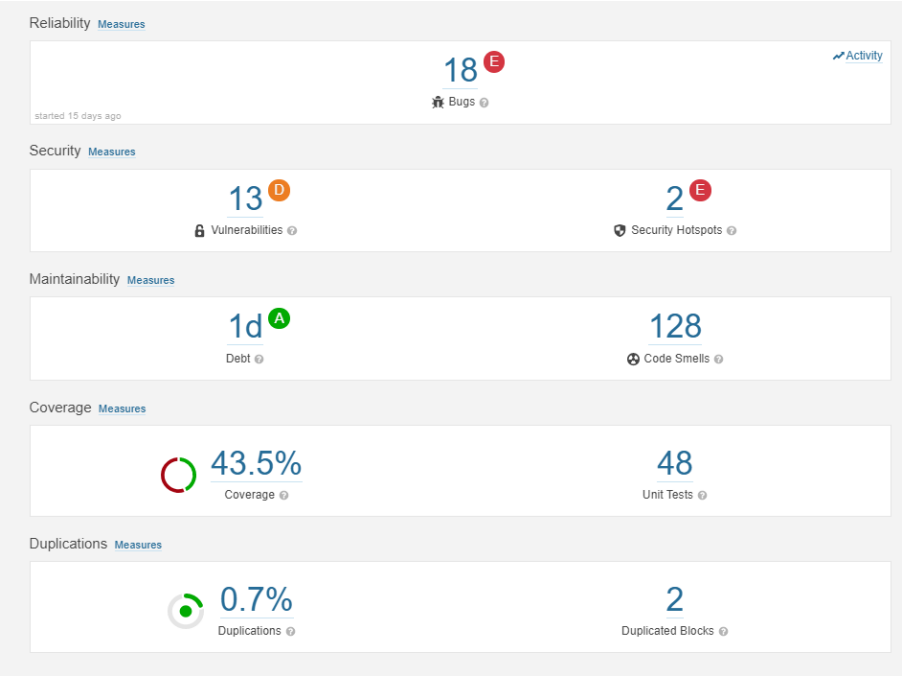
Grupo de prácticas	**G2-24**
Autores	**Rol**
Rodríguez Pérez, Francisco	Scrum Master
Colmenero Capote, Pablo	Scrum Team Member
Martin Núñez, Ángel	Scrum Team Member
Barragán Salazar, David	Scrum Team Member
Pastor Fernández, Ginés	Scrum Team Member
Müller Cejas, Carlos Guillermo	Product Owner

Contenido

Sprint 2

Resumen del análisis

Se realiza un análisis en profundidad del código fuente de la rama *master* y las métricas asociadas a este mismo gracias a la aplicación **SonarCloud**.



La primera característica que destaca de la imagen es la **fiabilidad** (probabilidad de que un sistema cumpla una determinada función bajo ciertas condiciones durante un tiempo determinado) de la aplicación. Se observa como SonarCloud lo califica con una E, resaltando problemas de seguridad con un total de 18 bugs.

La segunda característica es la **seguridad** (mide la vulnerabilidad del sistema y los puntos de acceso por los que puede existir un fallo crítico) de la aplicación. SonarCloud, en esta caso, califica como D la vulnerabilidad con 13 puntos que pueden causar problemas de seguridad, además de una calificación de E sobre los puntos de acceso que pueden ocasionar un fallo crítico (2 en este caso, siendo estos propios de la aplicación por defecto).

Respecto a la **mantenibilidad** del sistema (facilidad con la que se pueden arreglar los fallos de un sistema) SonarCloud califica la deuda técnica con una A obteniendo de manera estimada 1 día para arreglar los olores que desprende el código, 128 como se puede observar.

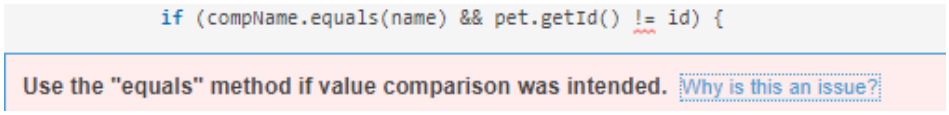
La **cobertura** de código (tests realizados para la funcionalidad de este mismo) se estima en un 43.5% con un total de 48 tests unitarios.

Para terminar, se aprecian las **duplicaciones** (código que se repite en diferentes funciones) con un total de 0.7% de código duplicado.

