

Universidad de Sevilla  
Escuela Técnica Superior de Ingeniería Informática  
Grupo G2-23

## Análisis de código fuente y métricas asociadas



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

Curso 2020 – 2021

Fecha	Versión
<03/05/2021>	V01r01

**Control de Versiones**

Fecha	Versión	Descripción
<03/05/2021>	v01r01	Versión inicial



## Índice

I.1.	Análisis de código entrega S2 .....	1
I.1.1.	Descripción general .....	1
I.1.2.	Análisis de potenciales bugs .....	1
I.1.3.	Olores (code smells) .....	2
I.2.	Análisis de código entrega S3 .....	6
I.2.1.	Descripción general .....	6
I.2.2.	Análisis de potenciales bugs .....	7
I.2.3.	Olores (code smells) .....	8
I.3.	Conclusiones .....	10

## I.1. Análisis de código entrega S2

### I.1.1. Descripción general

En el dashboard de SonarCloud podemos ver un resumen general del estado del proyecto en el momento de la entrega del S2:

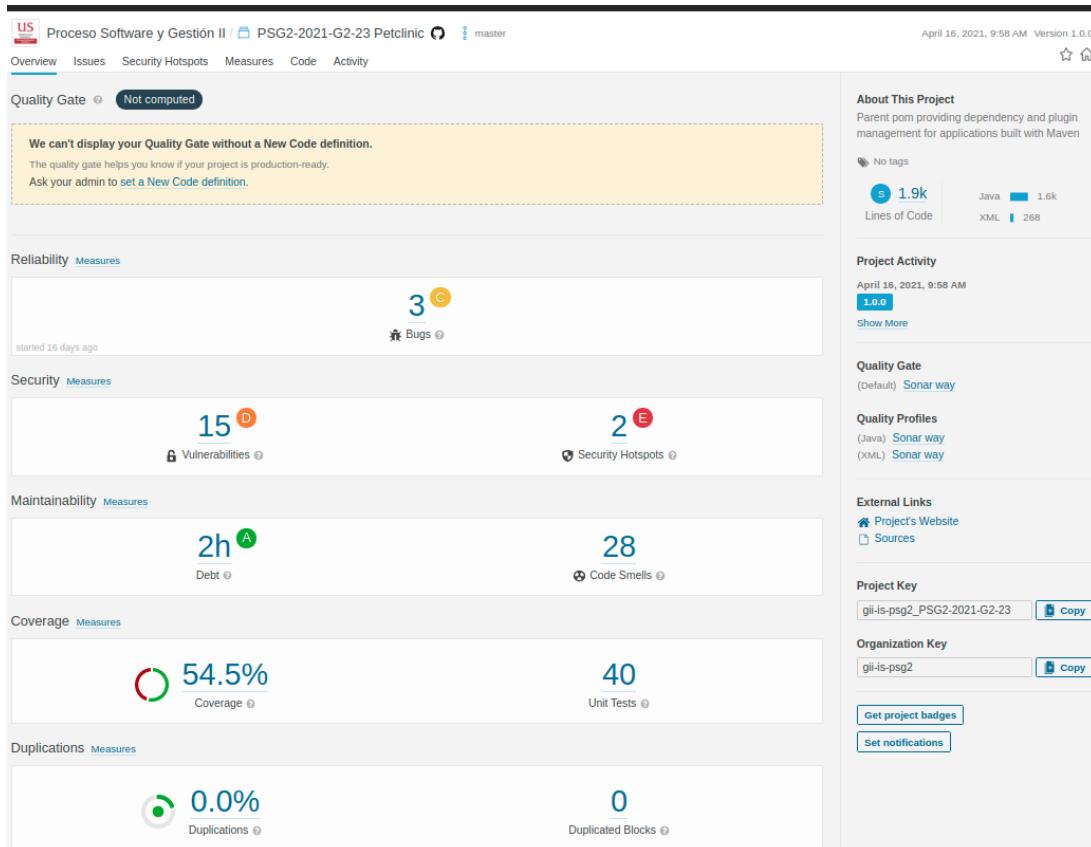


Fig 1: Dashboard de SonarCloud tras la entrega S2

Muestra 3 posibles bugs, varias vulnerabilidades de seguridad y una estimación del coste en tiempo de resolver la deuda técnica basándose en los olores (code smells) detectados.

A tener en cuenta que los tests sólo cubren en torno a la mitad del proyecto (54,5%) por lo que la calidad del mismo puede empezar a resentirse.

