



LINT REPORT

18 DE ABRIL DE 2022

E2.07

<https://github.com/antcamgil/Acme-Toolkits>

Jaime Borrego Conde: jaiborcon@alum.us.es

Antonio Campos Gil: antcamgil@alum.us.es

Ana Conde Marrón: anaconmar@alum.us.es

Gonzalo Martínez Fernández: gonmarfer2@alum.us.es

Jaime Moscoso Bernal: jaimosber@alum.us.es

Enrique Muñoz Pérez: enrmunper@alum.us.es

Tabla de contenido

Resumen ejecutivo 2

Tabla de revisión 3

Introducción 4

Visión general..... 5

Malos olores..... 5

 Menores 5

Bugs 6

 Mayor 6

Conclusiones 7

Bibliografía 8

Resumen ejecutivo

El presente documento recopila los malos olores que el *plugin Lint* ha encontrado en la aplicación *Acme Toolkits*, con el fin de detectarlos y darles solución. En consecuencia, se logra mejorar requisitos no funcionales relacionadas con la calidad del producto, como la mantenibilidad o la legibilidad del sistema.

Tabla de revisión

<i>Número de revisión</i>	<i>Fecha</i>	<i>Descripción</i>
1	18/03/2022	Desarrollo del documento

Introducción

Este documento nace con la finalidad de aportar una descripción de los malos olores detectados en el proyecto. Un mal olor es un indicio de un problema que en el futuro podría conllevar consecuencias graves, normalmente debido a que el código no se ha programado manera ideal, puesto que ello llevaría mayor tiempo y esfuerzo. Por ende, no resolverlos genera lo que se denomina «deuda técnica». Cuanto más avance el tiempo, si no se mejora el código, el manejo del proyecto, la introducción de cambios o el entendimiento del código puede llegar a hacerse imposible.

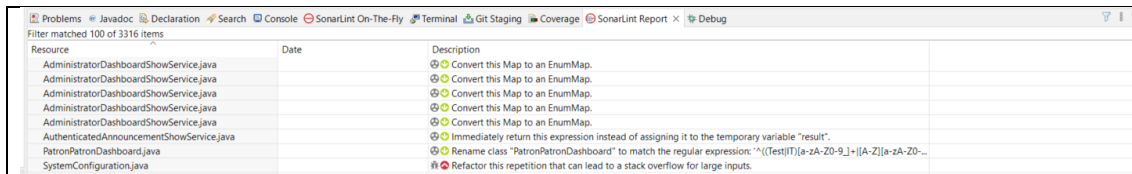
Para obtener los malos olores que contiene el proyecto *Acme-Toolkits*, se procesó el proyecto a través de la herramienta *SonarLint*. Este plugin ofrece un análisis de estos basándose en siete principios o dimensiones: no seguir los estándares de código, defectos potenciales, distribución de complejidad pobre, duplicados, exceso de comentarios, falta de test y un mal diseño.

Por lo tanto, como se ha mencionado anteriormente, mantener en un proyecto malos olores es tremendamente perjudicial. Con el fin de detectarlos, corregirlos y mejorar los resultados, a continuación, se listarán los malos olores inherentes del proyecto, explicando asimismo sendos orígenes, el porqué de su severidad y, de ser posible, se propondrá una solución. Una vez se hayan documentado, se corregirán en el proyecto.

Finalmente, una vez se profundice en los malos olores encontrados, se presentará una conclusión que permitirá estudiar y reflexionar sobre todos los conceptos y conocimientos que se han trabajado a lo largo de la realización del documento.

Visión general

Antes de analizar los malos olores pormenorizadamente, se ofrecen los resultados arrojados tras realizar un primer análisis sobre el proyecto.



Resource	Date	Description
AdministratorDashboardShowService.java		Convert this Map to an EnumMap.
AdministratorDashboardShowService.java		Convert this Map to an EnumMap.
AdministratorDashboardShowService.java		Convert this Map to an EnumMap.
AdministratorDashboardShowService.java		Convert this Map to an EnumMap.
AdministratorDashboardShowService.java		Convert this Map to an EnumMap.
AuthenticatedAnnouncementShowService.java		Convert this Map to an EnumMap.
PatronPatronDashboard.java		Immediately return this expression instead of assigning it to the temporary variable "result".
PatronPatronDashboard.java		Rename class "PatronPatronDashboard" to match the regular expression: "^(Test T a-zA-Z0-9_+ [A-Z] a-zA-Z0-9_+)\$".
SystemConfiguration.java		Refactor this repetition that can lead to a stack overflow for large inputs.