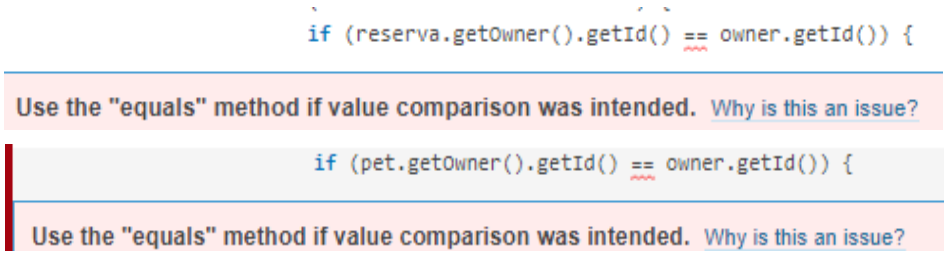
Bugs encontrados

1 - Uso de igualdad si se pretende comparar un valor

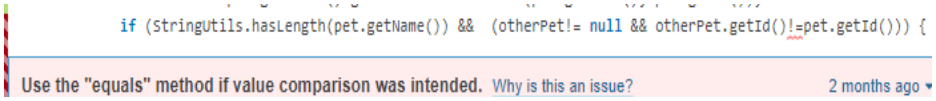
SonarCloud resalta este tipo de bug 5 veces, se produce cuando se utiliza una igualdad para comparar dos valores obtenidos a partir de un objeto. Estos valores pueden no ser correctos dado que es posible que no sean los valores reales sino las ubicaciones en memoria.



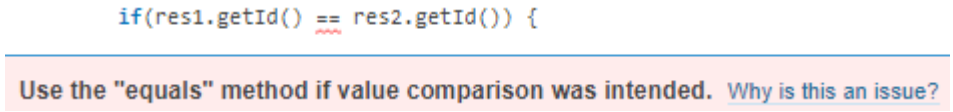
Model -> Owner



Controller -> ReservaController



Service -> PetService

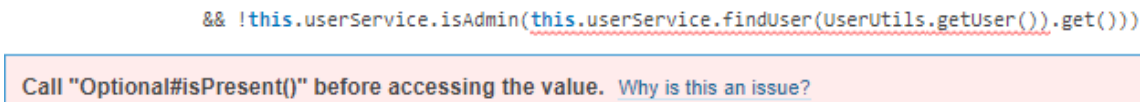


Service -> ReservaService

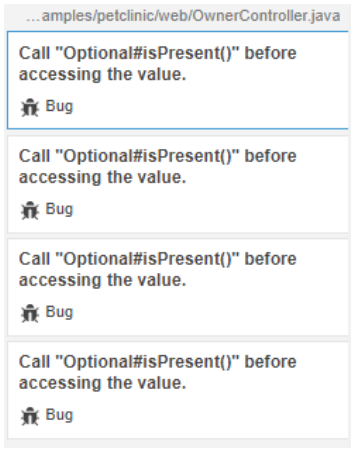
Para solucionar el bug, simplemente debemos cambiar todos los comparadores por el método equals (teniendo en cuenta que hace falta el operador "!", en caso de que queramos sustituir un "!=", delante de la expresión booleana).

2 - Obtener valor de un Optional sin saber si existe ese valor

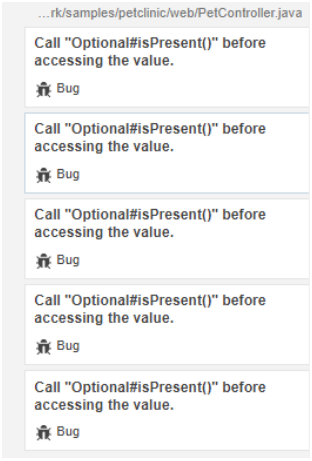
Este tipo de bug es destacado por SonarCloud cuando se intenta saber el valor de un objeto "*Optional*" sin saber si ese objeto existe (*uso de la* función *.isPresent()*). Este es el bug más repetido en todo el código (12 veces).



Ejemplo de bug con objeto Optional



Bugs de objeto Optional encontrados en OwnerController



Bugs de objeto Optional encontrados en PetContoller

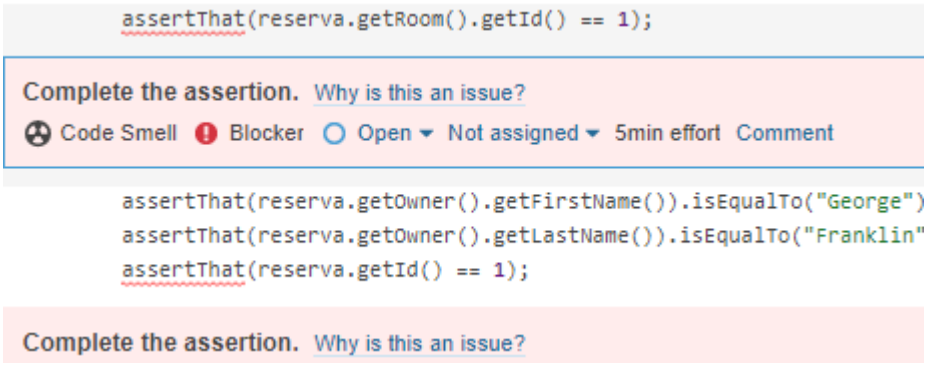
Para solucionar este bug debemos crear una variable opcional y comprobar que está presente con un "*assertTrue(**variable**.isPresent()*" y luego crear la variable que vamos a usar a partir de hacerle un "*get()*" a la variable opcional.

Olores encontrados

Olores blocker

1 - Completar el assert

Este olor se produce cuando un assert no está completo y puede que efectivamente no se verifique lo que se pretende probar.



