

PIQUÉ BLINDERS

PLAN DE MANTENIMIENTO SONARCLOUD

Introducción

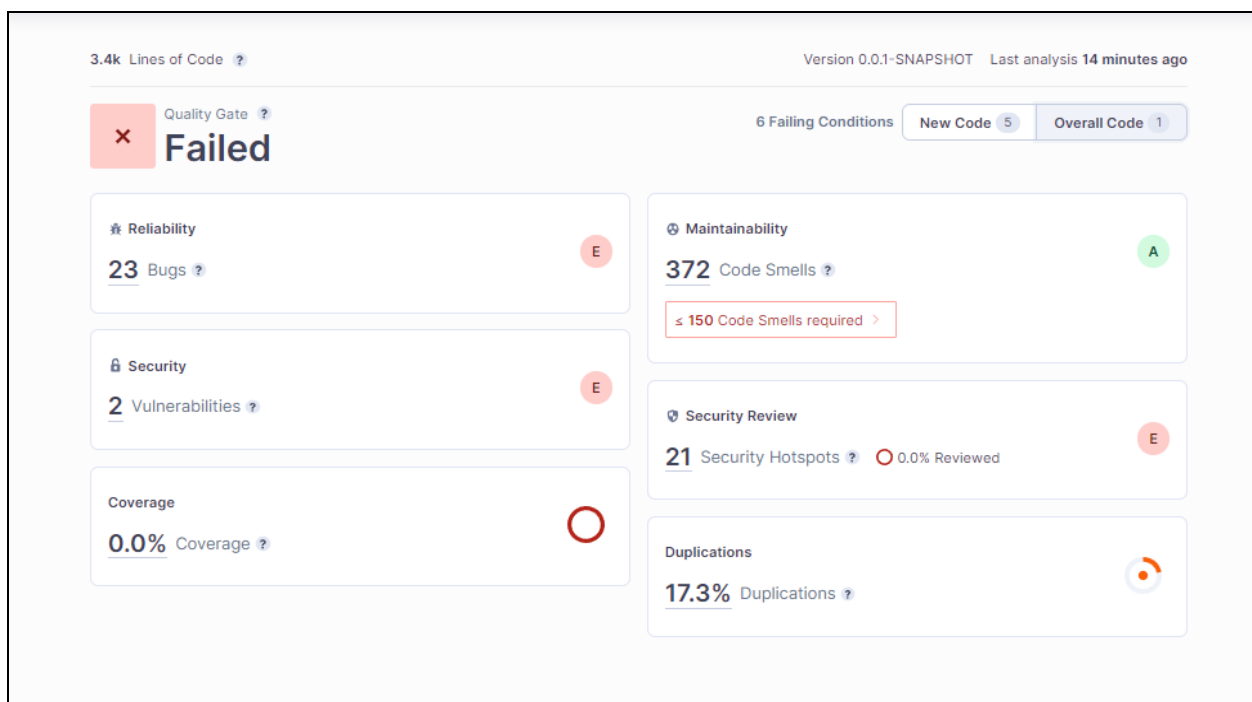
La realización de un Plan de Mantenimiento incluye el análisis de los medios de producción y su conservación, tus métodos de trabajo, objetivos, recursos y la especificidad de tu sector, todo ello con el objeto de optimizar la disponibilidad y la eficiencia de esos medios.



Realizaremos un plan de mantenimiento a través de la herramienta de SonarCloud es un servicio de análisis estático de código basado en la nube y que lo ofrece la empresa responsable de SonarQube. Además de medir la seguridad de nuestras aplicaciones, nos ayuda a mejorar la mantenibilidad y confiabilidad de nuestro código.

Primer análisis

Partiremos de este análisis inicial (hecho de forma manual) y a raíz de los resultados tomaremos las operaciones oportunas para resolver los problemas de cobertura, bugs, Code Smells, duplicaciones de código y vulnerabilidades.



Quality Gates

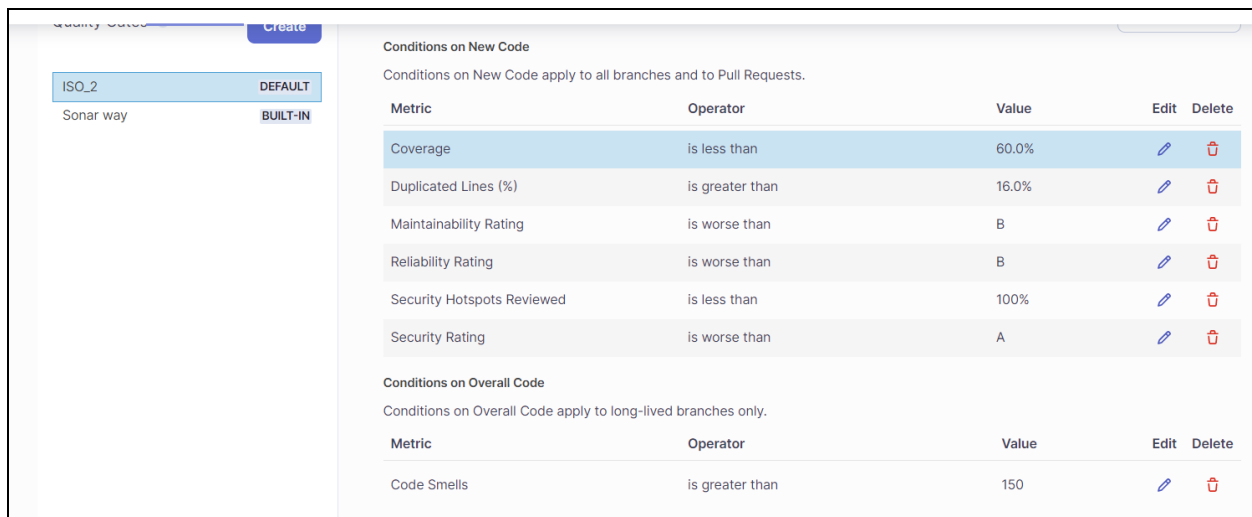
Los controles de calidad hacen cumplir una política de calidad en su organización al responder una pregunta: ¿mi proyecto está listo para su lanzamiento?

Para responder a esta pregunta, definimos un conjunto de condiciones contra las cuales se miden los proyectos. Por ejemplo:

- Cobertura de código superior al 60%
- ...

Idealmente, todos los proyectos utilizarán la misma puerta de calidad, pero eso no siempre es práctico. Por ejemplo, se puede dar la situación de que:

- La implementación tecnológica difiere de una aplicación a otra (es posible que no necesite la misma cobertura de código en el código nuevo para aplicaciones web que para las aplicaciones Java).
- Puede que precise garantizar requisitos más estrictos en algunas de sus aplicaciones...