### I.1.2. Análisis de potenciales bugs



En la sección *Reliability* se muestra el resultado de un análisis de posibles Bugs y una estimación de tiempo necesario para ser corregido cada uno de ellos:

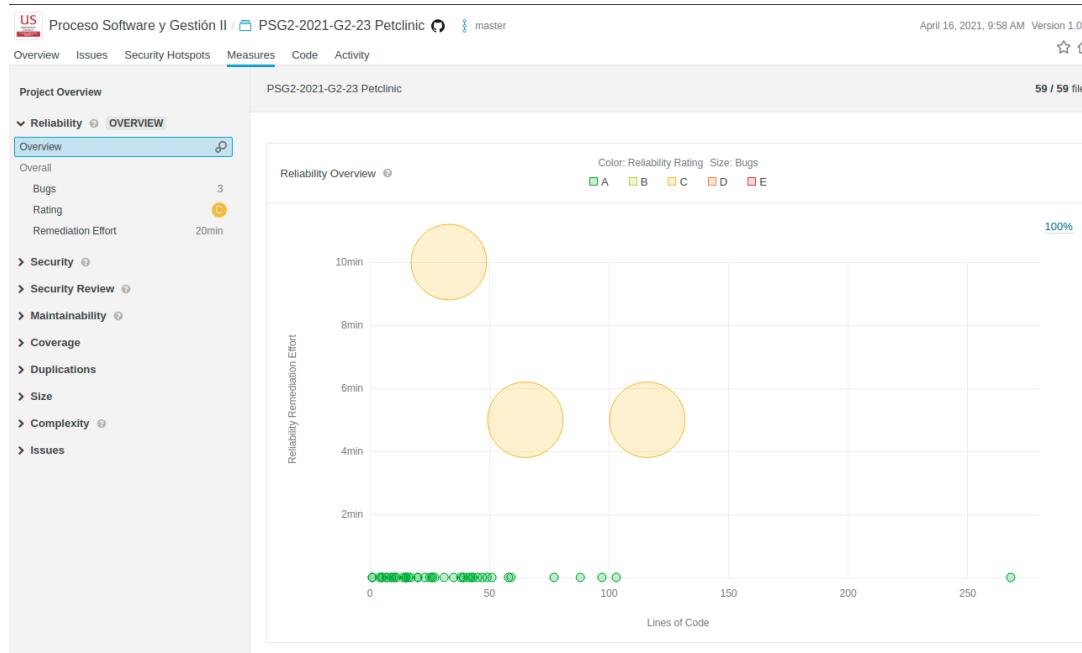


Fig 2: Posibles bugs y estimación de resolución de cada uno de ellos

Se muestra la detección de tres posibles bugs (errores que sólo se producen en determinadas circunstancias y no están siendo controlados por el código desarrollado), estimándose 10 minutos en la resolución de uno de ellos y 5 minutos en la resolución de cada uno de los otros dos.

#### I.1.3. Olores (code smells)

En la sección *Maintainability* se evalúan los riesgos a largo plazo. Se hace una estimación de la deuda técnica acumulada por desvíos en las buenas prácticas en codificación. La situación de nuestro proyecto en el momento de la entrega del S2 es:

	<b>Proceso de Software y Gestión 2 – G2-23</b> <b>Metodología de administración</b>
---	--