Código de donde proviene este tipo de olor

En este caso se puede solucionar el olor cambiando el *assertThat* por un assertEquals y sustituyendo el comparador por una coma.

Olores critical

1 - Definición de una constante en vez de una cadena de texto 'x'

Este olor indica que hay una cadena de texto 'x' que se repite en el código y que puede ser reemplazada por una constante.

Este tipo de olor es muy común en el análisis que ha hecho SonarCloud de nuestra aplicación, oliendo en diversas clases.

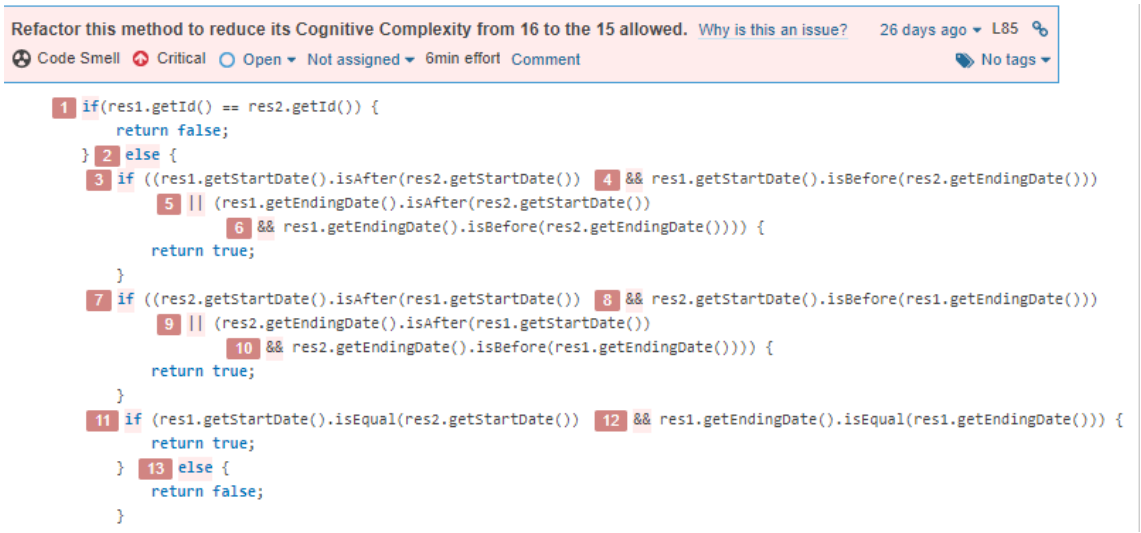
Ejemplos de este tipo de olor

En este caso puede existir un fallo a la hora de utilizar la cadena que se pasa por parámetro en las funciones (pudiendo existir un fallo y que por tanto no funcione correctamente).

Para este tipo de olores lo correcto es, tal como se dijo al principio de esta sección, reemplazar estas cadenas por constantes dado que estas pueden ser utilizadas en otras partes de código y, además, en caso de querer cambiar el contenido de esta, se puede cambiar fácilmente de una vez, caso contrario de lo que pasaría si mantenemos las cadenas repetidas, teniendo que cambiar cada cadena.

2 - La complejidad cognitiva es demasiado alta

Un olor muy común que ha detectado SonarCloud en nuestra aplicación es cuando la **complejidad cognitiva** de una función es demasiado alta y es difícil de entender para los humanos, además que puede presentar una futura dificultad a la hora de mantener el código.



Ejemplo de este tipo de olor

En este caso se entiende que SonarCloud lo clasifique como “Critical” dado que puede ocasionar diferentes problemas futuros de comprensión y mantenimiento.

Una posible solución sería comentar cada parte de código o declarar variables con nombres explicativos y utilizarlos en la función.

Olores major

1 - Renombrar campo

Es confuso tener un campo de una clase con el mismo nombre (dejando de lado las diferencias entre mayúsculas y minúsculas) que su clase adjunta, ocurriendo esto en una clase en concreto.

```
@XmlRootElement
public class Vets {

    private List<Vet> vets;
```

Rename field "vets" Why is this an issue?

Ejemplo de fallo en un campo

En este caso la solución es sencillamente renombrar el campo o variable de donde sale el olor (y por supuesto las posibles llamadas de este campo en otras clases o funciones).

2 - Bloque de líneas de código comentadas debe eliminarse

Es un olor característico cuando hay una serie de líneas de código comentadas que no son útiles y deben ser eliminadas.

```
//user.get().getAuthorities().add(authority);
```

This block of commented-out lines of code should be removed. Why is this an issue?

Ejemplo de olor de bloque de líneas de código comentadas

La solución se basa simplemente en comprobar si el código comentado es explicativo (es decir, explica el código sobre el que se comenta) o si simplemente es código que no se utiliza para nada y debe ser eliminados.