Conditions on New Code				
Conditions on New Code apply to all branches and to Pull Requests.				
Metric	Operator	Value	Edit	Delete
Coverage	is less than	60.0%		
Duplicated Lines (%)	is greater than	16.0%		
Maintainability Rating	is worse than	B		
Reliability Rating	is worse than	B		
Security Hotspots Reviewed	is less than	100%		
Security Rating	is worse than	A		

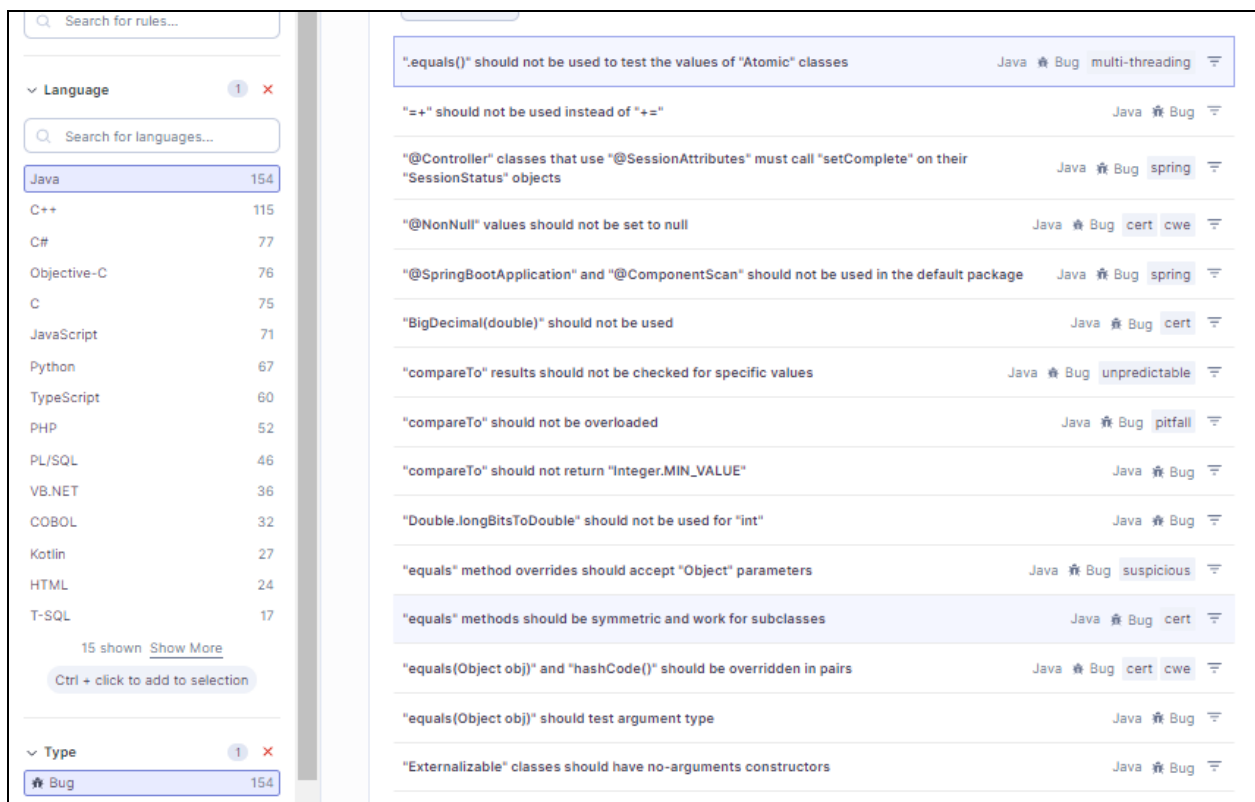
Conditions on Overall Code				
Conditions on Overall Code apply to long-lived branches only.				
Metric	Operator	Value	Edit	Delete
Code Smells	is greater than	150		

“Quality Gate aplicada a nuestro proyecto”

Mantenimiento preventivo

El software de mantenimiento preventivo está diseñado para simplificar y agilizar el proceso de gestión de operaciones de mantenimiento al evitar el tiempo de inactividad.

Para prevenir malas prácticas en código nos haremos cargo de estudiar las reglas de Java que se han establecido en sonarClub que están implicadas en la realización del código. Así como evitar el uso repetido de líneas de código, optimización del código, generación de comentarios...



Algunas de las acciones de mantenimiento preventivo que tomamos son la definición correcta de requisitos comprendiendo adecuadamente el dominio del problema, hacer un buen uso de las ramas de git para tratar de forma independiente las características del proyecto; y tratar con módulos y librerías ya testeadas y optimizadas.

Mantenimiento correctivo

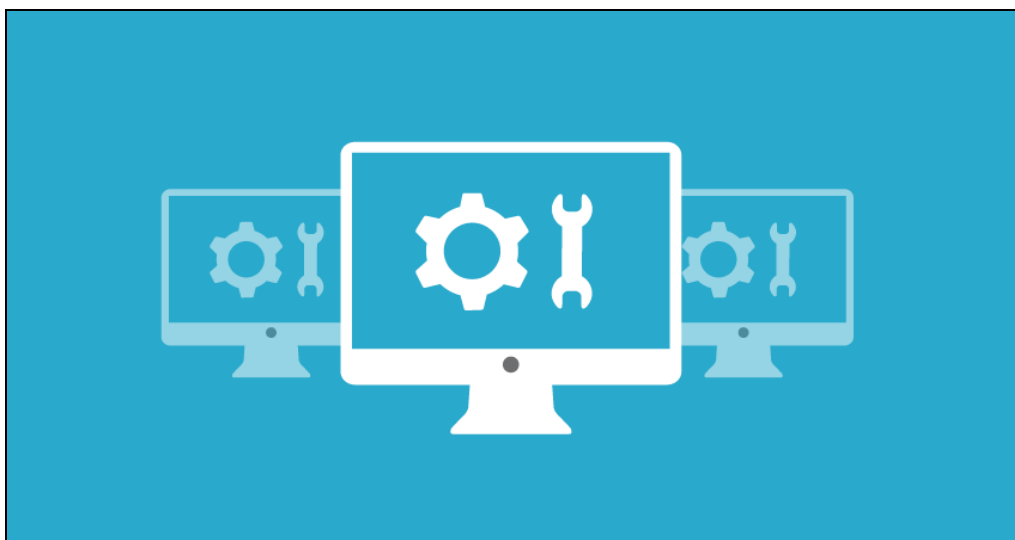
El mantenimiento correctivo del software es lo que normalmente se asociaría con el mantenimiento de cualquier tipo. El mantenimiento correcto del software aborda los errores y fallas dentro de las aplicaciones de software que podrían afectar varias partes de su software, incluido el diseño, la lógica y el código.