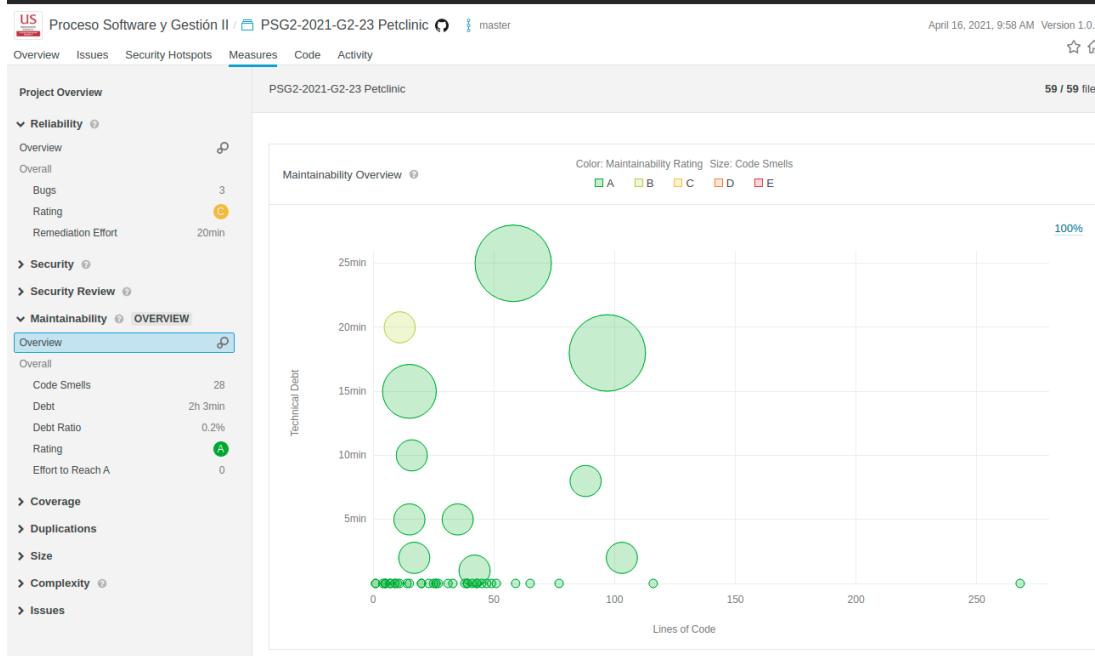
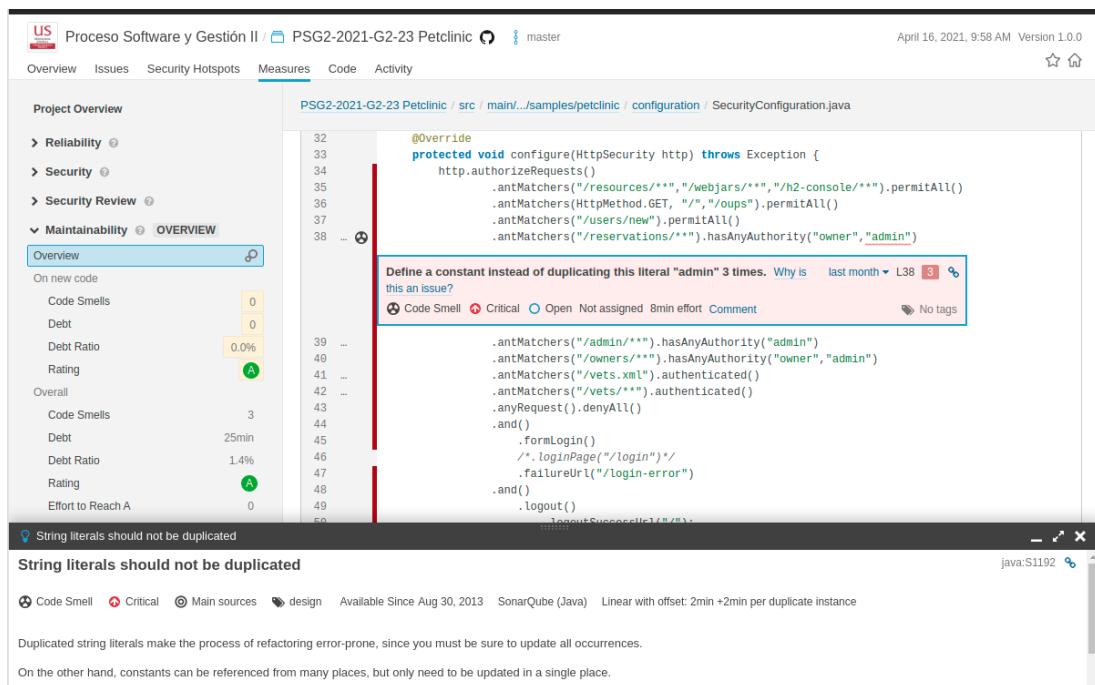


Fig 3: Code Smells tras la entrega S2

Han sido reconocidos 28 code smells, entre otros:

A) Change preventers: Cambiar algo en una parte del código implica cambios en otras partes.



