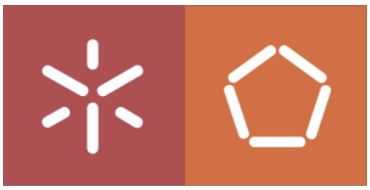


Universidade do Minho
Escola de Engenharia

Universidade do Minho
Mestrado Integrado em Engenharia Informática

ATS

Análise e Teste de Software
Trabalho Prático
Grupo 05

Janeiro 2021

Tânia Filipa Amorim Rocha
A85176

Maria Miguel Albuquerque Regueiras
A85242

Alexandra de Barros Reigada
A84584

Conteúdo

Lista de Figuras

1 Introdução

Este trabalho foi realizado no âmbito da Unidade Curricular de Análise e Teste de Software com o objectivo principal de analisar 99 projectos do ano lectivo anterior da UC de POO.

Numa primeira fase foi necessário analisar a qualidade do código das 99 aplicações através do **SonarQube**, avaliando quantitativamente e qualitativamente os seus bugs, code smells, métricas, entre outros parâmetros. Para efectuar esta leitura em "massa"será utilizado um script para abrir todos os projectos no **SonarQube**.

A segunda tarefa do trabalho diz respeito ao *refactoring* das várias aplicações tendo como objectivo analisar os diferentes tipos de smells e eliminá-los (caso seja possível). Nesta tarefa é também proposto utilizar um ficheiro em bash para efectuar o "mass refactoring"de todos os projectos.

Para a terceira fase deste trabalho foram realizados testes para dois os projetos, mais em concreto, testes unitários em **JaCoCo** ou através do sistema **EvoSuite**.

Na última fase deste trabalho prático, analisou-se o desempenho de 2 projetos de modo a avaliar a performance do software com e sem a influência dos diferentes tipos de smells, complementando com uma análise detalhada por smell.

2 Tarefa 1

Primeiramente, tivemos como objectivo analisar a qualidade do código fonte de 99 projectos da UC de POO do ano anterior. Foi necessário ter em conta que qualidade de software e qualidade de código são duas vertentes distintas em que cada uma delas está relacionada com diferentes aspectos e indicadores de qualidade.

Deve ser feita a questão, no entanto, da razão pela qual se deve medir estes aspetos de um software. De facto, existem vários motivos:

- Compreender questões relacionadas com o desenvolvimento do Software;
- Tomar decisões sobre esse desenvolvimento com base em factos e não apenas opiniões;
- Prever condições para desenvolvimentos futuros;
- Estimar o custo e o tempo de desenvolvimento;
- Para melhorar a qualidade do software em si.

Para analisar a qualidade de código fonte de uma aplicação podem sem utilizados vários recursos distintos que permitem observar diferentes métricas. O processo de avaliar a qualidade de um software procura a conformidade de requisitos funcionais e desempenho declarado pelo mesmo. Existem cinco tipos de traços principais que estão envolvidos numa analise de qualidade:

- **Confiabilidade** - Quando um software tem grande confiabilidade significa que tem uma taxa de falhas baixa. Tal é extremamente valioso para qualquer tipo de entrega tecnológica. Por essa razão, é preciso tê-la em consideração na análise de qualidade de software.
- **Testabilidade** - A testabilidade pode ser medida com base na quantidade de casos de teste que são necessários para encontrar possíveis falhas no sistema. É neste factor que entra a complexidade ciclomática.
- **Manutenção** - Este termo significa o quão fácil é modificar um produto de software, geralmente com o objetivo de corrigi-lo ou adaptá-lo de alguma forma. Contudo, não deve ser confundido com a capacidade de configurar o software.
- **Eficiência** - A métrica de eficiência refere-se ao tempo de execução das tarefas do software e o quanto é compatível com o grau de desempenho que apresenta.

  Vale a pena realçar ainda que a eficiência de um software pode ser afetada por diversos fatores externos, como a velocidade de processamento da máquina, a quantidade disponível de memória, desempenho do disco, entre outros.
- **Portabilidade** - Diz respeito à capacidade do código-fonte do produto ser utilizado noutras plataformas. É algo que envolve adaptar uma solução para diferentes organizações para operar em ambientes distintos com outros objectivos.
- **Usabilidade** -A facilidade com a qual o usuário consegue utilizar o software. A usabilidade é fundamental para o utilizador ter uma experiência positiva e não ter dificuldades em usar o software.
- **Reutilização** - Mede a quantidade de blocos de código que podem ser reutilizados.

Devido à quantidade de projectos em questão e ao facto de que, para ser utilizado a ferramenta de análise **SonarQube** é necessário uma determinada ordem estrutural de cada projecto para que este seja analisado. Foi, então, desenvolvido um *script* em *bash* que efectua o processamento do projeto para se adaptar ao formato de input que o software recebe. Este realiza todas as mudanças de directorias necessárias, criação dos ficheiros pom e construção do projecto para posterior introdução no **SonarQube**. Este método de resolução é considerada uma **Tarefa Extra** do enunciado e os procedimentos para a sua implementação são explicados a seguir.

2.1 Tarefa Extra 1

Como mencionado anteriormente, foi criado um *script* em *bash*, o *script* em questão incluí vários passos para estruturar os projetos. De seguida encontram-se as etapas para tal por ordem:

1. Primeiramente, exporta-se para um ficheiro todos os paths únicos onde são encontrados ficheiros *.java*, o que permite realizar uma listagem das diretorias por projeto que serão utilizadas.

```
find -name *.java -not -path "./*\ \ MACOSX/*"| uniq --check-chars 4 |sed  's/src.*/src/' | sed 's/\/\w*.java$//'| sed 's/\.\///' > output.txt
```

Figura 1: Procura de todos os paths únicos onde existem ficheiros *.java*.