A continuación explicaremos cómo abordaremos los siguientes departamentos sobre los que llevaremos a cabo distintos tipos de correcciones. Y posteriormente vamos a señalar las mejoras en el código tras haber aplicado las diferentes acciones para así presentar la evolución de las métricas de calidad de nuestro proyecto.

Para llevar a cabo correctamente el plan de mantenimiento debemos tener en cuenta la prioridad de los elementos a tratar, entendemos que un error es crítico si:

- Pone en riesgo o compromete la seguridad de la aplicación o del cliente.
- Afecta a la eficacia de la aplicación.
- Afecta a la eficiencia de la aplicación.

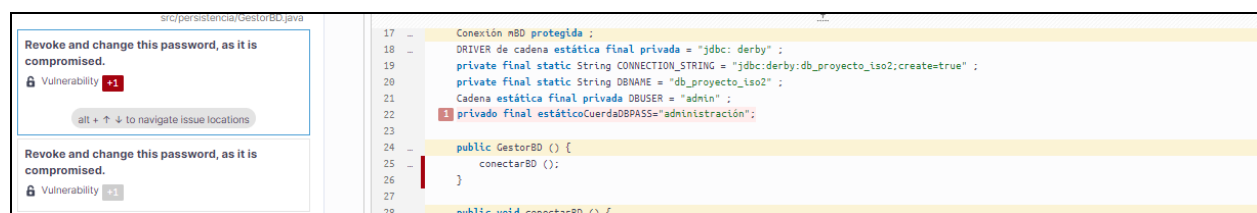
Ordenados por grado de criticidad.



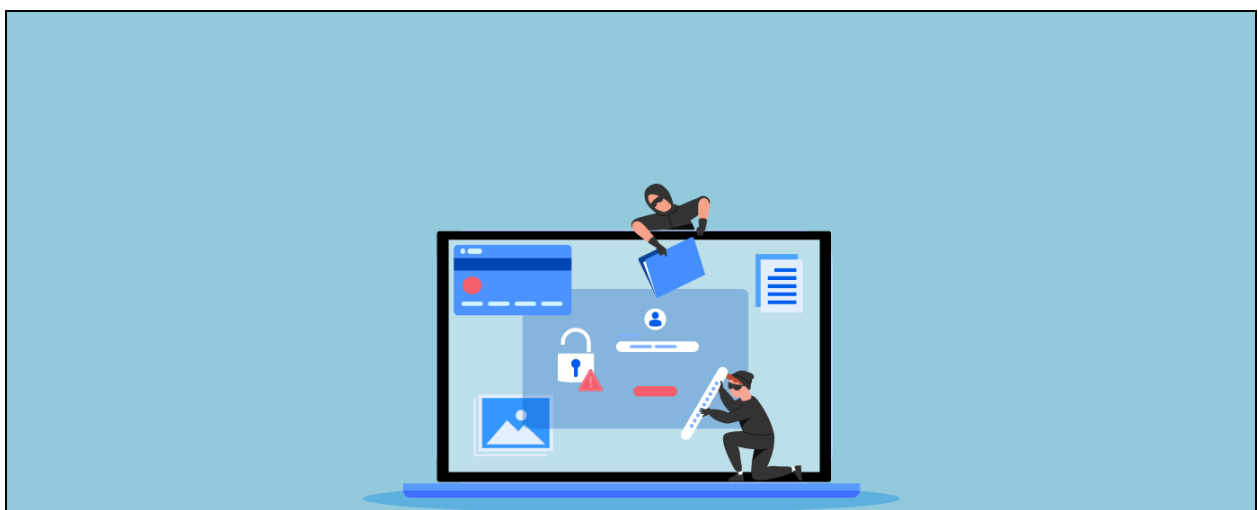
1. Security (Vulnerabilities)

La sección de seguridad va a ser la primera que trataremos y llevaremos a cabo un tipo de mantenimiento correctivo. Los exploits de vulnerabilidades de seguridad nos muestran cuán vulnerable puede ser realmente el código de software.

Los dos errores de vulnerabilidad son sobre un secreto codificado en su código. Se debe enumerar rápidamente dónde se usa este secreto, revocar y luego cambiarlo en cada sistema que lo use.