	Proceso de Software y Gestión 2 – G2-23 <b>Metodología de administración</b>
---	---

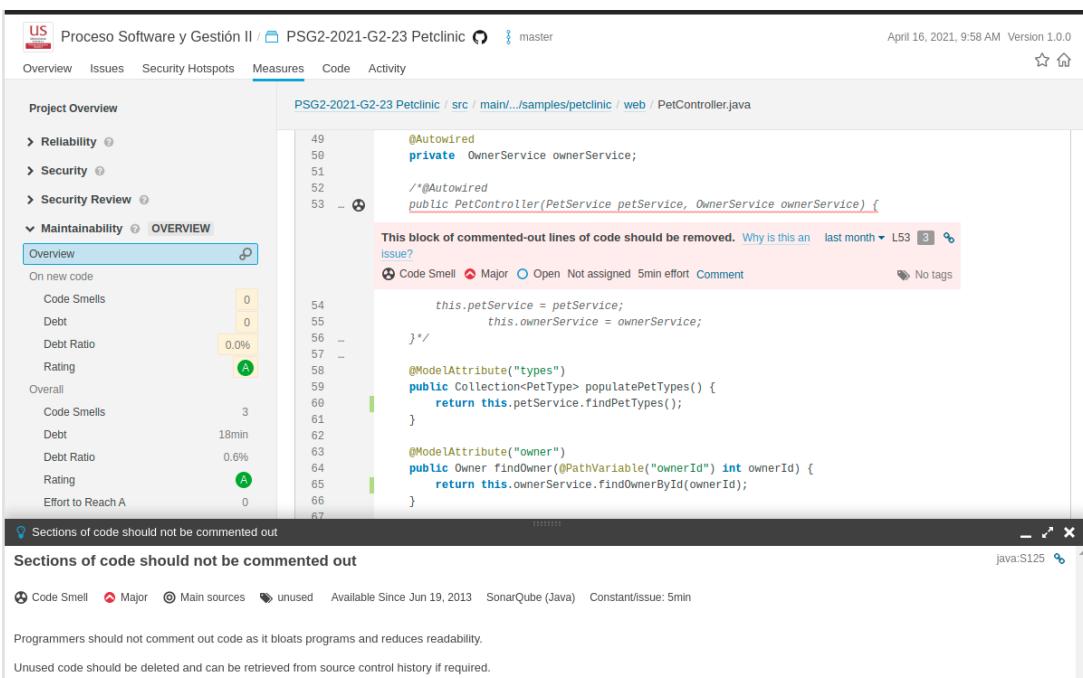
Fig 4: Change Preventer, varios usos del mismo literal

Se está usando un literal en lugar de usar una variable a la que asignar el valor de dicho literal.

La severidad asignada es crítica ya que cambiar el literal en un punto implica cambiarlo en otros. Cuantos más cambios más posibilidad de error y más lentitud en hacer efectivo dicho cambio.

La manera de resolver este problema es asignar el literal a una variable que se usará en todos los lugares del código en los que se está usando el literal.

B) Bloaters: Partes del código que llegan a crecer tanto que no pueden ser manejados de forma efectiva. En nuestra aplicación nos encontramos código comentado en lugar de ser eliminado:



The screenshot shows the SonarQube interface for the project 'Proceso Software y Gestión II / PSG2-2021-G2-23 Petclinic'. The 'Measures' tab is selected, showing the 'Project Overview' for the file 'PetController.java'. A code smell is highlighted at line 53, indicating a block of commented-out code that should be removed. The code snippet is as follows:

```

49     @Autowired
50     private OwnerService ownerService;
51
52     /*@Autowired
53     public PetController(PetService petService, OwnerService ownerService) {
54
55         this.petService = petService;
56         this.ownerService = ownerService;
57     }
58
59     @ModelAttribute("types")
60     public Collection<PetType> populatePetTypes() {
61         return this.petService.findPetTypes();
62     }
63
64     @ModelAttribute("owner")
65     public Owner findOwner(@PathVariable("ownerId") int ownerId) {
66         return this.ownerService.findOwnerById(ownerId);
67     }

```

A tooltip above the code reads: 'This block of commented-out lines of code should be removed. Why is this an issue?'. Below the code, there are buttons for 'Code Smell', 'Major', 'Open', 'Not assigned', '5min effort', 'Comment', and 'No tags'. At the bottom of the interface, there is a note: 'Programmers should not comment out code as it bloats programs and reduces readability.' and 'Unused code should be deleted and can be retrieved from source control history if required.'

Fig 5: Código comentado que dificulta la legibilidad del código en uso.

Se ha comentado código en lugar de eliminarlo.

La severidad asignada es Major ya que dificulta la lectura del código que sí está en uso.



La forma de resolver este problema es eliminar las líneas, ya que no se pierden por poder ser “rescatadas” en versiones previas del código fuente.

