PSG2

PROCESO SOFTWARE Y GESTIÓN II

Informe técnico - Análisis del Código Fuente y Métricas Asociadas

product owner: Carlos Müller Cejás

Carmen Mª Muñoz Pérez (Scrum Master)

Enrique Salazar Márquez

Javier Martínez Fernández

José Carlos Morales Borreguero

Rafael Ángel Jiménez Fernández

Contenido

[Métricas SonarCloud Sprint s2 3](#_Toc70515648)

[Panel de control 3](#_Toc70515649)

[Descripción y análisis de los bugs encontrados 4](#_Toc70515650)

[Descripción y análisis de los malos olores encontrados 4](#_Toc70515651)

* [Importación no utilizada 4](#_Toc70515652)
* [Importación duplicada 5](#_Toc70515653)
* [Eliminar importación innecesaria: las clases del mismo paquete siempre se importan implícitamente. 5](#_Toc70515654)
* [Sustituir la especificación de tipo en esta llamada al constructor por el operador diamante ("<>") 5](#_Toc70515655)
* [Definir una constante en lugar de duplicar un String varias veces 5](#_Toc70515656)
* [Devuelve inmediatamente una expresión en lugar de asignarla a una variable temporal 6](#_Toc70515657)
* [Eliminar el uso de "NoOpPasswordEncoder"; está obsoleto 6](#_Toc70515658)
* [Cambiar el nombre de una variable determinada que oculta el campo declarado una línea determinada 6](#_Toc70515659)
* [Renombrar el campo 6](#_Toc70515660)
* [el bloque de líneas de código comentadas debe ser eliminado 6](#_Toc70515661)
* [Añadir un constructor privado para ocultar el público implícito 7](#_Toc70515662)
* [Convertir la clase abstracta en una interfaz 7](#_Toc70515663)
* [Completar la tarea asociada a este comentario TODO 7](#_Toc70515664)
* [Definir y lanzar una excepción dedicada en lugar de utilizar una genérica 7](#_Toc70515665)
* [Eliminar campo privado no utilizado 8](#_Toc70515666)
* [Devuelve una colección vacía en lugar de null 8](#_Toc70515667)
* [Utilice isEmpty() para comprobar si la colección está vacía o no. 8](#_Toc70515668)
* [Cambie el nombre de esta variable local para que coincida con la expresión regular '^[a-z][a-zA-Z0-9]\*$' 8](#_Toc70515669)
* [Utilice assertThat(actual).hasToString(expectedString) en su lugar. 8](#_Toc70515670)
* [Elimina el modificador "público". 9](#_Toc70515671)

[Conclusión 9](#_Toc70515672)

[Métricas SonarCloud Sprint s3 9](#_Toc70515673)

[Panel de control 9](#_Toc70515674)

[Descripción y análisis de los bugs encontrados 10](#_Toc70515675)

[Descripción y análisis de los malos olores encontrados 10](#_Toc70515676)

* [Utilice las clases de la API de Java en lugar de las clases de Sun 10](#_Toc70515677)
* [Sustituir este uso de "String.class.isInstance()" por "instanceof String" 11](#_Toc70515678)

[Conclusión 11](#_Toc70515679)

### Métricas SonarCloud Sprint s2

### Panel de control

Una vez se ha terminado el Sprint S2, gracias a la plataforma Sonar Cloud somos capaces de obtener diferentes métricas que nos ayudarán a analizar el código de nuestro sistema para realizar mejoras e incrementar así la calidad de nuestro producto. A continuación, se mostrarán las métricas facilitadas por el Sonar Cloud para nuestro proyecto:

Captura de pantalla de computadora

Descripción generada automáticamente

Captura de pantalla de computadora

Descripción generada automáticamente

Captura de pantalla de computadora

Descripción generada automáticamenteCaptura de pantalla de computadora

Descripción generada automáticamenteCaptura de pantalla de computadora

Descripción generada automáticamenteCaptura de pantalla de computadora

Descripción generada automáticamente

Una vez se han visto las métricas anteriores, se procede a explicar con mayor profundidad qué significan los valores de estos atributos. Estos se pueden relacionar con ciertos aspectos del software, por lo que se explicarán según al aspecto al que afecten:

* **Fiabilidad:** la métrica asociada con este aspecto son los bugs. Estos están relacionados con la probabilidad que tiene el sistema de producir fallos inesperados. El estado “C” indica que al menos uno de los bugs es más grave. El valor de este atributo, que es 6, es aceptable, ya que no es una cifra muy alta por lo que se podrá subsanar en poco tiempo.
* **Seguridad:** las métricas asociadas con este aspecto son las vulnerabilidades y los puntos críticos de seguridad. En nuestro caso nuestro código posee 14 vulnerabilidades con un estado de tipo “D”, lo que significa que al menos una de estas es crítica. Por otro lado, existes tres puntos críticos con un estado “E”, lo que significa que menos del 30% de estos puntos han sido revisados (concretamente no se ha revisado ninguno ya que se ve un 0% en ese atributo). En este caso los valores son algo peores por lo que habría que realizar mejoras.
* **Mantenimiento:** las métricas relacionadas con este aspecto son la deuda técnica y los malos olores. En cuanto a la deuda técnica tenemos un valor de 6h con un estado “A”, lo que significa la ratio de endeudamiento técnico es menor al 5%. En cuanto a los malos olores tenemos 117 con un estado “A”, lo que significa que a pesar de ser un número alto estos malos olores se solucionan relativamente rápido ya que no hay ninguno grave. Por lo tanto, el aspecto del mantenimiento es bastante bueno.
* **Cobertura:** las métricas relacionadas con este aspecto son la cobertura y los tests unitarios. El atributo cobertura representa el porcentaje de líneas de código cubiertas por los tests, en nuestro caso es un 55.5%. Este valor no está mal, pero debería cubrir más líneas de código para una mejor calidad del sistema, ya que ayudaría a conseguir un buen funcionamiento del producto. El atributo de los tests unitarios muestra los tests de ese tipo que hay en el sistema, siendo 49 en nuestro caso
* **Duplicaciones:** las métricas asociadas con este aspecto son los atributos duplicaciones y bloques duplicados. En nuestro caso ambos valores son cero, siendo algo muy positivo.

Finalmente, hay que destacar que también aparece una métrica que puede ser de interés, que es el número de líneas de código en el proyecto. En nuestro caso tenemos entorno a 2.200 líneas de código.

### Descripción y análisis de los bugs encontrados

Como ya se ha mencionado en el apartado anterior, Sonar Cloud ha detectado 6 bugs en nuestro sistema. En nuestro caso todos se deben a la misma situación. A continuación, se mostrará una captura de la información proporcionada por Sonar Cloud:

Una captura de pantalla de una computadora

Descripción generada automáticamente

Por lo tanto, lo que está ocurriendo es que en vez de comparar dos instancias de java con el método equals, se está comparando, utilizando “==” o “!=”, lo que significa que no se está comparando el valor real, sino las localizaciones en la memoria. Este tipo de bug es muy sencillo de solventar, ya que solo habría que cambiar esos símbolos por el método equals().

### Descripción y análisis de los malos olores encontrados

Como ya se ha mencionado en el apartado anterior, Sonar Cloud ha detectado 117 malos olores en nuestro sistema. Se han encontrado diferentes malos olores, que se analizarán a continuación.

### Importación no utilizada

Este mal olor es el que más aparece en nuestro código. Este se produce cuando se importa algún artefacto que finalmente no ha sido necesario ya que no se ha utilizado en esa clase concretamente.

Esto se debe a que cuando un desarrollador está programando puede importar algo que en ese momento esté utilizando, pero que al final elimine aquello para lo que era necesario. También puede ser que lo haya añadido manualmente con el pensamiento de usarlo, pero al final no ha sido así.