2. De seguida, é feita a leitura de cada linha deste mesmo ficheiro e para cada uma são feitos todos os procedimentos necessário para a correta formatação e organização dos ficheiros do projeto para que o **SonarQube** consiga efectuar uma análise correta.

   O primeiro procedimento é criar uma directoria dentro de cada projecto com o próprio nome do projecto para onde serão movidos todos os componentes dos mesmos. Esta fase permite organizar todos os projectos de uma maneira inicial semelhante, visto que alguns projectos contém uma pasta *src* que depois originaria conflitos com o resto dos processos. Posteriormente é criada a directoria necessária ao bom funcionamento do **SonarQube**: *src/main/java*.

```
PP=$(echo "$line" | cut -c -2)
cd $PP
mkdir $PP
mv * $PP/
mkdir -p src/main/java
```

Figura 2: Criação das directorias.

3. A próxima etapa é procurar e copiar todos os ficheiros *.java* existentes no projecto para a directoria criada. Para isso são copiados todos os ficheiros e removidos todos os que não são *.java*.

```
find -name *.java -not -path "./*\_\_MACOSX/*"| sed 's/\/\w*.java$//'| sed 's/\.\///' | sort -u > ot.txt
while read -r line
do
    cp -a "$line"/. src/main/java

done < ot.txt
find src/main/java/ -type f ! -name '*.java' -delete
```

Figura 3: Cópia dos ficheiros para as directorias certas.

4. De seguida é gerado um ficheiro pom em cada projecto com o seu respectivo nome.

```
PAS=$(basename "$PWD")
echo '<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd
<modelVersion>4.0.0</modelVersion>
<groupId>groupId</groupId>
<artifactId>'$PAS'</artifactId>
<version>1.0-SNAPSHOT</version>
<properties>
<maven.compiler.source>1.8</maven.compiler.source>
<maven.compiler.target>1.8</maven.compiler.target>
</properties>
</project>
<!-- business data  -->' > pom.xml
```

Figura 4: Criação do ficheiro POM.

5. Por fim, são executados os comandos *maven* e *sonar* para concluir a formatação dos projetos:

```
mvn compile
mvn package
mvn sonar:sonar
```

Figura 5: Compilação do maven e sonar.

Após a análise é então possível observar os resultados iniciando o serviço do **SonarQube** e conectando-se ao localhost para poder ver de uma forma organizada todos os projetos no browser:

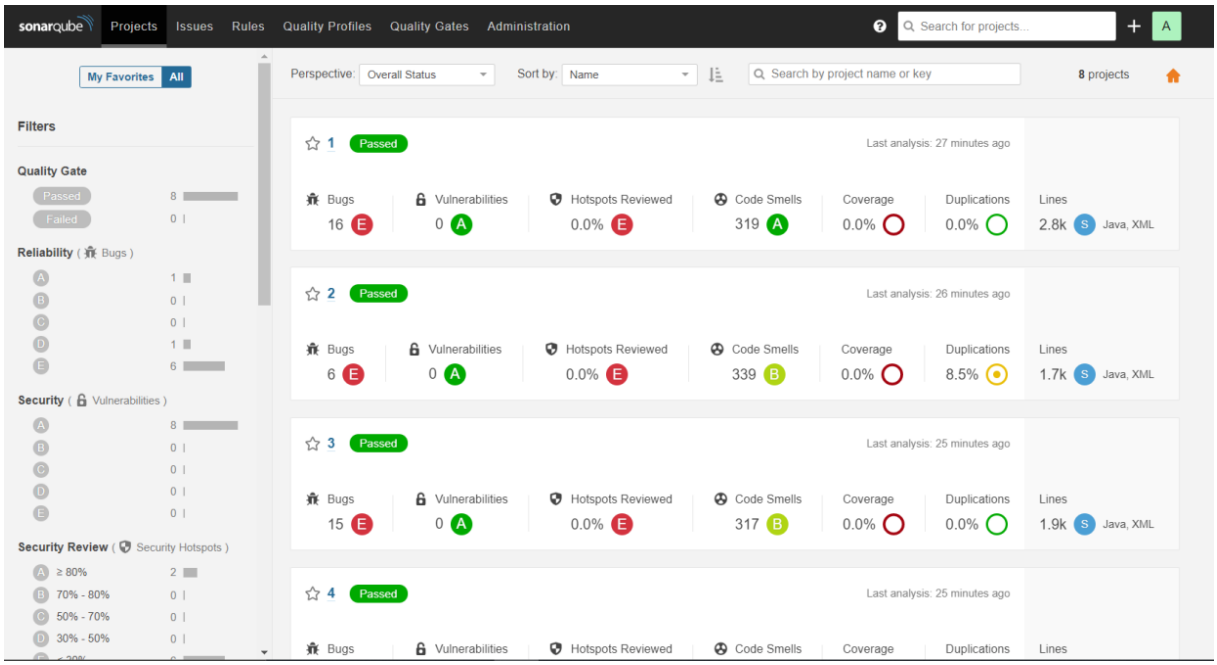


Figura 6: Resultados das análises dos projetos.

Podemos também analisar mais informações em cada projeto. O grupo tomou a decisão de analisar dois projetos em concreto a título de exemplo, o projeto '54' e '78' de formar a abordar o maior número de smells diferentes, e esta análise será apresentada de seguida.

2.2 Projecto 54

Podemos efetuar uma análise sobre as informações obtidas pelo **SonarQube** em projetos. Escolhendo o projeto '54' obtemos o que observa na figura seguinte. Da imagem podemos então concluir que o projeto '54' foi aprovado relativamente aos parâmetros de controlo do **SonarQube** e que existem 4 bugs, 437 code smells e uma percentagem de duplicados na ordem dos 3.6%.