C) Dispensables: Algo innecesario cuya ausencia podría hacer un código más eficiente y fácil de entender. Podemos encontrar en nuestro código una variable privada de una clase con el mismo nombre que la propia clase:

The screenshot shows the SonarQube interface for a Java project named 'Petclinic'. The 'Measures' tab is selected. On the left, there's a 'Project Overview' sidebar with sections for Reliability, Security, Security Review, and Maintainability. Under Maintainability, the 'Overview' section is active, showing metrics like Code Smells (0), Debt (0), and Rating (A). The main panel displays the 'Vets.java' file. Line 33 contains the problematic code: `private List<Vet> vets;`. A SonarQube tooltip appears over this line, stating 'Rename field "vets" Why is this an issue?' with severity 'Major' and 'Open'. Below the code, a note explains: 'It's confusing to have a class member with the same name (case differences aside) as its enclosing class. This is particularly so when you consider the common practice of naming a class instance for the class itself.' At the bottom of the tooltip, it says 'Best practice dictates that any field or member with the same name as the enclosing class be renamed to be more descriptive of the particular aspect of the class it represents or holds.'

Fig 6: campo de una clase con el mismo nombre que la clase

Se ha puesto el mismo nombre a una clase y a una variable privada suya.

La severidad asignada es Major ya que la lectura del código se presta a confusión.

La solución de este problema consiste en cambiar el nombre de la variable.

Otro olor de tipo Dispensables es el mantener importaciones que no se usan:



The screenshot shows the SonarQube interface for a Java project named 'PSG2-2021-G2-23 Petclinic'. The 'Measures' tab is selected. On the left, there's a 'Project Overview' sidebar with sections like Reliability, Security, Security Review, and Maintainability. The Maintainability section is expanded, showing metrics such as Code Smells (0), Debt (0), Debt Ratio (0.0%), Rating (A), and Overall (Code Smells 1, Debt 2min, Debt Ratio 0.1%, Rating A). Below this is a 'Unnecessary imports should be removed' section. The main panel displays the Java code for 'OwnerController.java' with line numbers 21 to 37. An annotation at line 27 highlights an unused import: 'import org.springframework.samples.petclinic.service.VetService;'. A tooltip for this annotation provides a link to remove it and includes information about the author (Laurentiu Bogdan) and effort (2min). The bottom of the code editor shows a note about unnecessary imports being handled by the IDE.

Fig 7: Import no usado

Se ha mantenido un Import a un paquete que no se usa en el código.

La severidad asignada es Minor ya que afecta poco a la lectura y comprensión del código.

La solución a este problema sería eliminar el Import.

## I.2. Análisis de código entrega S3

### I.2.1. Descripción general

Cuadro resumen del estado del proyecto en el momento de la entrega del S3:

	<b>Proceso de Software y Gestión 2 – G2-23</b> <b>Metodología de administración</b>
---	--