3 - Asignación de variable inútil

Un caso individual de un tipo de olor es una asignación de variable de manera inútil.

```
this.ownerService =ownerService;
```

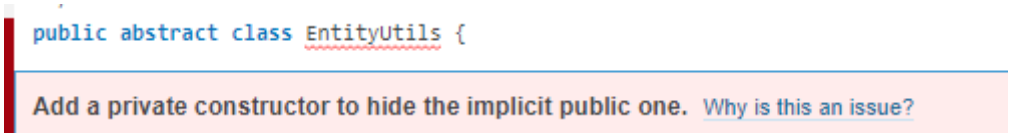
Remove this useless assignment; "ownerService" already holds the assigned value along all execution paths. Why is this an issue?

Único ejemplo de este tipo de olor

La solución pasa por comprobar si esa variable efectivamente no es utilizada en el código y si es así, borrarla.

4 - Agregue un constructor privado para ocultar el público implícito

Este olor proviene de las clases originales de **PetClinic** y se originan debido a que poseen un constructor público en la clase (en vez de uno privado).

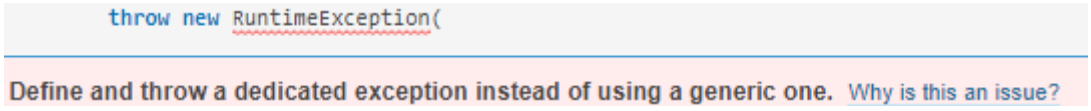


Clase que no implementa un constructor privado

Este olor se soluciona implementado un constructor privado para la clase en cuestión.

5 - Defina y lance una excepción dedicada en lugar de usar una genérica

Al igual que el olor anterior, este emana de una clase original de **PetClinic** y se produce cuando hay una excepción genérica, lo que evita que los métodos de llamada manejen excepciones verdaderas generadas por el sistema de manera diferente a los errores generados por la aplicación.

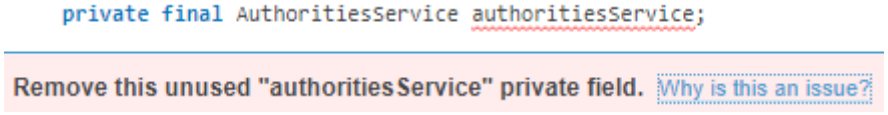


Ejemplo de excepción general

Para este olor no habría otra solución más que lanzar una excepción propia para cada caso.

6 - Campo sin uso

Este olor es parecido al olor visto en la parte 3, pero con la diferencia de que, aunque aquí si está bien hecha la declaración, no se usa la variable en ningún momento.



La solución más fácil es borrar la variable.

7 - Reemplazar System.out por Loggers

Un tipo de olor es el que sale cuando en el código cuando se utiliza un System.out por Loggers.

Al registrar un mensaje, hay varios requisitos importantes que deben cumplirse:

- El usuario debe poder recuperar fácilmente los registros.

- El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro.

- Los datos registrados deben registrarse realmente

- Los datos confidenciales solo deben registrarse de forma segura

Si un programa escribe directamente en las salidas estándar, no hay absolutamente ninguna forma de cumplir con esos requisitos. Por eso es muy recomendable definir y utilizar un registrador dedicado.

```
System.out.println("holis2");
```

Replace this use of System.out or System.err by a logger. Why is this an issue?

Ejemplo de uso de System.out

La solución más lógica es el uso de herramientas que permitan el uso de registros de mensajes. También se pueden eliminar los System.out innecesarios.

```
logger.log("My Message");
```

8 - Duplicación de bloques de código

Este olor se produce cuando hay fragmentos de código que se repiten en diferentes clases o ficheros.

2 duplicated blocks of code must be removed. Why is this an issue?

Una forma de solucionar este olor es crear una función con ese fragmento de código y llamar a esa función.

9 - Eliminar o completar trozos de código

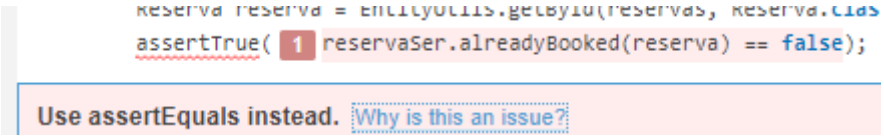
Olor proveniente de un trozo de código que no se ha completado o en su defecto no influye.

Ejemplo de trozo de código sin completar

Una solución rápida sería eliminar el código.

10 - Uso de assertEquals

Un olor que viene de un posible mal uso de assert.



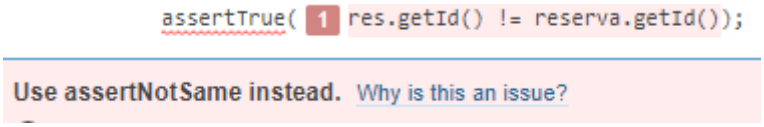
Ejemplo del mal uso de Assert

Una solución práctica sería la descrita en esta imagen. `Assert.assertEquals(a, b);`

Se compara una propiedad a con la b, en vez de realizar lo visto en el ejemplo.

11 - Uso de assertNotSame

Un olor que viene de un posible mal uso de assert.