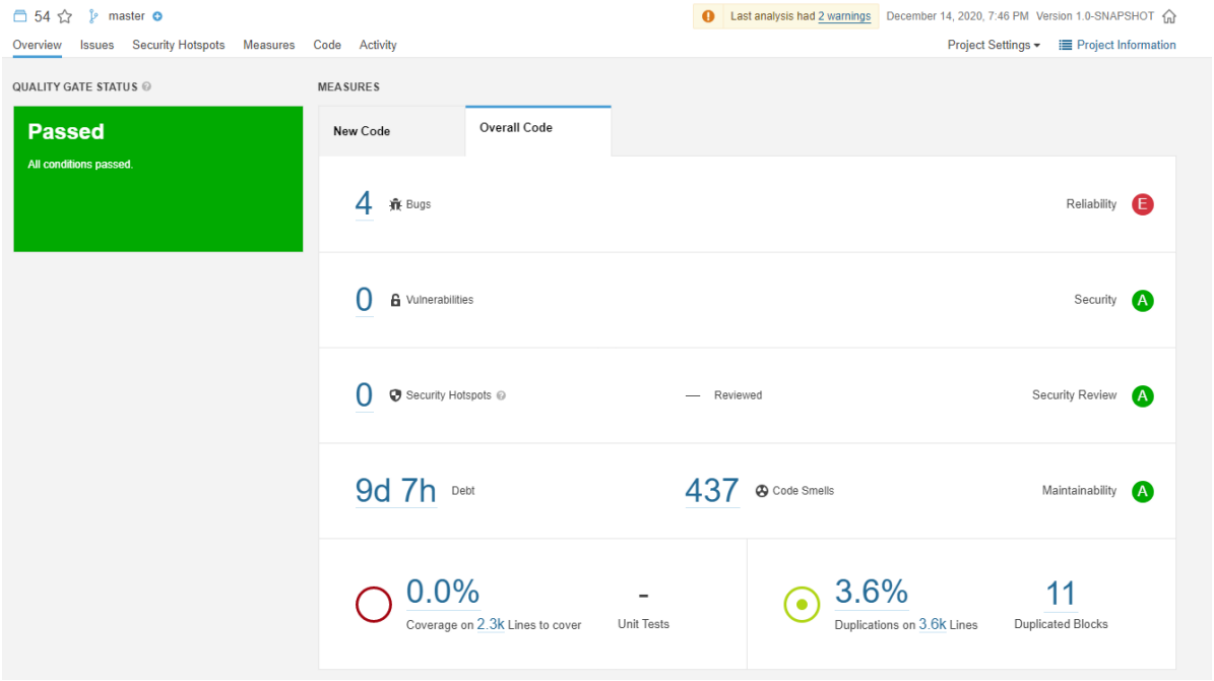


Figura 7: Projeto '54'.

Nas figuras seguintes apresentam-se exemplos de classes e a sua *technical depth* em forma de gráfico para tornar a sua visualização mais simples. Podemos observar que a classe TrazAqui.java apresenta uma *technical depth* de 1 hora e 40 minutos, algo aceitável. Contudo a sua *reliability rating* está classificada em E. Por outro lado, a classe

Nas figuras seguintes apresentam-se exemplos de classes e a sua *technical depth* em forma de gráfico para tornar a sua visualização mais simples *ViewUtilizador.java* apresenta uma *technical depth* de 1 dia e 3 horas mas uma *reliability rating* de A.

Figura 8: *Techinical Depth* e *Reliability Rating* do projeto 54 antes do refactoring.

Após uma análise detalhada dos resultados obtidos construiu-se a seguinte tabela com os valores quantitativos das diferentes métricas obtidas e, para além dessas, calcularam-se outras como: NCLOC (Non Comment Lines of Code), Functions/Classes e Statements/Functions. Estes valores foram obtidos fazendo pedidos através da API do **SonarQube** e selecionando qual o projeto e métricas que se queriam analisar.

Column1.metric	Column1.value
comment_lines	762
complexity	782
functions	493
duplicated_lines_density	3.6
code_smells	437
security_hotspots	0
sqale_rating	1.0
ncloc	3590
duplicated_lines	212
violations	441
cognitive_complexity	455
duplicated_blocks	11
bugs	4
security_rating	1.0
comment_lines_density	17.5
file_complexity	18.2

Figura 9: Métricas do projeto '54' antes do refactoring.

Através destes valores concluí-se que a sua *Cognitive Complexity* (que indica se o programa é difícil de compreender) encontra-se com o valor 455 e a *File Complexity* com uma taxa de 18.2%. A sua *Complexity* tem ainda um valor de 782.

Uma das partes mais importantes na análise efetuada foi verificar os casos de bugs no código visto ser um grande foco na correção de um programa. De seguida analisam-se alguns bugs deste projeto em concreto.

2.2.1 Bugs

Segundo a documentação da ferramenta **SonarQube**, bug é um problema que representa algo de errado no código, ou seja, um erro que ainda não foi detectado e que quando o for será, provavelmente, na pior circunstância possível. Assim, de forma a precaver possíveis vulnerabilidades a partir da análise gerada do código, podemos observar os seguintes bugs.

Neste primeiro projeto a ser analisado, existem 4 bugs, no entanto são apenas considerados 2 pois estes são semelhantes entre si.