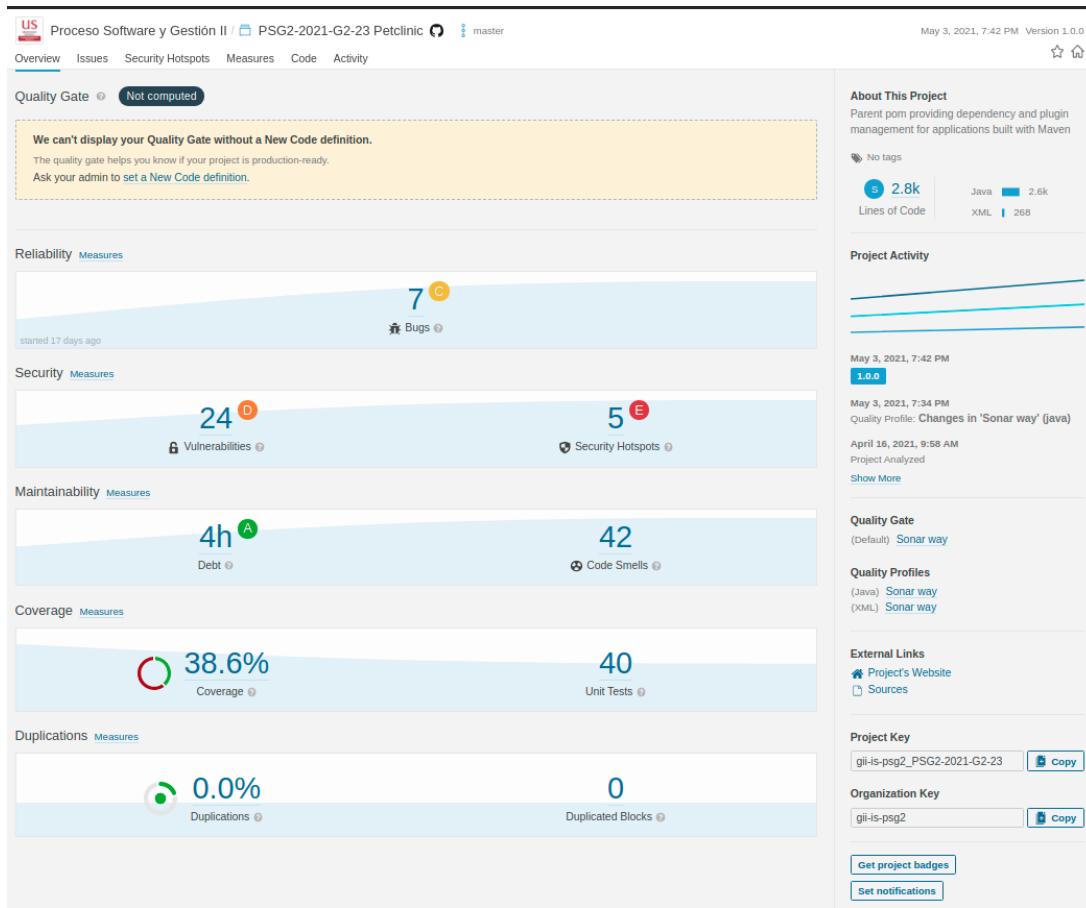


Fig 8: Dashboard S3

Muestra 7 posibles bugs, un incremento en vulnerabilidades de seguridad de aproximadamente un 40% respecto de S2 (de 15 a 24) y prácticamente se ha duplicado la estimación temporal necesaria para pagar la deuda técnica (de 2 horas con 28 code smells a 4 horas con 42 code smells).

Ya que no se han añadido nuevos tests pero sí ha aumentado la cantidad de código del proyecto el porcentaje cubierto por dichos tests ha disminuido (del 54,5% al 38,6% actual).

### I.2.2. Análisis de potenciales bugs

El aumento de líneas de código ha llevado aparejado un aumento en el número de bugs y en el tiempo estimado en resolverlos:

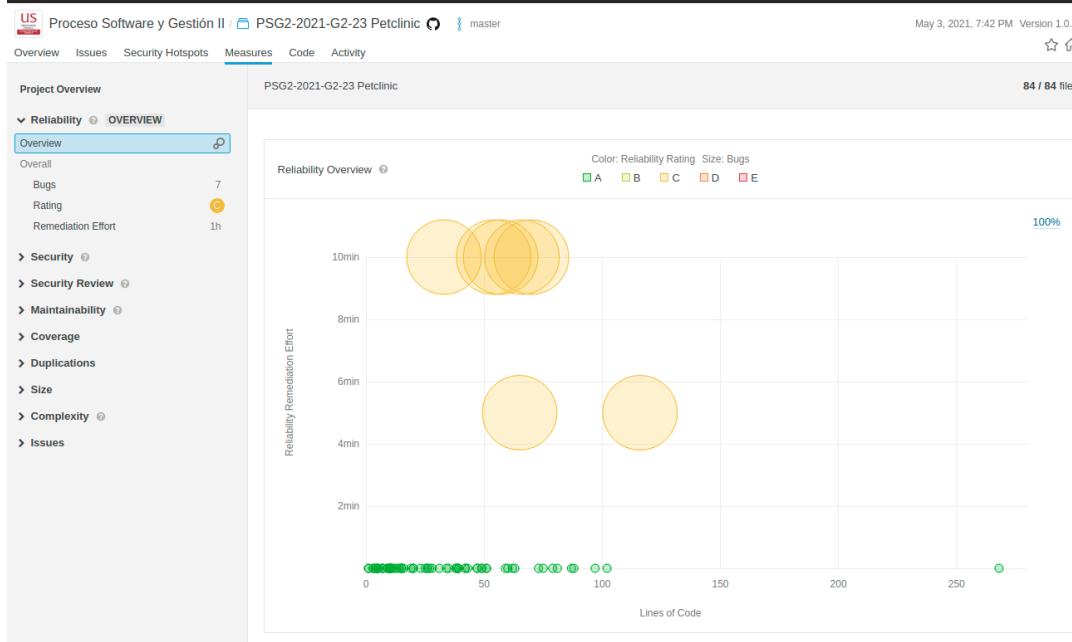


Fig 9: Posibles bugs y estimación de resolución de cada uno de ellos

Los bugs señalados son de la misma naturaleza que los preexistentes en S2, es decir, acceso a variables que podrían estar vacías pero no se está comprobando con `IsEmpty()` y comparaciones entre dos variable de sus posiciones de memoria en lugar de sus respectivos valores (se usa `==` o `!=` en lugar del método `equal()`).