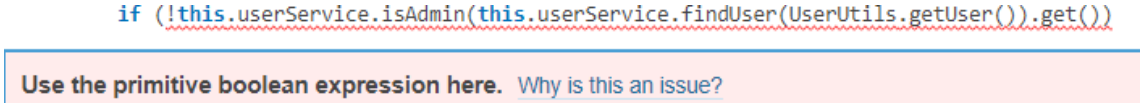
Ejemplo de mal uso de Assert

Una solución práctica sería realizar assertNotTheSame de dos valores.

Olores minor

1 - Posible mal uso a la hora de realizar un if

El primer olor de categoría “minor” ocurre en fragmentos de código en los que necesitamos saber condiciones para ciertas propiedades.

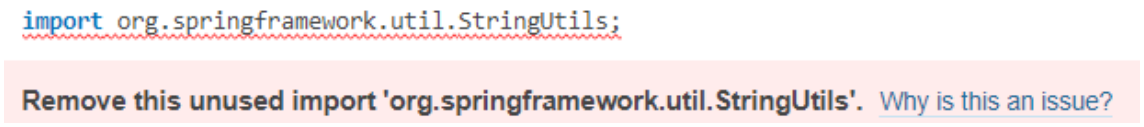


Ejemplo de if sin uso de boolean primitivo

En este aspecto a la hora de hacer la comprobación hemos querido preservar este modelo de evaluar la condición.

2 - Import sin uso

Un olor que implica un import de una clase o paquete que no se usa en ninguna funcionalidad

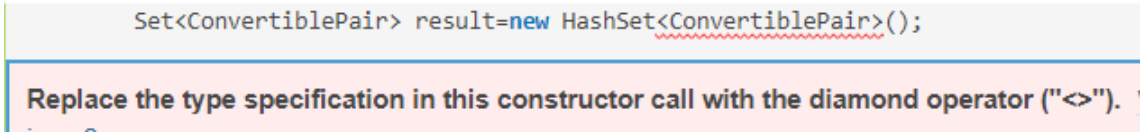


Ejemplo de import sin usar

La solución es simple, borrar los *imports* que no se utilizan con el comando *Ctrl+Shift+O*.

3 - Especificación de tipo inútil

En este caso el olor viene de un tipo declarado en un constructor.

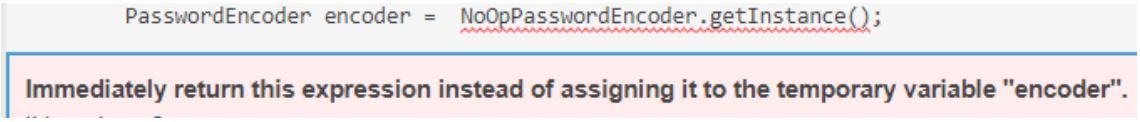


Ejemplo de tipo declarado en un constructor.

Esto ocurre debido a que Java por defecto asigna el tipo de lista o conjunto que se quiere crear, siendo inútil la especificación. Por lo tanto, el olor se corregirá al ser borrada esta especificación.

4 - Retornar directamente la expresión en lugar de asignar a una variable

Olor creado a partir de una mala asignación.



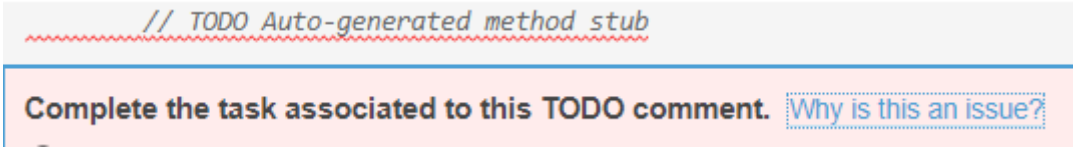
Ejemplo de una incorrecta asignación

La solución rápida es eliminar la asignación y que se retorne de manera directa la expresión que inicializaba la variable.

Olores info

1 - Completar las funciones con comentarios TODO

Un olor que ocurre cuando se genera una función con un comentario TODO.



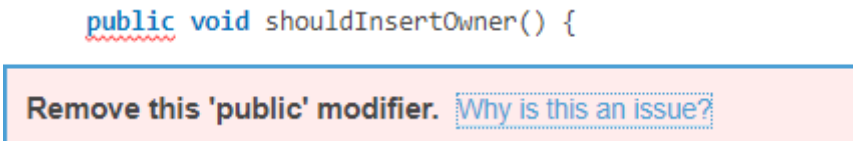
Ejemplo de comentario TODO

En nuestra aplicación hay funciones con comentarios TODO que no necesariamente implica completar esa función con código pues de por si esa función ya es eficaz por si sola.

Para corregir el olor simplemente se borraran los //TODO (no sin antes comprobar que no haya nada por hacer)

2 - Eliminar un modificador "public"

Olor que ocurre en nuestra aplicación en los tests unitarios. El fallo puede implicar el uso de estos tests en otras clases.



Ejemplo de modificador "public"

Una solución es eliminar el modificador public de los tests unitarios.

Conclusiones

Tras realizar el análisis de este Sprint, se observa una gran cantidad de bugs y malos olores. Esto se debe, en gran parte, a la falta de conocimientos que tenía el *Scrum Team* sobre esos conceptos, pues se enseñaban en la asignatura posteriormente.