O primeiro/segundo é o seguinte:

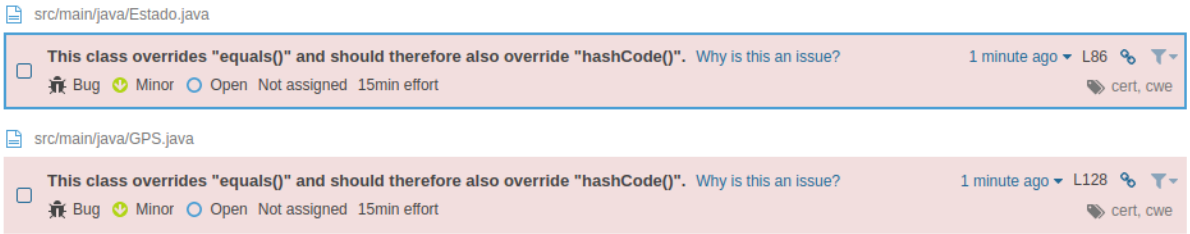


Figura 10: Bug 1 e 2.

Este bug surge quando existe o seguinte bloco de código:

```
/**
 * Metodo que determina se dois Estados sao iguais.
 * @return boolean verdadeiro caso os valores da String estado dos dois sejam iguais.
 */
public boolean equals(Object o){
```

This class overrides "equals()" and should therefore also override "hashCode()". 2 minutes ago L86
Why is this an issue?
Bug Minor Open Not assigned 15min effort cert, cwe

```
    if(o == this) return true;
    if(o == null || o.getClass() != this.getClass()) return false;
    Estado e = (Estado) o;
    return this.estado.equals(e.getEstado());
}
```

Figura 11: Bug 1 e 2 - código.

Este bug existe quando não é efetuado o *overwrite* do *hashCode()* em cada classe que der *overwrite equals()*. Se não o for feito, irá ocorrer uma violação do contrato geral do *Object.hashCode()*, o que impedirá a classe de funcionar corretamente em conjunto com todas as coleções baseadas em hash, incluindo HashMap, HashSet e Hashtable.

O terceiro/quarto bug tem estruturas semelhantes entre eles como no caso dos bugs anteriores :

src/main/java/TrazAqui.java

Use try-with-resources or close this "ObjectOutputStream" in a "finally" clause. Why is this an issue? 2 minutes ago L415
Bug Blocker Open Not assigned 5min effort cert, cwe, denial-of-service, leak

Use try-with-resources or close this "ObjectInputStream" in a "finally" clause. Why is this an issue? 2 minutes ago L425
Bug Blocker Open Not assigned 5min effort cert, cwe, denial-of-service, leak

Figura 12: Bug 3 e 5.

O bug pertence ao seguinte bloco de código:

```
/**
 * Grava um ficheiro em binario
 */
public void gravarObj() throws IOException{
    ObjectOutputStream o = new ObjectOutputStream(new FileOutputStream("Projeto/docs/objeto.txt"));
```

Use try-with-resources or close this "ObjectOutputStream" in a "finally" clause. 2 minutes ago L415
Why is this an issue?
Bug Blocker Open Not assigned 5min effort cert, cwe, denial-of-service, leak

```
    o.writeObject(this);
    o.flush();
    o.close();
}
```

Figura 13: Bug 3 e 4 - código.

Estes Bugs existem devido à criação de ficheiros que não são fechados posteriormente após a sua utilização. Esta chamada final é aconselhada que seja feita num bloco de código "finally", caso contrário uma excepção pode impedir que a chamada seja feita.

2.3 Projecto 78

Para análise e refactoring, foi escolhido também o projeto '78' por ter uma *technical debt* com valor alto (seriam necessários 15 dias para corrigir code smells, bugs e código duplicado) e um número elevado de bugs e código replicado. Este teve a seguinte análise no **SonarQube**:

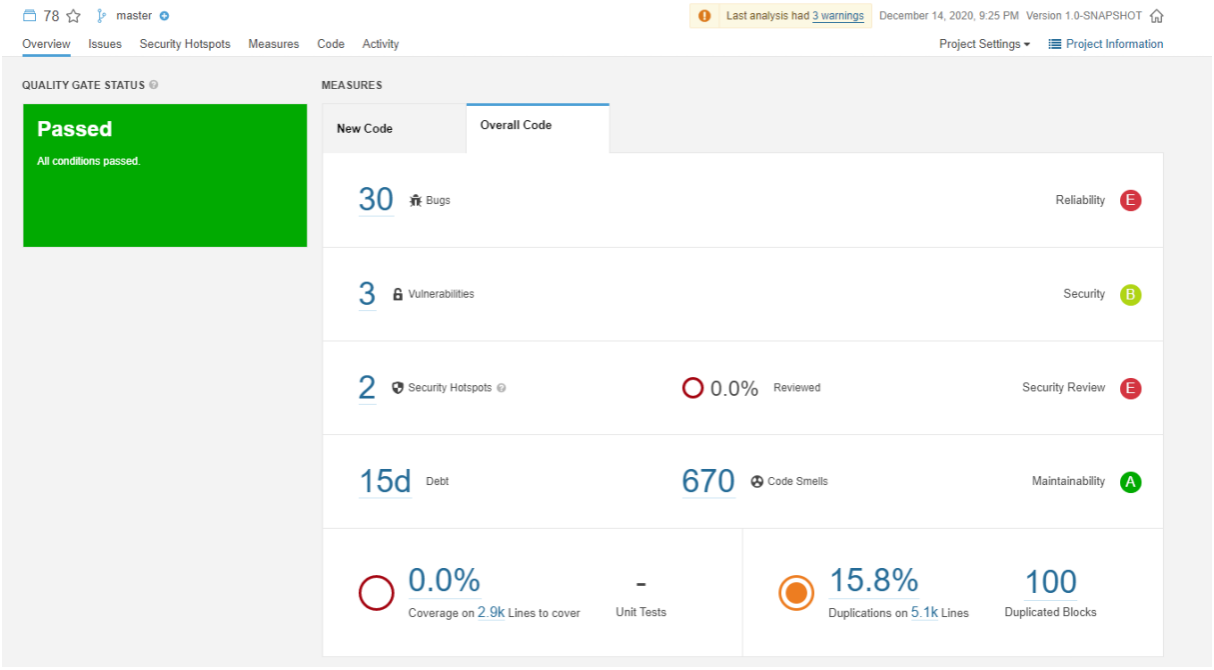


Figura 14: Projeto '78'.