#### I.2.3. Olores (code smells)

Al no haberse corregido los code smells de S2 y al haberse desarrollado más código durante S3 el número de code smells ha subido casi hasta duplicarse:

	<b>Proceso de Software y Gestión 2 – G2-23</b> <b>Metodología de administración</b>
---	--

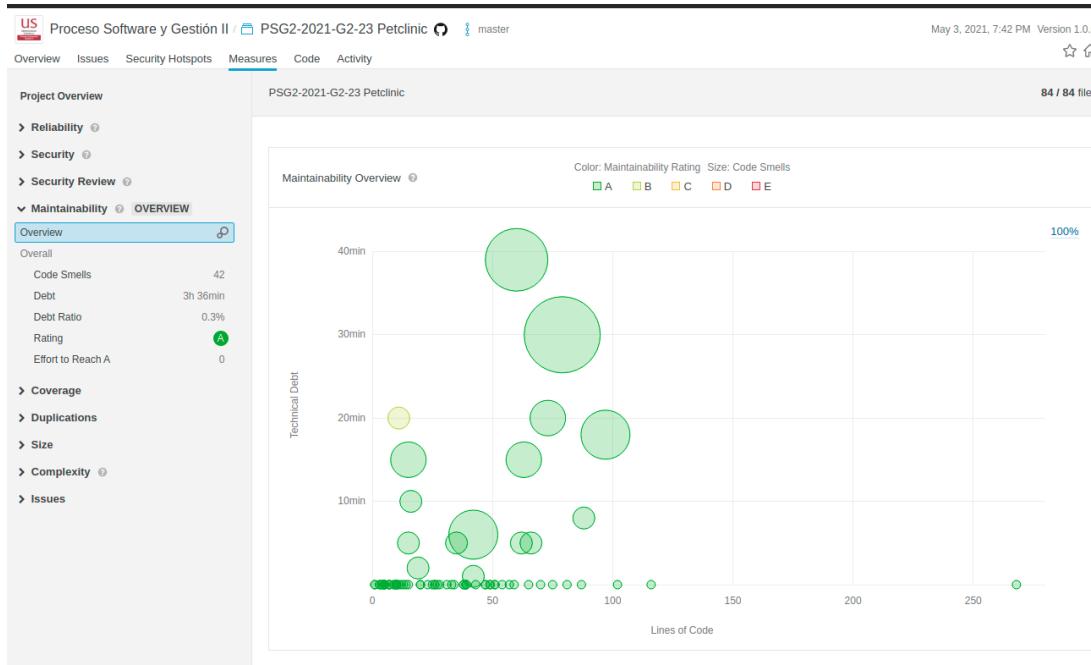
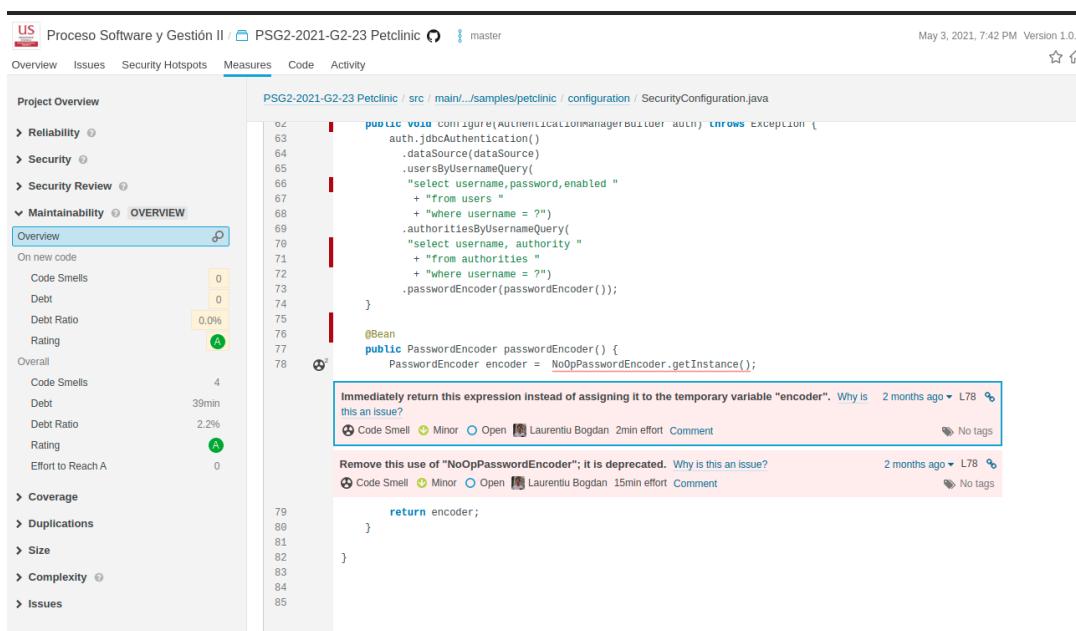