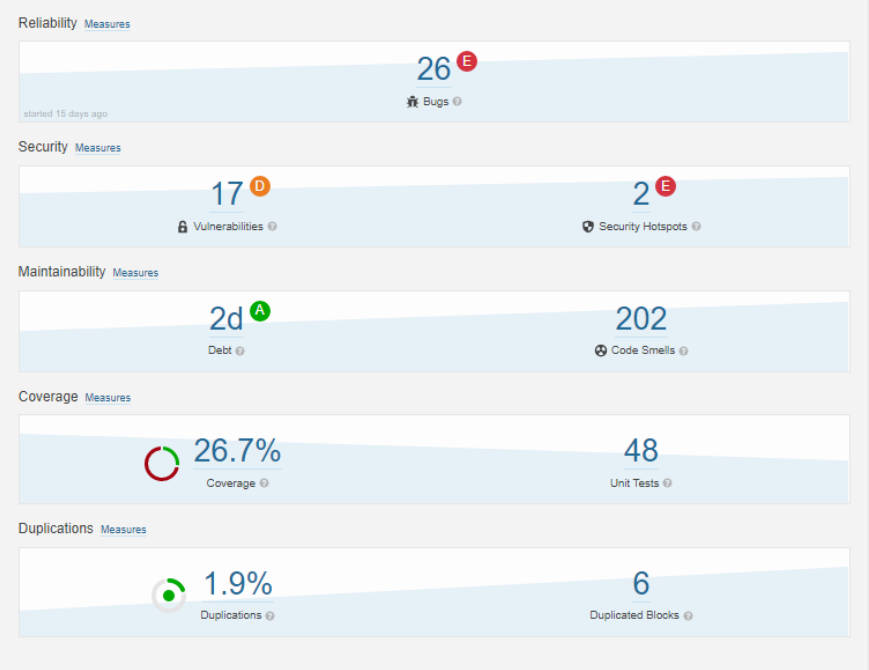
Aún así, los defectos encontrados no tienen una gran repercusión negativa y, tal y como muestra la deuda técnica no se tardarían mucho en corregir. Además, muchos de esos fallos estaban ya incluidos en el proyecto base proporcionado.

Es por ello, que el equipo de trabajo se muestra satisfecho con el trabajo realizado y piensa que se ha implementado un código más que apto. Sin embargo, se da por hecho que, aunque en menor medida debido a los conocimientos adquiridos, en el Sprint 3 el número de bugs y malos olores se incrementará debido a una mayor dificultad en las tareas.

Sprint 3

Resumen del análisis

Tras culminar las tareas del Sprint 3, se vuelve a realizar un análisis del código que se encuentra en la rama *master*, a través de SonarCloud.



Una vez se poseen los resultados, se observa que la **fiabilidad**, se sigue calificando como una E, ya que los bugs se han visto incrementados, de 18 a 26.

Por otro lado, el apartado de **seguridad** es el que menos se ha visto afectado, ya que las vulnerabilidades, que siguen siendo calificadas con una D, han sufrido un incremento menor, pasando de 13 a 17 y los puntos de acceso (también E) se han mantenido en 2, es decir, los mismos que traía la aplicación por defecto.

En cuanto a la **mantenibilidad**, se han aumentado los olores en 74 unidades. Sin embargo, y aunque también ha aumentado en un día, no ha afectado a la calificación de la deuda técnica, que sigue siendo una A.

La **cobertura**, por su parte, si ha sufrido una gran caída, pues los test unitarios no han aumentado y, por tanto, el porcentaje ha caído a un 26.7%.

Finalmente, las **duplicaciones** también han crecido de 2 a 6, lo que supone un aumento del porcentaje que se sitúa ahora en un 1.9%.

Bugs encontrados

Se han de tener en cuenta los bugs del Sprint 2, puesto que, aunque han aumentado su número de apariciones, no se repetirán para evitar la saturación del documento.

1 – Los métodos no deben llamar a métodos de la misma clase con valores "*@Transactional*" incompatibles

SonarCloud marca este bug 3 veces, dado que el método *findAll()* del servicio está marcado con la etiqueta "*@Transactional*".

Para solucionar el bug eliminaremos la etiqueta "*@Transactional*" en el método *findAll()* de los servicios.

Olores encontrados

Se han de tener en cuenta los olores del Sprint 2, puesto que, al igual que con los bugs, no se repetirán.

Olores blocker

No han variado con respecto a los existentes en el Sprint 2. Acceder a los olores blocker del sprint 2.

Olores critical

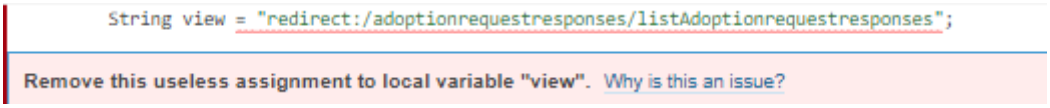
No han variado con respecto a los existentes en el Sprint 2. Acceder a los olores critical del sprint 2.