O projeto '78' compilou e foi aprovado no **SonarQube**. Apresentava 670 code smells, 30 bugs e uma percentagem de duplicados na ordem dos 15.8%.

Nas figuras seguintes apresentam-se exemplos de classes e a sua *technical depth* em forma de gráfico para tornar a sua visualização mais simples como no projeto anterior. Podemos observar que a classe LeituraFicheiros.java, apesar de ter uma *technical depth* aceitável, a sua *reliability rating* está classificada em E (com a cor vermelha). No outro lado do espetro, a classe *View_Login.java* apresenta uma alta taxa de *technical depth* de 1 dia e 4 horas.

Figura 15: *Techinical Depth* e *Reliability Rating* do projeto '78' antes do refactoring.

Após uma análise detalhada dos resultados obtidos construiu-se a seguinte tabela com os valores quantitativos das diferentes métricas obtidas e, para além dessas, calcularam-se outras como: NCLOC (Non Comment Lines of Code), Functions/Classes e Statements/Functions.

Column1.metric	Column1.value
violations	703
sqale_rating	1.0
comment_lines_density	24.2
functions	654
bugs	30
security_hotspots	2
code_smells	670
comment_lines	1629
cognitive_complexity	624
duplicated_blocks	100
complexity	1158
duplicated_lines	1575
duplicated_lines_density	15.8
security_rating	2.0
file_complexity	17.5
ncloc	5115

Figura 16: Métricas do projeto '78' antes do refactoring.

Através destes valores, podemos concluir que o projeto tem uma grande quantidade de código duplicado (1575 linhas) com a densidade mencionada de 15.8%. Para além disso, a sua *Cognitive Complexity* tem valor de 624 assim como a *File Complexity* está a 17.5%.

2.3.1 Bugs

Para além dos bugs já referidos acima, este projeto apresenta outros 3 tipos de bugs. Primeiro bug:

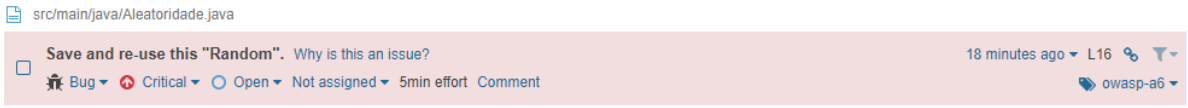


Figura 17: Bug 1.

O bug pertence ao seguinte bloco de código:

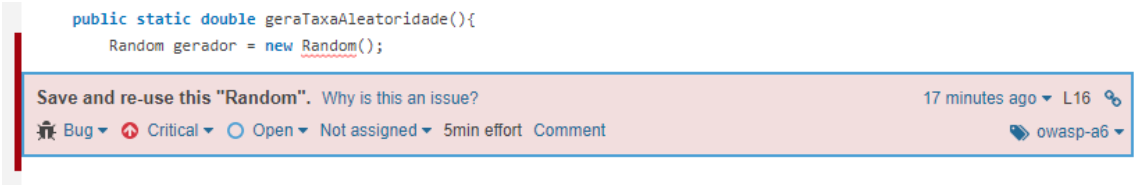


Figura 18: Bug 1 - código.

Este bug existe devido à ineficiência de criar um novo objeto *Random* sempre que é necessário. Desta forma, deve ser criada uma variável da classe de forma a não desperdiçar este objeto e reutilizá-lo.

O segundo bug:

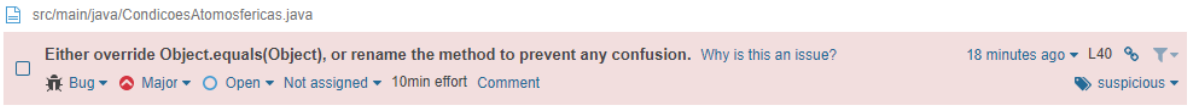


Figura 19: Bug 2.

O bug pertence ao seguinte bloco de código:

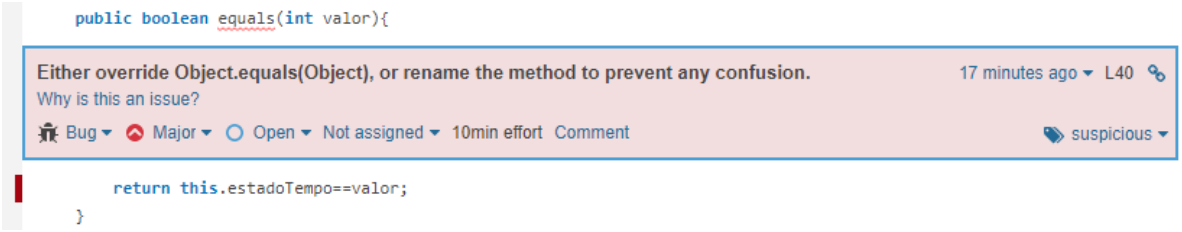


Figura 20: Bug 2 - código.

O método *equals* é um nome dum método que deve apenas ser utilizado para dar *override* do método *Object.equals(Object)*. Neste caso, este método não é utilizado com esse propósito logo deve ser alterado para dar efetivamente *override* do método *equals* ou alterado o nome.

Terceiro bug:

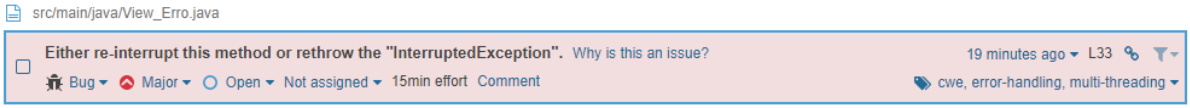


Figura 21: Bug 3.