Este mal olor no es muy grave, aunque puede ocasionar problemas de legibilidad del código, ya que puede causar confusión en los desarrolladores que trabajen en el código. Aún así, es un mal olor fácil de solventar, ya que solo hace falta eliminar las importaciones, algo muy sencillo de realizar con el Eclipse de forma rápida; además es fácil acostumbrase a revisar que esto no suceda ya que existe una refactorización automática para ello.

### Importación duplicada

Este mal olor es el que más aparece en nuestro código. Este se produce cuando se importa algún artefacto más de una vez en una misma clase.

Esto se debe a que un desarrollador puede haber añadido a mano la importación sin darse cuenta de que ya estaba importada. Esto también se puede dar porque se hayan dado conflictos a la hora de fusionar dos ramas, y ambas hayan añadido esa importación y el desarrollador no se ha dado cuenta de eliminar una.

Este mal olor no es muy grave, ya que bastaría con eliminar las réplicas.

### Eliminar importación innecesaria: las clases del mismo paquete siempre se importan implícitamente.

Este mal olor se produce cuando se importa la clase misma sobre la que se está trabajando.

Esto se debe a que un desarrollador puede haber añadido manualmente esa importación sin percatarse de que está trabajando con esa misma.

Este mal olor no es nada grave ya que no da lugar a malentendidos y bastaría con eliminar las esas importaciones.

### Sustituir la especificación de tipo en esta llamada al constructor por el operador diamante ("<>")

Este mal olor se produce cuando al declarar el tipo de un Collection se indica su tipo tanto en el constructor como en la declaración.

Así, en lugar de tener que declarar el tipo de un Collection tanto en su declaración como en su constructor, ahora se puede simplificar la declaración del constructor con <>, y el compilador inferirá el tipo. Esto se realiza para reducir el uso de palabras innecesarias del código genérico.

Este mal olor no es grave, ya que es muy fácil de solucionar y no tiene graves implicaciones. Por ello, simplemente habrá que eliminar aquellos tipos innecesarios que hayan sido añadidos en situaciones de este tipo.

### Definir una constante en lugar de duplicar un String varias veces

Este mal olor se produce cuando se utiliza el mismo String en distintos sitios dentro de una misma clase en el código. Esto puede dar lugar a errores en la refactorización de código, ya que habría que actualizar todos los String sin excepción.

Para evitar esto, se pueden utilizar constantes, y que se pueden utilizar las veces necesarias en el código, pero si es necesario un cambio en su valor sólo harña falta hacer una modificación.

Este error es más grave que los anteriores, ya que facilita la presencia de errores en el código si no se tiene cuidado. Aún así, su solución es sencilla, ya que sólo habría que crear una constante cuyo valor sea el String correspondiente.

### Devuelve inmediatamente una expresión en lugar de asignarla a una variable temporal

Este mal olor se produce cuando existe un método que crear una variable temporal para almacenar algo, pero inmediatamente después la devuelve o la lanza.

Este mal olor no es muy grave, ya que sus consecuencias no son muy importantes. Incluso hay desarrolladores que no están de acuerdo con esta práctica, ya que dicen que ayuda a la legibilidad del código. Aún así, el nombre del método debería ser suficiente para entenderlo.

Así eso sería fácil de solucionar, ya que únicamente habrá que devolver directamente los valores pertinentes, eliminando así esos innecesarios valores temporales.

### Eliminar el uso de "NoOpPasswordEncoder"; está obsoleto

Este mal olor se produce cuando se utiliza un método que está obsoleto.

Una vez desaprobadas, las clases e interfaces y sus miembros deben evitarse, en lugar de utilizarse, heredarse o extenderse. La desaprobación es una advertencia de que la clase o la interfaz ha sido superada, y eventualmente será eliminada. El periodo de desaprobación permite hacer una transición suave para alejarse de la tecnología que envejece y que pronto se retirará.

Para solucionarlo, se deberá usar otro método que no esté obsoleto.

### Cambiar el nombre de una variable determinada que oculta el campo declarado una línea determinada

Este mal olor se produce cuando se sobrescribe o se hace sombra a una variable declara en un ámbito externo.