Malos olores

Menores

[Convert this Map to an EnumMap](#)

Este mal olor surge a raíz de una mala elección de tipo de *Map* para los atributos del panel de administrador. Aparece únicamente en la clase *AdministratorDashboardShowService*, específicamente en las siguientes líneas:

```
final Map<PatronageStatus, Long> mapTotalNumberOfPatronagesByStatus = new HashMap<>();  
final Map<PatronageStatus, Double> averagePatronagesBudgetByStatus= new HashMap<>();  
final Map<PatronageStatus, Double> deviationPatronagesBudgetByStatus= new HashMap<>();  
final Map<PatronageStatus, Double> minimumPatronagesBudgetByStatus= new HashMap<>();  
final Map<PatronageStatus, Double> maximumPatronagesBudgetByStatus= new HashMap<>();
```

Según propone la descripción del mal olor, sustituir *HashMap* por *EnumMap* mejoraría el rendimiento de la aplicación, pues es un tipo de dato que trabaja mejor cuando las claves de los *Map* son valores enumerados.

Su severidad es menor porque su aparición no supone un riesgo para el buen funcionamiento del sistema, únicamente empeora un poco su rendimiento.

Como solución, se propone seguir la recomendación proporcionada por *SonarLint*.

```
final Map<PatronageStatus, Long> mapTotalNumberOfPatronagesByStatus = new  
EnumMap<>(PatronageStatus.class);  
final Map<PatronageStatus, Double> averagePatronagesBudgetByStatus= new  
EnumMap<>(PatronageStatus.class);  
final Map<PatronageStatus, Double> deviationPatronagesBudgetByStatus= new  
EnumMap<>(PatronageStatus.class);  
final Map<PatronageStatus, Double> minimumPatronagesBudgetByStatus= new  
EnumMap<>(PatronageStatus.class);  
final Map<PatronageStatus, Double> maximumPatronagesBudgetByStatus= new  
EnumMap<>(PatronageStatus.class);
```

[*Immediately return this expression instead of assigning it to the temporary variable "result"*](#)

El origen de este mal olor se encuentra en la clase *AuthenticatedAnnouncementShowService*. El motivo por el que esta asignación se considera un mal olor es porque afecta a la legibilidad y a la mantenibilidad, al crear una variable que no se modifica, sino que solo se utiliza para devolver el valor asignado.

```
@Override
public Announcement findOne(final Request<Announcement> request) {
    assert request != null;
    final int announcementId = request.getModel().getInteger("id");
    final Announcement result =
        this.repository.findOneAnnouncementById(announcementId);
    return result;
}
```

De nuevo, la severidad de este mal olor es menor, porque solo afecta ligeramente a la legibilidad y a la mantenibilidad del sistema, pero en ningún caso podría llegar a bloquearlo o a constituir un obstáculo demasiado grande.

Una solución posible es devolver directamente el valor de la llamada al método, en lugar de asignarlo a una variable temporal.

```
return this.repository.findOneAnnouncementById(announcementId);
```

[*Rename class "PatronPatronDashboard" to match the regular expression "..."*](#)

Este mal olor tiene lugar en la clase *PatronPatronDashboard* (es un test). Se considera un problema que los test sobre una entidad no sigan una estructura. En el caso de este proyecto, los test se nombran como el nombre de la entidad más la palabra «Test». Sin embargo, para el caso de la entidad *PatronDashboard*, su test no sigue esta convención.

Se trata de un mal olor menor porque no está afectando al buen funcionamiento de la aplicación, aunque sí se puede considerar un error grave en cuanto a la estructura del proyecto, pues se ha ignorado la estructura acordada.

Para solucionarlo, se renombrará el nombre de la clase a *PatronDashboardTest*.

Bugs

Mayor

[*Refactor this repetition that can lead to a stack overflow for large inputs*](#)

Este bug surge en la clase *SystemConfiguration*, en el patrón para el atributo de divisas aceptadas por el sistema. El problema radica en que el motor regex de Java emplea métodos recursivos para implementar *backtracking*. Al usar el carácter «*», estas llamadas recursivas pueden generar un desbordamiento en la pila. Por lo tanto, su severidad es mayor y no se recomienda su uso.

```
@NotBlank
@Pattern(regexp = "[A-Z]{3}{,}[A-Z]{3}*" )
protected String acceptedCurrencies;
```

Para evitar conflictos mayores, se recomienda utilizar «*+» (cuantificador posesivo) en lugar del símbolo anteriormente mencionado.

Conclusiones

El estudio y la realización de este documento ha permitido una mejora tanto en el entendimiento de la importancia de los malos olores como de sus respectivas causas, aunque se esperaban resultados peores.

Se ha descubierto una gran variedad de causas que pueden empeorar el sistema, desde la mala elección de un tipo hasta una expresión regular que provoca un uso excesivo de memoria. Esto puede causar gran estupor, pues muestra la amplia gama de detalles que se deben de tener en cuenta para que el código sea óptimo.

Anteriormente se mencionó que la aparición de la deuda técnica se debía al sacrificio de la optimización del código en favor del desarrollo de código con mayor celeridad posible. Esto se puede observar fácilmente en este documento, pues algunos de los ejemplos encontrados se habrían podido evitar en su momento con un poco más de atención y cuidado.

A pesar de la existencia de malos olores, se considera que tanto el trabajo realizado en este documento como el pequeño número de malos olores en un primer análisis han sido muy satisfactorios y con un alto valor de aprendizaje por parte de todos los miembros del grupo.

Bibliografía

Intencionalmente en blanco.