O bug pertence ao seguinte bloco de código:

Figura 22: Bug 3 - código.

Este tipo de exceções não devem ser ignoradas, devem ser relançadas ou a thread deve ser interrompida chamando *Thread.interrupt()*. Se esta exceção não for bem tratada, corre-se o risco de se perder o facto da *thread* ter sido interrompida.

3 Tarefa 2

Esta tarefa tem como objetivo analisar os projetos novamente após o refactoring das mesmas para analisar até que ponto o refactoring de uma aplicação poder benéfico na qualidade de código na mesma.

Como primeira introdução nesta tarefa foi analisado o que é o refactoring de uma aplicação.

Refactoring de uma aplicação é uma técnica muito útil e lógica de utilização porque permite reparar alguns problema de mau funcionamento ou menor performance e ainda permite limpeza do código para uma mais fácil compreensão. Geralmente, o fator de legibilidade do código de uma determinada aplicação é um fator importante que nem sempre é tomada como prioridade com deve, isto porque aquando do desenvolvimento de uma aplicação a existência de uma data limite pode levar aos seus desenvolvedores tomarem como prioridade o acabamento da mesma e não a sua legibilidade. Daí a importância desde técnica que passar por analisar todo o software fazendo modificações a nível estrutural.

Esta técnica visa reduzir o seguintes code smells:

- Código Duplicado
- Código e métodos não utilizados
- Parâmetros/funções longos
- Classes grandes
- Outros

Geralmente o refactoring é aplicado em situações de adição de novos métodos e na análise e pesquisa de bugs para poder avaliar a estrutura interna do sistema.

3.1 Code smells

Os smells mais visíveis à vista e com complexidade menor geralmente são do tipo:

- Duplicações de código
- Código não utilizado
- Métodos demasiados extensos ou com demasiados parâmetros;
- Classes demasiado grandes.

Este tipo de smells encontram-se nos projetos em análise e podemos analisar a maneira como os diminuir.

3.2 Red code smells

Um outro tipo de smell aos quais se deve tomar mais cuidado e geralmente são menos visíveis no entanto com maior complexidade são denominados **red smells** e podem ser do tipo:

- Operações aritméticas;
- Concatenação de strings com o operador '+';
- Uso de *exceptions*;
- Uso de gets e sets;
- Criação de objetos;
- Uso de tipos de dados primitivos;
- Copias de arrays manualmente;
- Entre outros.

Este tipo de smells podem pesar consideravelmente na complexidade, performance e consumo de energia. Alguns exemplos da diferença podem ser observados a seguir:

Micro-benchmark	Version	Consumption (Ws)	Energy reduction (%)
Array copying	Manual array copy	102.8	
	System array copy	63.8	37.9
Matrix iteration	By-column iteration	53,776.8	
	By-row iteration	102.6	99.8
String handling	String concatenation (+)	4,456.1	
	String builder	271.7	93.9
Use of arithmetic	Add constant to double	5,152.5	
operations	Add constant to float	5,089.3	1.2
	Add constant to long	3,643.5	29.2
	Add constant to int	838.8	83.7
Exception	Use Exception	14,108.6	
handling	No Exception	28.1	99.8
Object field access	Accessor-based access	9,190.0	
	Direct access	1,700.8	81.4
Object creation	On-demand creation	813.1	
	Object reuse	461.6	43.2
Use of primitive	Use of object data types	3,082.3	
data types	Use of primitive data types	2,356.2	23.5

Figura 23: Diferenças no consumo de energia dependendo do método de implementação

Após uma analise neste tipo de smells procedesse ao refactoring dos mesmos através de plugins disponíveis no IDEA IntelIJ.

3.3 Casos de Refactoring

3.3.1 Projeto 54

Analisando alguns exemplos de smells anteriormente apresentados presentes no projeto 54, foi feito o refactoring dos mesmos para ser possível obter conclusões do impacto destes mesmos smells.

- Code smells com gravidade *Blocker* :

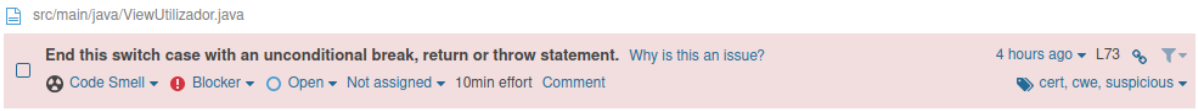


Figura 24: Code Smell 1

Este smell era causado pela falta de um *break;* no final de um case de um *switch*. Era resolvível facilmente adicionando o mesmo.

- Code smells com gravidade *Major*:

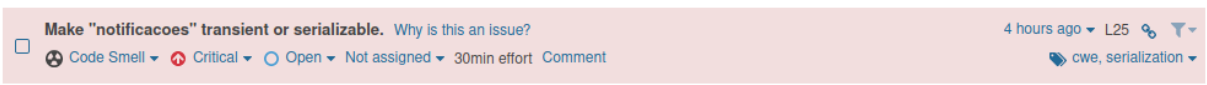


Figura 25: Code Smell 2

Smell de facil resolução, é necessário apenas necessário adicionar *transient* na declaração da variável *notificações*.

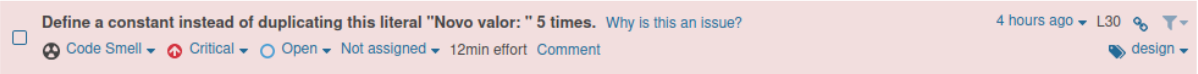


Figura 26: Code Smell 3

Este smell é apresentado varias vezes repetidas no *Sonarqube* o que indica que existe varias instâncias em que é utilizada uma "literal"ou String idêntica sem ser utilizada através de uma variável. A resolução a este tipo de smells passa por isso mesmo, ou seja, declarar uma variável com esse valor ou descrição e fazer uso dela nos pontos corretos.