Así, esta situación puede afectar tanto a la legibilidad como el mantenimiento del código. Debido a esto, un desarrollador podría confundir variables y usarlas de manera errónea.

Este mal olor es grave, ya que la confusión de variables puede provocar un caos para los desarrolladores que trabajen con el código, sobre todo si no es suyo originalmente. Para solucionarlo, simplemente hay que cambiar de nombre a esas variables.

### Renombrar el campo

Este mal olor se produce cuando el atributo de una clase tiene el mismo nombre que la clase a la que pertenece.

Esta no es una buena práctica, ya que lo óptimo sería escoger un nombre más descriptivo para ese atributo, mejorando así la legibilidad.

Así, este problema no es muy grave y tiene una solución sencilla, ya que solo habría que cambiar el nombre del atributo correspondiente.

### el bloque de líneas de código comentadas debe ser eliminado

Este mal olor se produce cuando existen bloques de comentarios en el código.

Esta situación se produce por diversas razones. En muchas ocasiones un desarrollador trabaja en una idea y la comenta porque aún no funciona y no quiere subir a su rama el código con errores y al final no lo utiliza, otras veces es porque no lo va a utilizar, pero aún cree que le puede ser útil. En otras ocasiones puede ser código comentado útil pero que aún no funciona como se esperaba. Este mal olor puede llegar a ser grave, ya que puede dar lugar a grandes confusiones entre los desarrolladores ya que no sabes por qué motivo está ahí el código si no está indicado.

La manera de solucionar esto es evitar lo máximo posible comentar métodos enteros en el código fuente del sistema, ya que es fácil olvidar que están ahí.

### Añadir un constructor privado para ocultar el público implícito

Este mal olor se produce cuando se crean clases que poseen un constructor público por defecto que lo proporciona Java implícitamente pero no se añade un constructor privado.

Esta situación no es excesivamente grave, pero hay que tenerlo en cuenta ya que sirve para tener solo y exclusivamente un único objeto de una clase y que este objeto siempre sea el mismo.

Así, es fácil de solucionar, ya que habría que añadir un constructor privado a cada clase que no lo tenga.

### Convertir la clase abstracta en una interfaz

Este mal olor se produce cuando una clase abstracta sin campo directo o heredado debe convertirse en una interfaz por la función “método por defecto” de Java 8.

Este mal olor no es muy grave, ya que en ocasiones ni siquiera es necesario debido a alguna librería o por intereses del desarrollador.

La forma de solventarlo sería convertir las clases correspondiente a interfaz.

### Completar la tarea asociada a este comentario TODO

Este mal olor se produce cuando en un método el código es requerido, pero aún no se ha implementado, o el desarrollador ha olvidado eliminar el tag “TODO Auto-generated method stub”.

Realmente este mal olor es informativo por lo que no tiene gravedad y te sirve para darse cuenta si algún método no se ha completado. En nuestro caso ha sido porque no se ha eliminado el tag, ya que el método si estaba completo.

Así, sólo habría que eliminar ese comentario.

### Definir y lanzar una excepción dedicada en lugar de utilizar una genérica

Este mal olor se produce cuando en un método concreto se lanza una excepción genérica.

Este error tiene cierta importancia, ya que el uso de excepciones genéricas como Error, RuntimeException, Throwable y Exception evita que los métodos de llamada manejen las excepciones verdaderas generadas por el sistema de forma diferente a los errores generados por la aplicación.

Así, para solucionar este mal olor habría que crear unas excepciones personalizadas creadas por el desarrollador.

### Eliminar campo privado no utilizado

Este mal olor se produce cuando se declara un campo privado, pero después no se utiliza.

Esta situación no es muy grave, pero puede empeorar el mantenimiento y legibilidad del código, ya que los desarrolladores se pueden preguntar para qué está ahí esa variable.

Así sería fácil de solucionar, ya que solo habría que eliminar las variables que no se usan.