Fig 10: Code Smells tras la entrega S3

Algunos de los nuevos code smells surgidos en S3 son:

A) Dispensables: código innecesario.



```

62    public void configure(AuthenticationManagerBuilder auth) throws Exception {
63        auth.jdbcAuthentication()
64            .dataSource(dataSource)
65            .usersByUsernameQuery(
66                "select username,password(enabled"
67                + "from users "
68                + "where username = ?")
69            .authoritiesByUsernameQuery(
70                "select username, authority"
71                + "from authorities "
72                + "where username = ?");
73        }
74    }
75
76    @Bean
77    public PasswordEncoder passwordEncoder() {
78        PasswordEncoder encoder = NoOpPasswordEncoder.getInstance();

```

Immediately return this expression instead of assigning it to the temporary variable "encoder". Why is this an issue? 2 months ago L78 %

Remove this use of "NoOpPasswordEncoder"; it is deprecated. Why is this an issue? 2 months ago L78 %

Fig 11: Dos ejemplos de Dispensable smell en el mismo fragmento de código



Nos encontramos con un método obsoleto que se ha usado y que en la próxima actualización de software podría dejar de funcionar y, en el mismo trozo de código tenemos una variable que define justo la línea antes de ser devuelta en un return.

### B) Object-Orientation Abusers: Uso incorrecto de la orientación a objetos:

The screenshot shows the SonarQube interface for a project named "PSG2-2021-G2-23 Petclinic". The left sidebar displays various metrics and reports. The main area shows a Java file named "EntityUtils.java" with two highlighted code smells. The first smell is at line 33, marked with a red circle, and the second is at line 49, also marked with a red circle. Both smells are categorized as "Code Smell" and "Major". The first smell is described as "public abstract class EntityUtils {". The second smell is described as "Convert the abstract class "EntityUtils" into an interface.". The code snippet for EntityUtils.java is as follows:

```
23 * ULLIY: IMPLUS for handling entities. Separate from the baseentity class mainly
24 * because of dependency on the ORM-associated ObjectRetrievalFailureException.
25 *
26 * @author Juergen Hoeller
27 * @author Sam Brannen
28 * @see org.springframework.samples.petclinic.model.BaseEntity
29 * @since 29.10.2003
30 */
31
32 public abstract class EntityUtils {
33
34     /**
35      * Look up the entity of the given class with the given id in the given collection.
36      * @param entities the collection to search
37      * @param entityClass the entity class to look up
38      * @param entityId the entity id to look up
39      * @return the found entity
40      * @throws ObjectRetrievalFailureException if the entity was not found
41     */
42     public static <T extends BaseEntity> T getById(Collection<T> entities, Class<T> entityClass, int entityId)
43         throws ObjectRetrievalFailureException {
44         for (T entity : entities) {
45             if (entity.getId() == entityId && entityClass.isInstance(entity)) {
46                 return entity;
47             }
48         }
49         throw new ObjectRetrievalFailureException(entityClass, entityId);
50     }
51
52 }
53
54 }
```

Fig 12: Dos ejemplos de Objetc-Orientation Abusers smell en el mismo fragmento de código

Se trata de una clase abstracta que no debería tener constructores públicos y posiblemente además sea más recomendable que se redefiniera como interfaz en lugar de mantenerla como clase abstracta.

## I.3. Conclusiones

En la entrega S2 hubo en general una aceptable calidad del código, el cual arrastraba problemas menores y fáciles de resolver (tanto en tiempo como en dificultad) al tratarse de un momento temprano de desarrollo.

Para la S3 no se han corregido los errores anteriores y, al incrementar la cantidad de código, ha aumentado el número de smells y disminuido el porcentaje de código cubierto por los tests ya que estos no se han implementado.

	Proceso de Software y Gestión 2 – G2-23
	<b>Metodología de administración</b>

En definitiva, sigue habiendo una cantidad razonablemente pequeña de errores cuyas correcciones no implicarían mucho tiempo pero es previsible que, de no corregirlos el aumento de la deuda técnica aumente rápidamente.