- Code smells com gravidade *major*:

Figura 27: Code Smell 4

Podemos observar que, para além destes, existem code smells relativos a comentários presentes no programa daí que foram removidos para os corrigir.

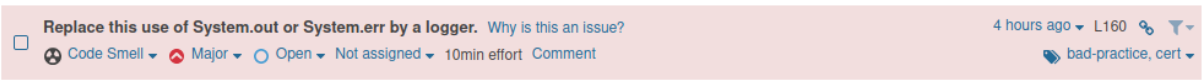


Figura 28: Code Smell 5

Um dos code smells que mais se repetia no programa era devido ao uso método *System.out.println*. Como este é usado em muitas estâncias é o causador de grande parte do número de code smells presente. Tal como sugerido pelo *Sonarqube* foi implementado um método *System.err* que visa substituir o anterior.

Para esta mesma implementação foi necessário instanciar o método denominado *Logger* que permite então o seu uso:

```
// Create a Logger
Logger logger
        = Logger.getLogger(
        ViewUtilizador.class.getName());
logger.log (Level.INFO, msg: "O que pretende fazer?");
logger.log (Level.INFO, msg: "\t1 -> Criar uma nova encomenda.");
logger.log (Level.INFO, msg: "\t2 -> Ver o histórico de Encomendas");
logger.log (Level.INFO, msg: "\t3 -> Ver os 10 clientes que mais usaram esta APP");
logger.log (Level.INFO, msg: "\t4 -> Classificar Voluntario/Empresa");
logger.log (Level.INFO, msg: "\t5 -> Definições");
logger.log (Level.INFO, msg: "\t0 -> Logout");
```

Figura 29: Code Smell 6

Quando se introduziu este método surgiu um outro smell relacionado com a concatenação de strings. Para o eliminar, foi utilizado o $\text{STRING}_FORMAT como se pode verificar na seguinte imagem.$

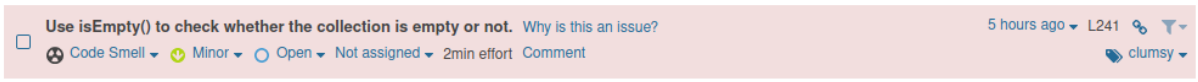


Figura 31: Code Smell 8

- Code smells com gravidade *minor*:

  Este code smell apenas sugeria para utilizar o método isEmpty() quando se pretende verificar se uma collection está vazia por isso foi efetuado isso para reduzir o mesmo.

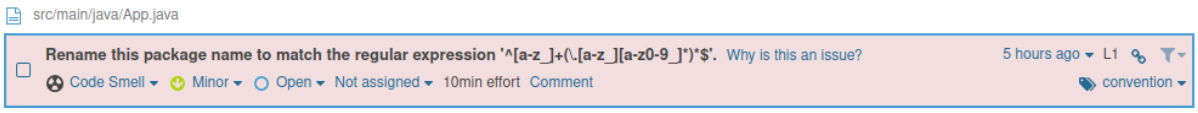


Figura 32: Code Smell 9

Este código smell deve-se à utilização de packages com o nome a começar por letra maiúscula. Foi resolvido com a renomeação dos mesmos.

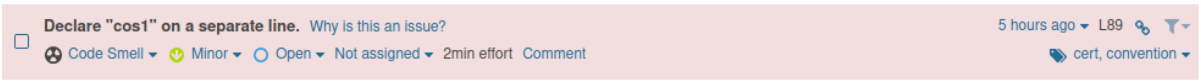


Figura 33: Code Smell 10

Um code smell que se repetia várias vezes era devido a declarações seguidas na mesma linha de código. Foi apenas necessário reformatar as mesmas para ficarem em linhas seguidas.

- Streams :

  Foi lecionado ao longo do semestre que streams podem aumentar o consumo de energia de um sistema. É geralmente feita uma análise com stremas e sem strems para escolher a opção mais eficiente. No caso em questão foi utilizado *refactoring* em alguns streams para futura análise:

```
float peso = (float) l.stream().mapToDouble(x -> x.getProduto().getPeso()).sum();
```

```
double sum = 0.0;
for (LinhaDeEncomenda x : l) {
    double v = x.getProduto ().getPeso ();
    sum += v;
}
float peso = (float) sum;
```

Figura 34: Exemplo 1 de Stream antes e depois de Refactoring

3.3.2 Projecto 78

Para além dos smells referidos acima no projeto 54, é possível encontrar diferentes no projeto 78.

```
return this.model.topNClientesMaisEncomendaram( n: 10)
                 .stream().map(IUtilizador::getId)
                 .collect(Collectors.toList());
```

```
List<String> list = new ArrayList<> ();
for (IUtilizador iUtilizador : this.model.topNClientesMaisEncomendaram ( n: 10)) {
    String id = iUtilizador.getId ();
    list.add (id);
}
return list;
```

Figura 35: Exemplo 2 de Stream antes e depois de Refactoring

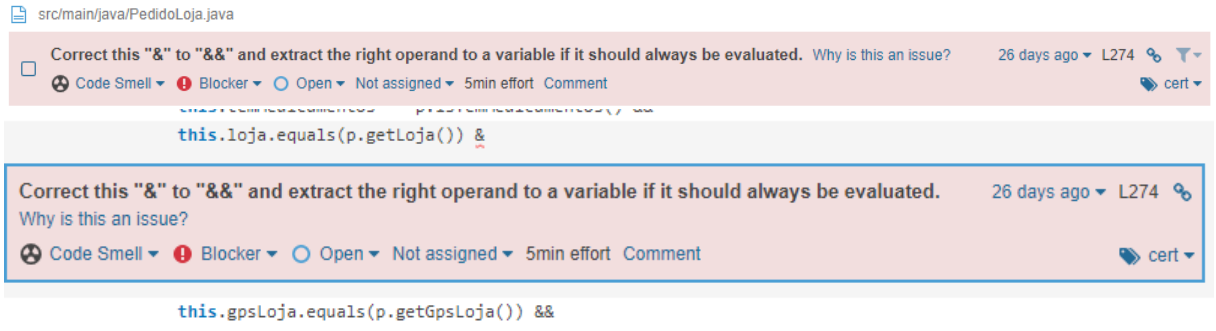


Figura 36: Code Smell 11

Este smell surge quando o operando da direita não está a ser avaliado o que implica substituir o operando por .