### Devuelve una colección vacía en lugar de null

Este mal olor se produce cuando un método devuelve un null.

Devolver null en lugar de un array o colección real obliga a los que llaman al método a comprobar explícitamente la nulidad, lo que los hace más complejos y menos legibles, por lo que este mal olor tiene bastante importancia.

La manera de mejorar esta situación es cambiar a aquellos métodos que devuelven directamente un null para que devuelvan una colección vacía.

### Utilice isEmpty() para comprobar si la colección está vacía o no.

Este mal olor se produce cuando se utiliza Collection.size() para comprobar si una colección está vacía o no.

Este mal olor no es muy importante, pero puede perjudicar en la eficiencia y legibilidad del código. La complejidad temporal del size() puede llegar a ser bastante más grande.

Así, para solucionar esta situación, basta con utilizar isEmpty() para comprobar si una colección está vacía.

### Cambie el nombre de esta variable local para que coincida con la expresión regular '^[a-z][a-zA-Z0-9]\*$'

Este mal olor se produce cuando se nombra una variable sin seguir los estándares establecidos.

Este error no es grave. El hecho de que el equipo use una nomenclatura común ayuda en la eficacia y legibilidad del código.

Así, simplemente habría que renombrar de forma estandarizada esas variables.

### Utilice assertThat(actual).hasToString(expectedString) en su lugar.

Este mal olor se produce cuando en lugar de utilizar el método hasToString() en los tests unitarios, se utiliza otro método que realiza lo mismo pero es menos eficiente.

Este mal olor no es muy grave. AssertJ contiene muchos métodos de aserción específicos para tipos comunes. Ambas versiones probarán las mismas cosas, pero la dedicada proporcionará un mejor mensaje de error, simplificando el proceso de depuración.

Así, para cambiarlo sólo habría que sustituir un método por otro.

Como este mal olor hay otros como “utilice assertThat(actual).isEmpty() en su lugar” o “utilice isNotZero() en su lugar”.

### Elimina el modificador "público".

Este mal olor se produce cuando se está realizando un test unitario y se añade “public” en el método.

Este mal olor es informativo, por lo que no tiene importancia. Así. las clases de prueba de JUnit5 pueden tener cualquier visibilidad menos la privada, sin embargo, se recomienda utilizar la visibilidad por defecto del paquete, que mejora la legibilidad del código.

Por ello, simplemente habría que eliminar ese modificador público para utilizar la que esté por defecto.

### Conclusión

En definitiva, nuestro proyecto tiene unas métricas satisfactorias. No se ha acumulado mucha deuda técnica. Una vez analizados los malos olores y los bugs, se ha visto que en su mayoría son sencillos de solventar, por lo que eso es muy bueno. Por otro lado, la cobertura de tests debería mejorar.

De cara al futuro, no solo habrá que solventar esos aspectos, sino que habrá que esforzarse en evitar que se produzcan, asimilando así buenas prácticas y recortando aún más la deuda técnica.

### Métricas SonarCloud Sprint s3

### Panel de control

A continuación, se mostrarán las métricas facilitadas por el Sonar Cloud una vez añadido el valor de nuevas funcionalidades implementadas en el sistema:

Captura de pantalla de computadora

Descripción generada automáticamente

Captura de pantalla de computadora

Descripción generada automáticamente

Una captura de pantalla de una computadora

Descripción generada automáticamenteUna captura de pantalla de una computadora

Descripción generada automáticamenteUna captura de pantalla de una computadora

Descripción generada automáticamente

Dado que ya han sido explicado el significado y el ámbito al que pertenecen cada uno de los atributos, se procederá a comentar la modificación que han sufrido cada uno de ellos. Hay que tener en cuenta que los aspectos anteriores no han sido mejorados, por eso casi todos los valores se verán incrementados.