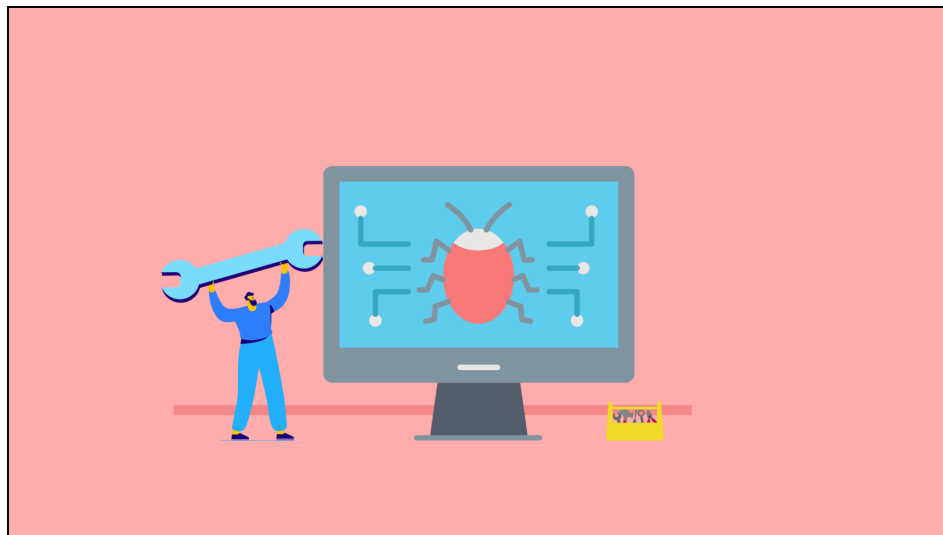
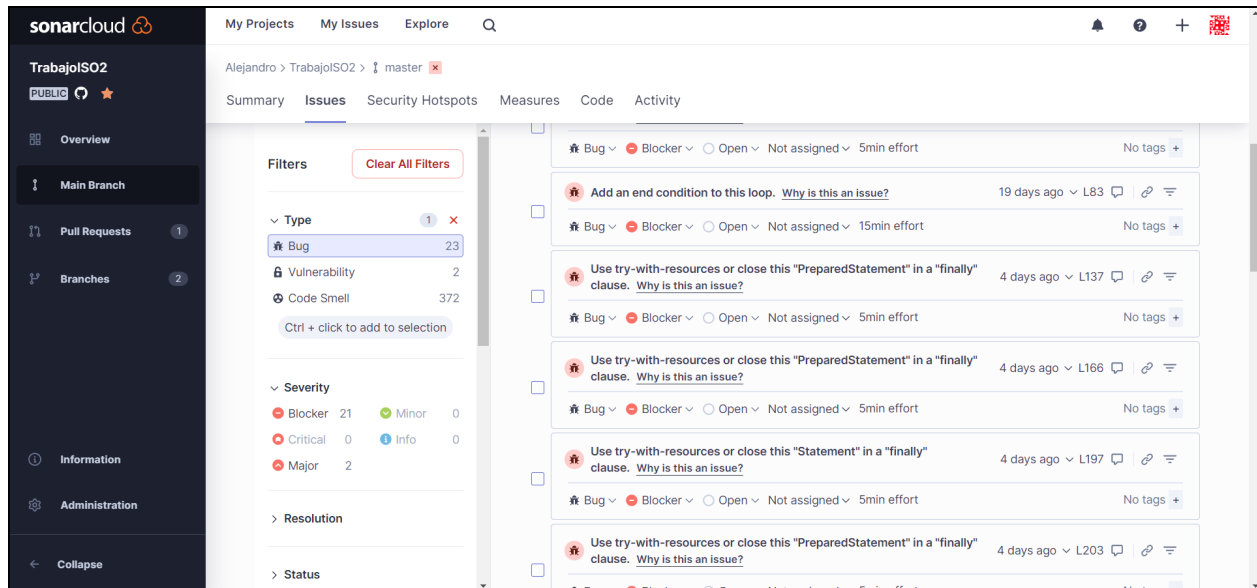


⁽¹⁾ Debido a que la solución a este fallo de vulnerabilidad es complejo y el contexto del proceso de solución se desvía de las competencias de la asignatura evitaremos las incidencias en seguridad.



2. Reliability (Bugs)

La fiabilidad se enfoca en los errores, un problema que representa algo incorrecto en el código. Si esto no se ha roto todavía, lo hará, y probablemente en el peor momento posible. Esto necesita ser arreglado cuanto antes.



1- Bug que sugiere cerrar "PreparedStatement" en una cláusula "finally".

Las conexiones, secuencias, archivos y otras clases que implementan la "Closeable" deben cerrarse después de su uso. Además, esa "close" llamada debe realizarse en un bloque "finally", de lo contrario, una excepción podría evitar que se realice la llamada.

Si no se cierran correctamente los recursos, se producirá una fuga de recursos que podría comprometer información de la aplicación.

Por lo tanto, podemos corregir el error tal y como se muestra en la siguiente imagen:

```
207
208     try {
209         // Datos iniciales de estudiantes
210         pstmt = this.mBD.prepareStatement(
211             "insert into ESTUDIANTES (DNI, NOMBRE, APELLIDOS, PASSWORD, TITULACION, CUALIF
212         pstmt.setString(1, "00000000A");
213         pstmt.setString(2, "Pepe");
214         pstmt.setString(3, "Perez");
215         pstmt.setString(4, "PepePerez");
216         pstmt.setString(5, "Ingenieria Informatica");
217         pstmt.setString(6, "Ingeniero SW");
218         pstmt.executeUpdate();
219
220     } catch (SQLException e) {
221         System.out.println(e.getErrorCode());
222         System.out.println(e.getSQLState());
223         System.out.println(e.getMessage());
224     } finally {
225         if (pstmt != null)
226             pstmt.close();
227     }
```


2- Bug bucle infinito

```
while (rs.next()) {
    List<Object> v = new ArrayList<Object>();
    int i = 1;
    while (true) {
        try {
            v.add(rs.getObject(i));
            i++;
        } catch (SQLException e) {
            break;
        }
    }
}
```

 Add an end condition to this loop. [Why is this an issue?](#) 23 days ago ▾ L85  

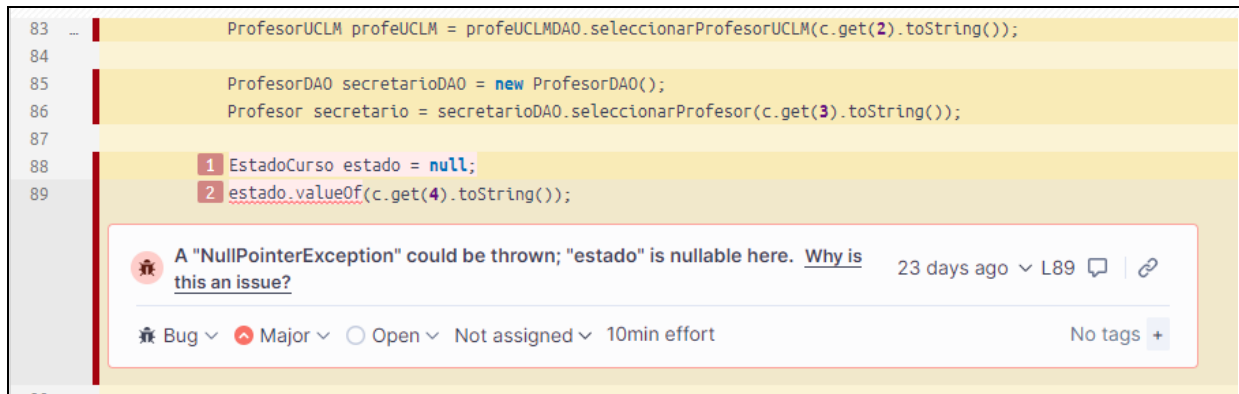
 Bug ▾  Blocker ▾  Open ▾ Not assigned ▾ 15min effort No tags 

```
74
75         while (rs.next()) {
76             List<Object> v = new ArrayList<Object>();
77             int i = 1;
78             boolean j=true;
79             while (j) {
80                 try {
81                     v.add(rs.getObject(i));
82                     i++;
83                 } catch (SQLException e) {
84                     j=false;
85                 }
86             }
87             resultado.add(v);
88         }
89     }
```

Se dió una solución al bug al establecer una condición de salida producida por una excepción de SQL (concretamente SQLException).