Olores major

Con respecto a los olores major del sprint 2, se ha encontrado un nuevo tipo de este olor:

1 – Variable local con un valor que no es usado

Este tipo de olor, muy parecido a "Asignación de variable inútil" del anterior Sprint. En este caso, se le da a la variable un valor que no es usado en ninguna instrucción posterior.



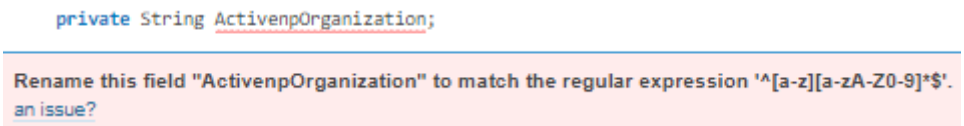
Por ello, una solución posible podría pasar por comprobar que verdaderamente la variable no se esté usando y, si es así, borrarla.

Olores minor

Con respecto a los olores minor del sprint 2, se han encontrado cuatro nuevos tipos de este olor:

1 – Renombrar "x" para que coincida con una expresión regular

Este olor nos indica que no está usada una expresión regular para nombrar ese atributo. Para que la programación en equipo sea más eficaz se debe intentar mantener una nomenclatura fija.



La solución pasaría por usar el formato *camelCase* para nombrar variables, es decir comenzando su nombre por letra minúscula.

2 – Convertir "x" en *static final constant* o en un atributo *private*

Este olor nos indica que utilizar *public* para nombrar atributos no respeta el principio de encapsulación y tendría varias desventajas:

- No se pueden añadir comportamientos adicionales

- La representación interna está expuesta y no puede ser modificada posteriormente
- Los valores están sujetos a cambios desde cualquier parte del código.

```
public Double totalDonation ;
```

Make totalDonation a static final constant or non-public and provide accessors if needed. Why is this an issue?

Por ello, la posible solución pasaría por usar *private* en lugar de *public*, lo que evitaría modificaciones no autorizadas.

3 – Reemplazar la expresión *if-then-else* por un *return*

Este olor nos indica que se ha usado una expresión *this-then-else* innecesariament.

```
private Boolean mismaCausa(Causa cau1, Causa cau2) {
    if(cau1.getId() == cau2.getId()) {
        return false;
    }else {
        if (cau1.getName() != cau2.getName()) {
```

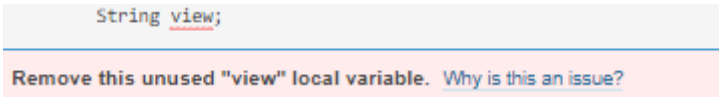
Replace this if-then-else statement by a single return statement. Why is this an issue?
Code Smell ▾ Minor ▾ Open ▾ Not assigned ▾ 2min effort Comment

```
            return true;

        } else {
            return false;
        }
    }


}
```

La solución, esta vez, estaría en reemplazar esta expresión, pues está dentro de un *if-else* donde en el "*if*" ya encontramos el retorno del *false*. Esto hace que el segundo sea totalmente innecesario.

4 – Eliminar la variable "x" que no se utiliza

Este olor nos indica que la variable inicializada no se utiliza posteriormente.



La solución, por tanto, sería borrar esta línea de código.

Olores info

No han variado con respecto a los existentes en el Sprint 2. Acceder a los olores info del sprint 2.

Conclusiones

Tras realizar el análisis de este Sprint, el Scrum Team se muestra bastante satisfecho con la mayoría de los resultados.

Los bugs se han incrementado en un 50%, mientras que los malos olores han aumentado menos de un 60%. Por su parte, la cobertura ha decrecido al no realizarse más test y el código duplicado también ha aumentado. Estos resultados dan a entender que:

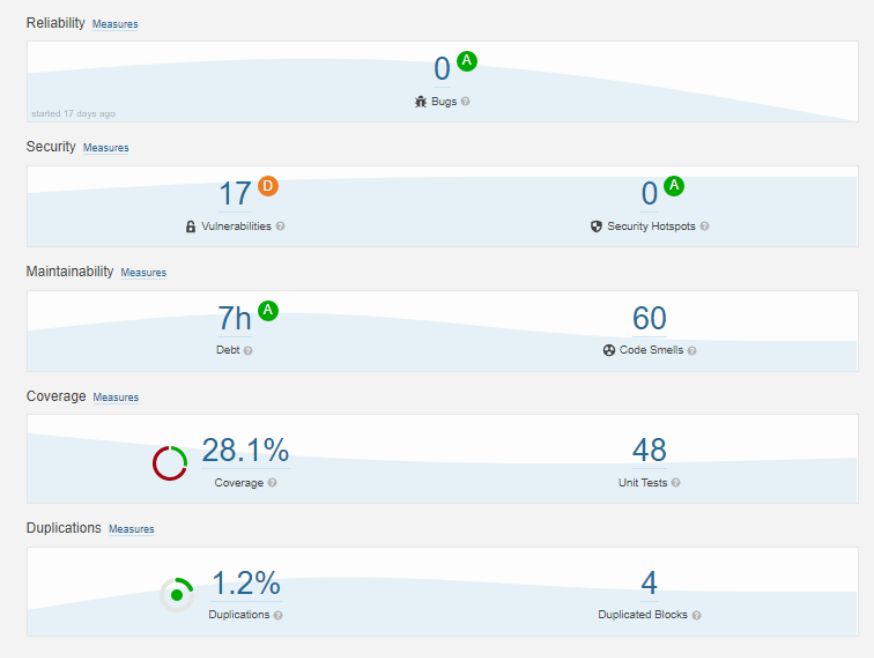
La complejidad del código es mayor, por lo que, de forma casi irremediable tenían que empeorar los datos con respecto al anterior sprint. Sin embargo, estos no han subido de forma lineal, sino que, debido a los conocimientos adquiridos por el equipo, se han producido en menor cantidad.