* **Fiabilidad:** en esta ocasión se han encontrado 8 bugs y el estado continúa siendo “C”.
* **Seguridad:** en cuanto a las vulnerabilidades ahora hay 17 de estado “D” y los puntos críticos han aumentado a un valor de 4 con estado “E”. Ambos estados continúan igual que antes.
* **Mantenimiento:** le deuda técnica ha aumentado considerablemente, ya que ahora es de un día, aunque sigue siendo aceptable ya que posee un estado “A”. Por otro lado, actualmente hay 119 malos olores y el estado es “A” también.
* **Cobertura:** en cuanto al porcentaje de la cobertura a disminuido ligeramente a un 53.5%. Hemos realizado todos los tests que nos ha dado tiempo en relación con las nuevas funcionalidades, pero es posible que los que no se hayan llevado acabo hayan deteriorado el porcentaje.
* **Duplicaciones:** como en el análisis anterior, estas métricas continúan con un valor de 0.

Finalmente, en cuanto al número de líneas de código en el proyecto actualmente tenemos en torno a 2900 líneas.

### Descripción y análisis de los bugs encontrados

Se han encontrado 8 bugs, de los cuales 6 de ellos ya estaban presentes por lo que se han explicado con anterioridad. Así sólo se añadirán los nuevos.

De los dos nuevos bugs, uno de ellos es debido a la misma situación que los otros 6. El otro se mostrará a continuación:

Una captura de pantalla de una computadora

Descripción generada automáticamente

Este bug ha ocurrido porque se está accediendo a un valor que es posible que sea nulo. En ese caso podría saltar la excepción “NullPointerException”, pero también podría exponer información de depuración que sería útil para un atacante, o podría permitir a un atacante saltarse las medidas de seguridad. Por ello este bug es importante que se arregle.

Para arreglarlo hay diversas maneras. Una de ellas es comprobar primero si el valor de ese objeto es nulo o no, y así solo se accederá a él en caso de que no lo sea. Otra manera es hacer que ese objeto no pueda ser nulo.

### Descripción y análisis de los malos olores encontrados

Se han encontrado diversos malos olores. Aquí solo se describirán aquellos malos olores que se hayan encontrado nuevos, ya que la mayoría de los anteriores continúan aún sin subsanarse.

### Utilice las clases de la API de Java en lugar de las clases de Sun

Este mal olor se produce porque se ha importado las clases de los paquetes *sun.\** o *com.sun.\**. Estos no forman parte de la API de Java y dichas clases están casi siempre envueltas por clases de la API de Java que deberían utilizarse en su lugar.

Esta situación puede derivar en varios problemas, por lo que es algo grave. Entre ellos destaca la compatibilidad, ya que esta no se garantiza con otras versiones de Java. También puede haber problemas si se cambia a otro proveedor de Java como es OpenJDK.

Así, la solución sería utilizar clases de la API de Java para evitar estos posibles conflictos.

### Sustituir este uso de "String.class.isInstance()" por "instanceof String"

Este mal olor se produce se quiere comprobar si una variable puede ser lanzada a algún tipo de forma estática y no se utiliza *instanceof.*

El método isInstance() de java.lang.Class funciona de forma diferente y realiza la comprobación de tipos sólo en tiempo de ejecución, por lo que los tipos incompatibles no se detectarán en una fase temprana del desarrollo, lo que puede dar lugar a código muerto. En caso de instanceof String, producirá un error en tiempo de compilación en caso de tipos incompatibles. Estos son problemas destacables, por lo que tiene bastante importancia este mal olor.

Para solucionarlo habría que cambia el método que se ha usado por instanceof String.

### Conclusión

En definitiva, nuestro proyecto en este momento tiene unas métricas aceptables, pero muy mejorables. La deuda técnica ha incrementado, pero sigue siendo correcta. Una vez analizados los malos olores y los bugs, que apenas han surgido nuevos casos, por lo que es una buna noticia. Por otro lado, la cobertura de tests debería mejorar ya que ha disminuido y es algo más baja de lo esperado..

De cara al futuro, no solo habrá que solventar esos aspectos, sino que habrá que esforzarse en evitar que se produzcan, asimilando así buenas prácticas y recortando aún más la deuda técnica.