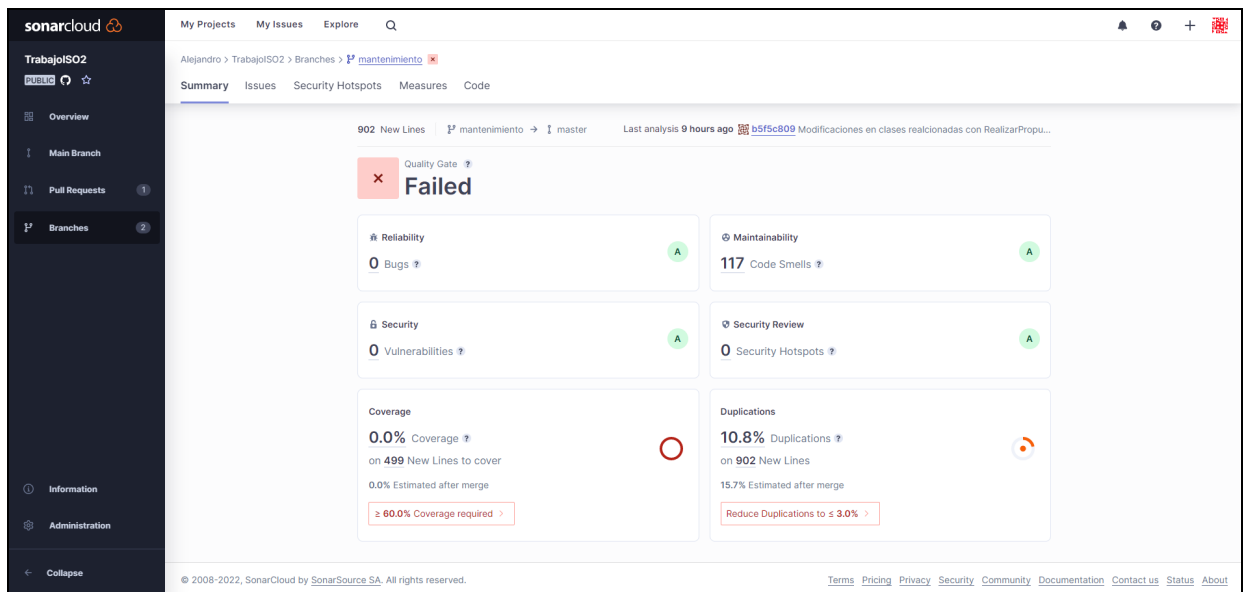
3- Bug arroja NullPointerException

Son 6 los casos a solucionar por este tipo de error. Hay que evitar inicializar éstas variables a null.



Para ello, sustituiremos las dos líneas de conflicto por la siguiente sentencia de código:

"EstadoCurso estado = EstadoCurso.valueOf(c.get(4).toString());"



4- Bug minor expresiones compareTo

Este bug se produce por utilizar -1 en lugar de 0 al realizar una comparación.

Si bien la mayoría de los métodos compareTo devuelve -1, 0 o 1, algunos no lo hacen, y probar el resultado de a contra compareTo un valor específico que no sea 0 podría generar falsos negativos.



```
127 ...
128 ...
129 ...
130 ...
131 ...
132 ...
133 ...
134 ...
135 ...
136 ...
137 ...
138 ...
139 ...
140 ...
141 ...
142 ...
143 ...
144 ...
145 ...
146 ...
```

src/negocio/controllers/GestorConsultas.java

Only the sign of the result should be examined.

Bug

Only the sign of the result should be examined.

Bug

...gocio/controllers/GestorPropuestasCursos.java

Only the sign of the result should be examined.

Bug

3 of 3 shown

Only the sign of the result should be examined. Why is this an issue? 27 minutes ago L134

Bug Minor Open Not assigned 5min effort No tags

```
public boolean ComprobarFechas(Date FechaInicio, Date FechaFin) {
    boolean bool = true;

    if (FechaFin == null || FechaInicio == null) {
        bool = false;
    } else {
        if (FechaFin.compareTo(FechaInicio) == -1 || FechaFin.compareTo(FechaInicio) == 0) {
            bool = false;
        }
    }

    return bool;
}

public boolean ComprobarEstadoCursoConFecha(EstadoCurso estado, Date FechaInicio, Date FechaFin) {
    boolean bool = true;

    if (estado == null) {
        bool = false;
    }
}
```


En lugar de poner la comparación == -1, hemos optado por poner un menor a 0, que es lo mismo ya que cuando se utiliza el método compare to, únicamente nos devuelve tres valores :-1 ,0 y 1, siendo así nuestra comparación nueva correcta.

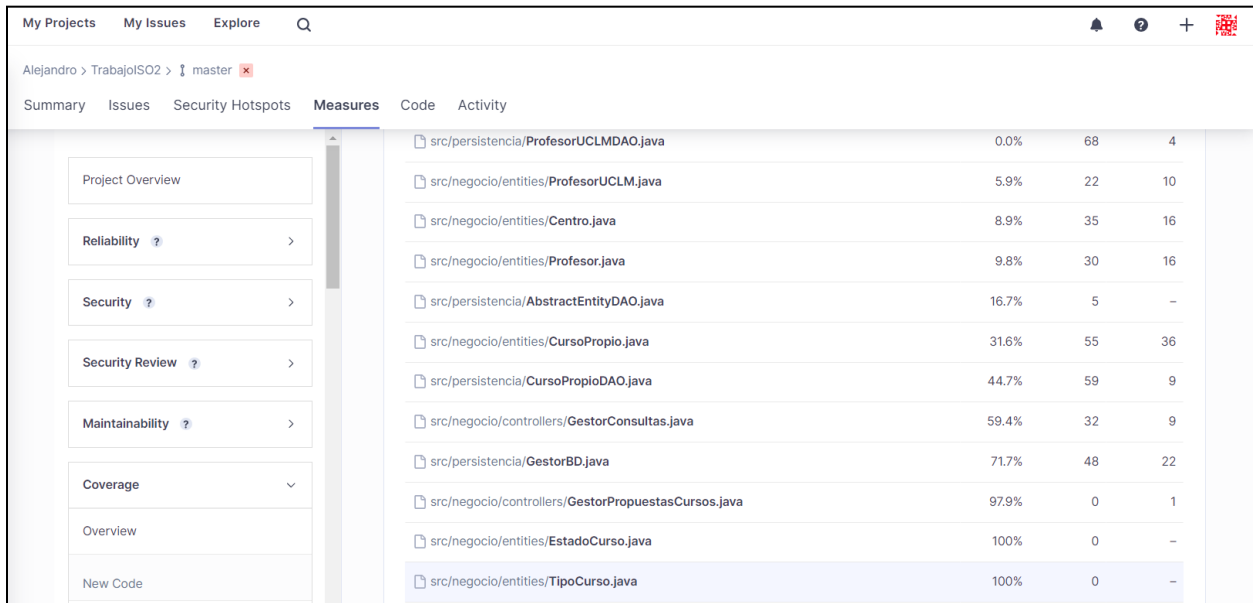
3. Coverage

La cobertura de código hace referencia a un mecanismo de medición de la calidad del software a través de la revisión de las pruebas. Específicamente, esta herramienta determina el rendimiento de las pruebas teniendo en cuenta su nivel de aplicación general.