Por ello, el grupo confía en haber realizado un buen trabajo y cree que, aunque lógicamente en global se hayan empeorado los datos, se ha mejorado la calidad del código realizado y, además, tras la refactorización, la mayoría de los resultados disminuirán sus valores.

Sprint 3 tras la refactorización

Resumen del análisis

Tras la refactorización de código se realiza un nuevo análisis a través de SonarCloud, donde se aprecia una notable mejora.



En dicho análisis se observa como el apartado de **fiabilidad**, los bugs han descendido de 26 que había al finalizar las tareas, hasta 0, consiguiendo así una calificación de A.

Por otro lado, el apartado de **seguridad** no se ha visto apenas afectado, pues, aunque los puntos de acceso han bajado a 0, y por tanto se califican ahora como A, las vulnerabilidades se han mantenido en 17, lo que equivale a una D.

Por otro lado, la **mantenibilidad** es la que ha sufrido una notoria mejora. Pues, aunque la calificación de A en la deuda técnica no haya cambiado, se ha pasado de 2 días a 7h. Esto se debe a que los malos olores se han reducido de 202 a 60.

La **cobertura**, sin embargo, se ha mantenido prácticamente igual, pues los test unitarios no han cambiado. Eso sí, la cobertura ha mejorado hasta un 28.1%.

Finalmente, las **duplicaciones** han descendido hasta un 1.2%, pues ahora solo encontramos 4 bloques duplicados.

Olores encontrados

Se han de tener en cuenta los olores encontrados durante el Sprint 2 y el Sprint 3, pues estos no se repetirán para evitar la saturación del documento.

Olores blocker

No han variado con respecto a los existentes en el Sprint 2. Acceder a los olores blocker del sprint 2.

Olores critical

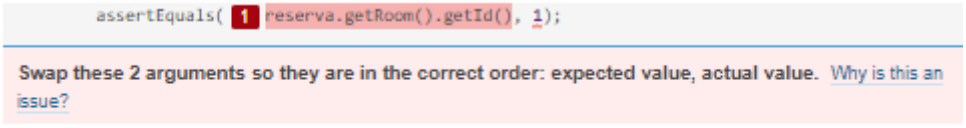
No han variado con respecto a los existentes en el Sprint 2. Acceder a los olores critical del sprint 2.

Olores major

Además de los olores major encontrados anteriormente durante el sprint 2 y el sprint 3. Se ha encontrado un nuevo tipo de olor:

1 – Cambiar el orden de los valores para que se encuentren en el orden correcto

Este olor nos indica que se ha invertido el orden de los valores que se debían incluir en el assertEquals().



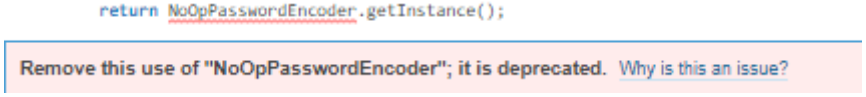
Debido a esto, la solución sería, simplemente, cambiar el orden .

Olores minor

Con respecto a los distintos tipos de olores minor del sprint 2 y el sprint 3, solo se ha encontrado un nuevo tipo:

1 – La expresión “x” está obsoleta

Este olor nos indica que la expresión utilizada está obsoleta.



Por eso, la solución sería modificar esa expresión que nos marquen por otra que no esté obsoleta.

Olores info

Tras la refactorización se han eliminado todos los olores info.

Conclusiones

Tras realizar el análisis después de la refactorización, el equipo de trabajo está muy satisfecho con los resultados.

Los bugs han disminuido un 100% y los malos olores han hecho lo propio en más de un 70%. Los test, sin embargo, no han variado en exceso pues no se ha realizado ninguno más. Por su parte, las duplicaciones de código, aunque en menor medida, también han descendido.

Esto indica que se ha realizado una buena refactorización del código, pues los bugs han desaparecido y la bajada de los malos olores en un 70% podría haber sido más, pero, de forma inconsciente, se han creado nuevos defectos durante el proceso.

Como conclusión, el grupo confía en que el proyecto, que ahora es mucho más mantenible y fiable, pueda continuar en esta línea. Además, se espera mejorar el apartado de cobertura con la realización de nuevos test y terminar de eliminar todos los bloques de código duplicados.