En un inicio, establecimos en la “Quality Gate” que necesitábamos una cobertura para el proyecto superior a un 60%. Pero tras haber analizado las clases de la aplicación en busca de candidatas para establecer un plan de pruebas que cubriera correctamente las necesidades de nuestro código, finalmente nos ha bastado con una cobertura del 14,1 %.

El plan de pruebas es el siguiente:

 Test_PIQUE_BLINDERS



The screenshot shows a web-based code coverage tool interface. On the left, there is a sidebar with navigation links: 'My Projects', 'My Issues', 'Explore', and a search bar. Below these are tabs for 'Summary', 'Issues', 'Security Hotspots', 'Measures' (selected), 'Code', and 'Activity'. The 'Measures' tab displays a table with columns for file names, coverage percentages, and counts. The table lists various Java files and their corresponding coverage percentages, ranging from 0.0% to 100%.

File	Coverage	Count 1	Count 2
src/persistencia/ProfesorUCLMDAO.java	0.0%	68	4
src/negocio/entities/ProfesorUCLM.java	5.9%	22	10
src/negocio/entities/Centro.java	8.9%	35	16
src/negocio/entities/Profesor.java	9.8%	30	16
src/persistencia/AbstractEntityDAO.java	16.7%	5	-
src/negocio/entities/CursoPropio.java	31.6%	55	36
src/persistencia/CursoPropioDAO.java	44.7%	59	9
src/negocio/controllers/GestorConsultas.java	59.4%	32	9
src/persistencia/GestorBD.java	71.7%	48	22
src/negocio/controllers/GestorPropuestasCursos.java	97.9%	0	1
src/negocio/entities/EstadoCurso.java	100%	0	-
src/negocio/entities/TipoCurso.java	100%	0	-

El porcentaje indicado en la imagen se corresponde a la cobertura en cada clase.

4. Maintainability (Code Smells)

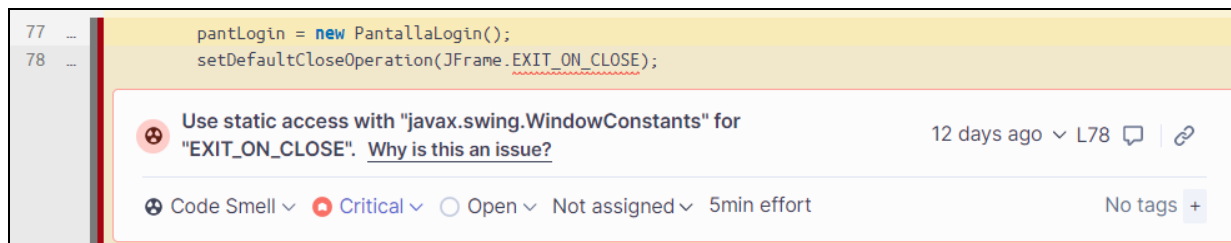
Los *"Code Smells"* son manifestaciones de fallas de diseño que pueden degradar la mantenibilidad del código. Como tal, la existencia de *"Code Smells"* parece un indicador ideal para las evaluaciones de mantenibilidad. Sin embargo, para lograr evaluaciones completas y precisas basadas en *"Code Smells"*, necesitamos saber qué tan bien reflejan los factores que afectan la mantenibilidad. Esta parte del código es confuso y difícil de mantener.

Empezaremos con los *"Code Smells"* que puedan afectar en mayor medida al funcionamiento del programa.

La prioridad es la siguiente: Blocker, Critical, Major, Minor, Info.

1- Critical

1.1- Use static access with "javax.swing.WindowConstants" for "EXIT_ON_CLOSE". 15 casos.



En aras de la claridad del código, nunca se debe acceder a los *"static"* miembros de una clase utilizando el nombre de un tipo derivado. *"base"* Hacerlo es confuso y podría crear la ilusión de que existen dos miembros estáticos diferentes. Solución:

Importar "javax.swing.WindowConstants" y Sustituir " JFrame .EXIT_ON_CLOSE" por "WindowConstants.EXIT_ON_CLOSE".

1.2- Empty method.

Añadir al método vacío: "throw new UnsupportedOperationException();" "

1.3- Make "x" transient or serializable.

```
private List<Centro> centros;
```

Make "centros" transient or serializable. Why is this an issue? 6 days ago ▾ L51 | |

Code Smell ▾ Critical ▾ Open ▾ Not assigned ▾ 30min effort No tags +

```
private List<CursoPropio> cursos;
```

Make "cursos" transient or serializable. Why is this an issue? 6 days ago ▾ L52 | |

Code Smell ▾ Critical ▾ Open ▾ Not assigned ▾ 30min effort No tags +

```
private List<Profesor> profesores;
```

Make "profesores" transient or serializable. Why is this an issue? 6 days ago ▾ L53 | |

Code Smell ▾ Critical ▾ Open ▾ Not assigned ▾ 30min effort No tags +

Los campos de una clase serializable deben ser serializables o trascendentes incluso si la clase nunca se serializa o deserializa explícitamente. Se considera un error crítico ya que con miembros de datos no transitorios y no serializables podría causar fallas en el programa y abrir la puerta a atacantes.

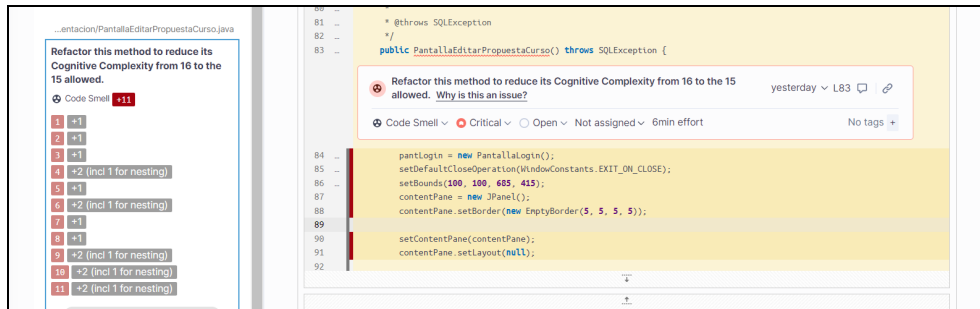
```
public void actionPerformed(ActionEvent e) {
```

Add a nested comment explaining why this method is empty, throw an UnsupportedOperationException or complete the implementation. Why is this an issue? |

Code Smell ▾ Critical ▾ Open ▾ Not assigned ▾ 5min effort

```
}
```

Para solucionarlo haremos varias clases serializables (*import java.io.Serializable + implements*



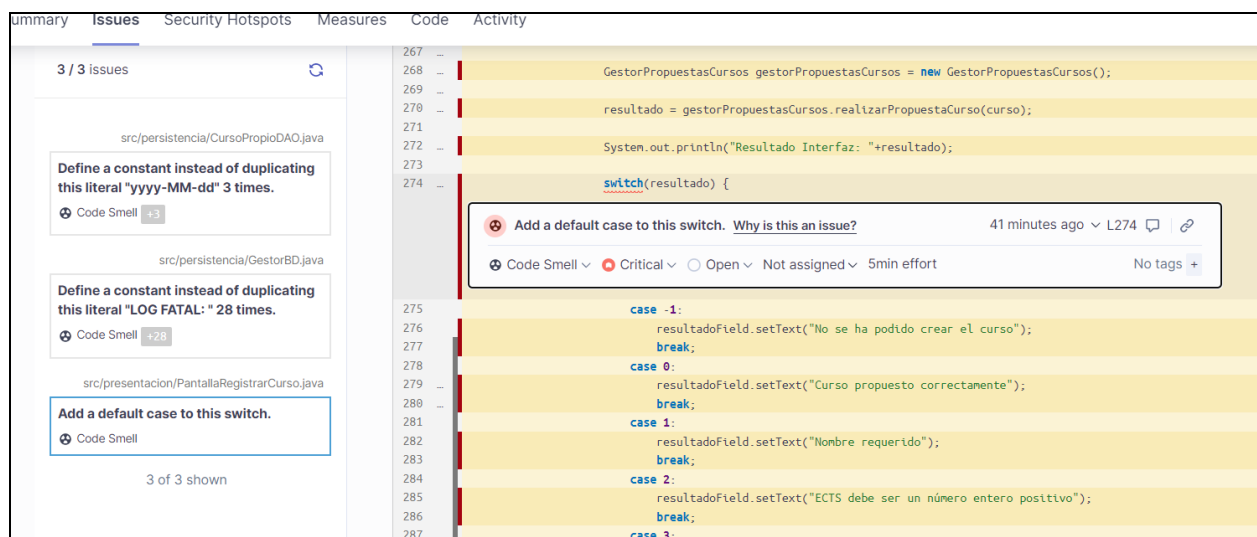
Serializable) y en caso de haya variables que no necesiten ser serializadas usaremos el modificador “transient”.

1.4- Reducir complejidad cognitiva.

El siguiente ejemplo, muestra uno de los casos en los que el código presenta una capacidad cognitiva demasiado alta. Esto se produce debido a la presencia de operaciones binarias y condiciones anidadas que aumentan la complejidad cognitiva del programa.

Para solucionarlo, delegamos el trabajo de algunas operaciones binarias a métodos que se encuentren fuera de la clase afectada.

1.5-switch sin “default case”.



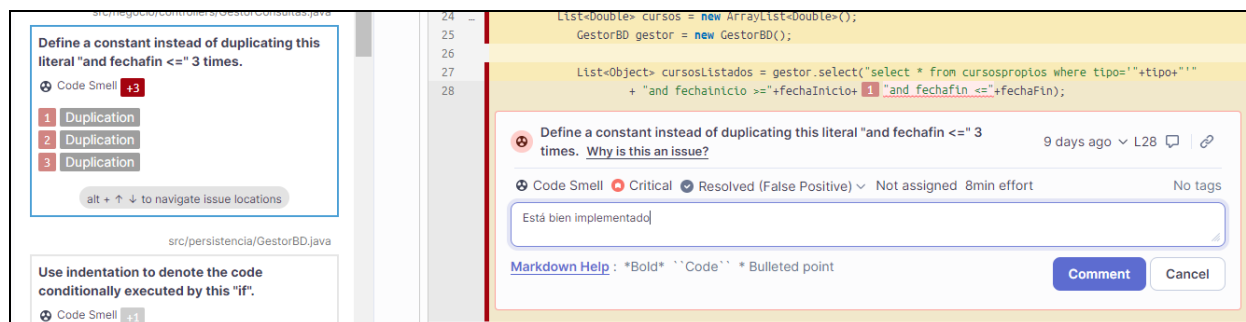
Simplemente hace falta añadir el caso “default”.

2- Mayor

La mayoría de avisos de “code smell” major eran de propósito general, como por ejemplo:

- Eliminar asignaciones inútiles.
- Implementar un log para la gestión de mensajes de error o informativos.
- Devolver una colección de datos vacía en lugar de *null*.
- Uso de constructores con menos de 5 parámetros
- Eliminación de variables no utilizadas en clases sin implementar

Algunos “code smell” como el siguiente caso, fueron considerados falsos positivos, por lo que fueron resueltos sin necesidad de modificar el código.



Aquí, sonar detecta que hay una duplicación en el código cuando realmente no es necesario considerar usar una variable para el String: “and fechafin <=” ya que en su lugar estaríamos usando un duplicado de la variable establecida.

3- Minor

Entre los “code smell” de tipo minor hay que resolver errores casi innecesarios aunque no debemos ignorar :

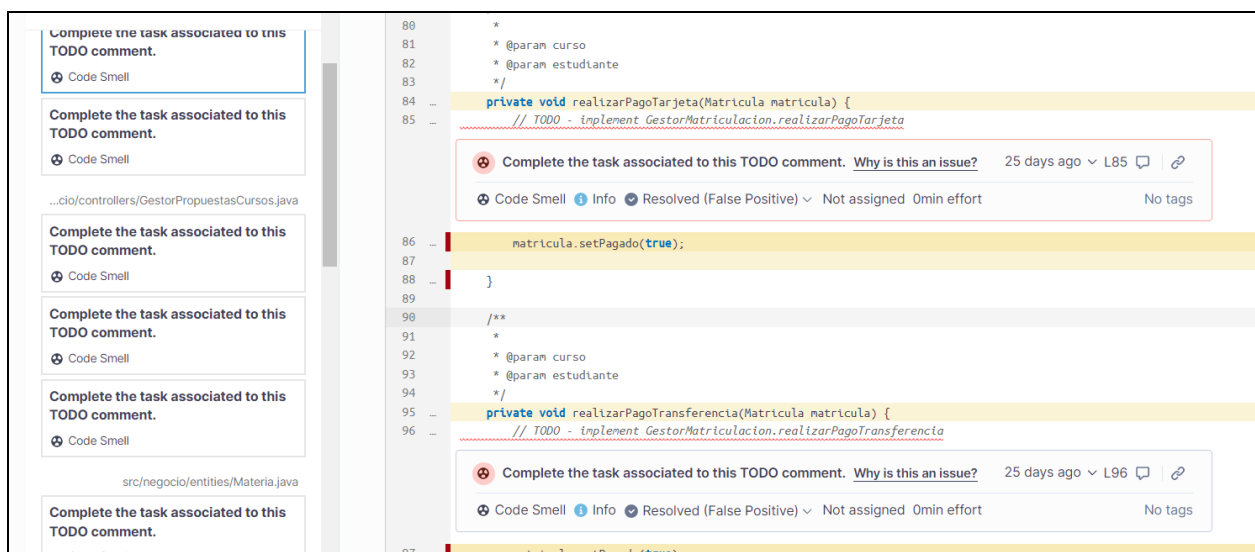
Descartar los “imports” de excepciones o clases no utilizadas. Como es el caso de:

```
import java.text.ParseException; no se utiliza
import javax.swing.JList; no se utiliza
import java.awt.Window.Type; no se utiliza
```


Debido a la baja importancia de esta sección de “code smell” (afectan en una muy leve medida al código) muchos de ellos los hemos considerado como falsos positivos. Es el caso de: code smell por iniciar por mayúscula o asignar a una variable booleana como true en vez de poner solamente true.

4- Info

Se han declarado como falsos positivos las etiquetas TODO que se usan comúnmente para marcar lugares donde se requiere más código, pero que el desarrollador desea implementar más adelante. En nuestro caso, se han revisado y se ha añadido el código necesario por lo que éstas notificaciones han sido descartadas.

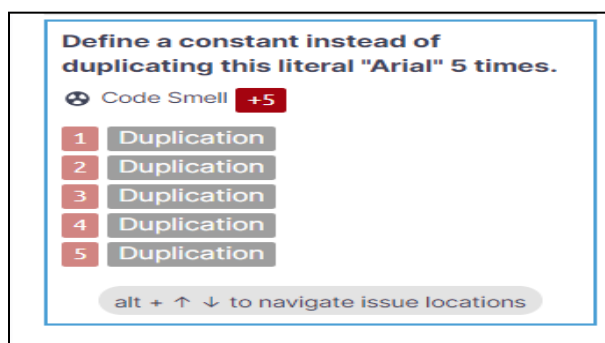


5. Duplications

Este término se utiliza cuando hablamos de un código fuente que aparece más de una vez, ya sea dentro de uno o diferentes programas, de propiedad o mantenido, por la misma entidad.

La duplicación de código es generalmente considerada una señal de estilo de programación pobre, ya que un buen desarrollo está más asociado a la reutilización del mismo.

La principal desventaja es el mantenimiento del código, lo cual se convierte en una tarea mucho más costosa.



Los "code smell" engloban la mayoría de casos de duplicaciones y los más graves. Por esta razón, serán tratados posteriormente a los "code smell"

Por ejemplo en este caso definimos una variable final para establecer el tipo de letra en lugar de repetir el String "Arial" en 5 ocasiones. Además de esta forma solo tendremos que modificar una única línea del código si queremos introducir otro tipo de letra.

Hay más casos similares a este, pero al ser la solución del mismo tipo, consideramos redundante añadir estos sucesos al plan de mantenimiento. Ejemplo: "patronFecha" o "logFatal".

6. Security Review (Security Hotspots)

Un “Security Hotspots” destaca un fragmento de código sensible a la seguridad que el desarrollador debe revisar.

Parte de los errores de esta sección estaban relacionados con el problema de seguridad y los saltamos por los motivos ya comentados con anterioridad. ⁽¹⁾

```
Lista <Objeto> t = ( Lista <Objeto> ) cursosListados .get( i );
cursoPropio .setNombre( t .get( 1 ).toString());
prueba {
    cursoPropio = cursoPropioDAO .seleccionarCurso(( int ) t .get( 0 ));
} captura (Excepción e ) {
    mi .imprimirStackTrace();
}
//habria que saber como poner un precio a la matricula, en este caso, he puesto 1250 euros
//habria que poner cursos.add(cursoPropio.getPrecioMatricula)
cursos .add( 1250.99 );
}
```

¿Por qué se considera una mala práctica la excepción.printStackTrace()?

Este fallo es un consejo de buen uso. Realmente no pasa nada por ponerlo, pero una manera mas correcta seria, por ejemplo usar un “**System.out.println(e.toString());**” para mostrar el error sin comprometer información. Si dejas esa printStackTrace, en un aplicación que está en producción, la salida será redirigida al usuario y un seguimiento de la pila nunca debe ser visible para los usuarios finales (por motivos de seguridad y experiencia del usuario).

Sustituyendo printStackTrace por esta línea de código solventamos el aviso: **System.out.println(e.toString());**

*Hemos sustituido esta línea de código por un log de errores pero el funcionamiento es el mismo.

Análisis final

★ **TrabajoISO2** PUBLIC

✓ Passed

Last analysis: 12/29/2022, 3:34 AM • 4k Lines of Code • Java, XML

A

0Bugs

A

0Vulnerabilities

A

100%Hotspots Reviewed

A

99Code Smells

14.0%Coverage

16.0%Duplications

4k Lines of Code ? Version 0.0.1-SNAPSHOT Last analysis 5 minutes ago

A 204f6c08

 declarar variable static en mayuscula (critic code smell)

✓ Quality Gate ?

New Code Overall Code

Passed

🔧 Reliability

0 Bugs ?

A

🔧 Maintainability

99 Code Smells ?

A

🔒 Security

0 Vulnerabilities ?

A

🔒 Security Review

0 Security Hotspots ?

A

Coverage

14.0% Coverage ?

Duplications

16.0% Duplications ?