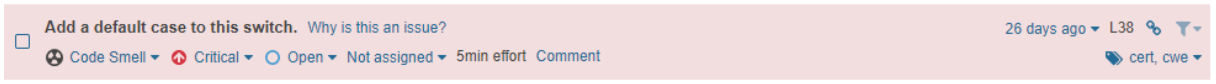


Figura 37: Code Smell 12

Este smell refere que é necessário adicionar um caso por defeito ao *switch* de forma a que, caso não seja validada nenhum caso do *switch*, então entrará no caso *default*.

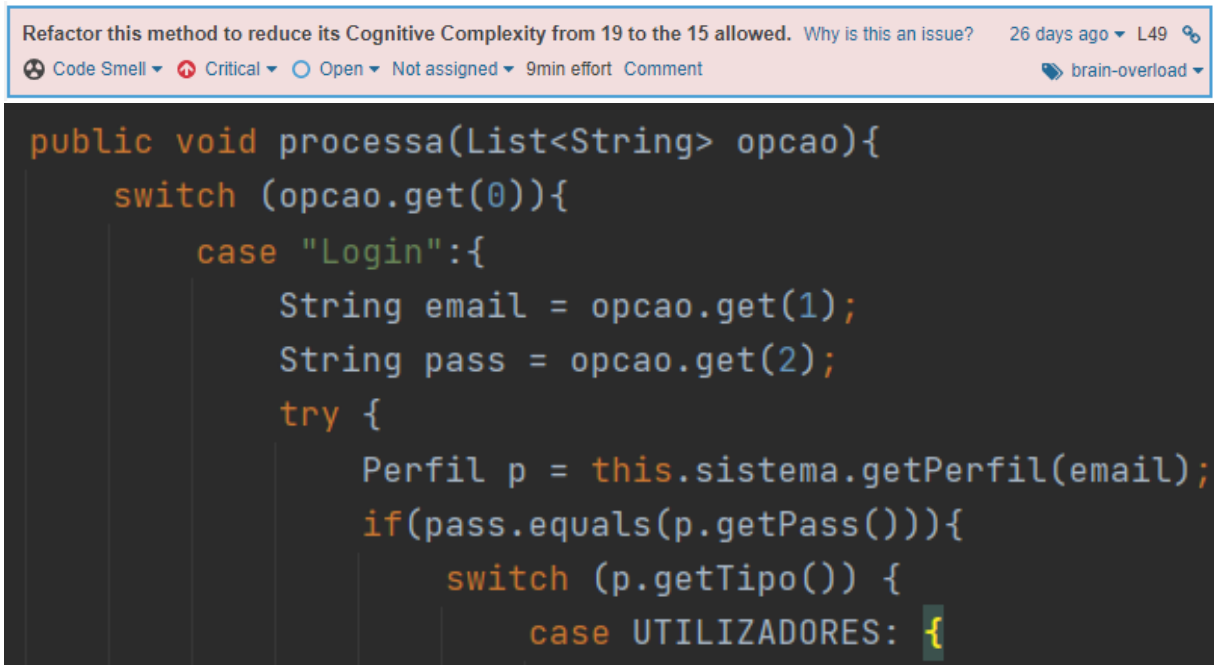


Figura 38: Code Smell 13

Este smell refere que deve-se diminuir a complexidade de um método de 19 para 15. Isto acontece quando temos um método onde existe um *switch* dentro dum *if and else* que está, por sua vez, dentro de um *switch*. De forma a eliminar este smell, deve-se dividir o método em vários. Assim, foi criado um método caseLogin que é chamado dentro do *switch case* e por sua vez, dentro deste existe o *id and else* e ainda um *switch* diminuindo asssim a complexidade do método.

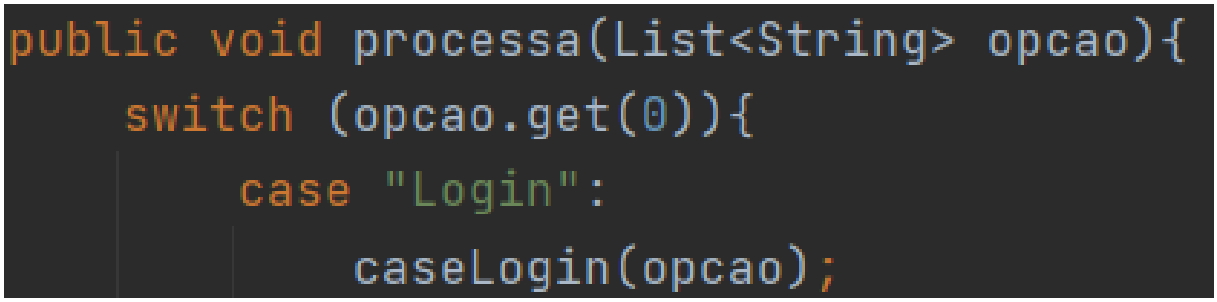


Figura 39: Code Smell 14

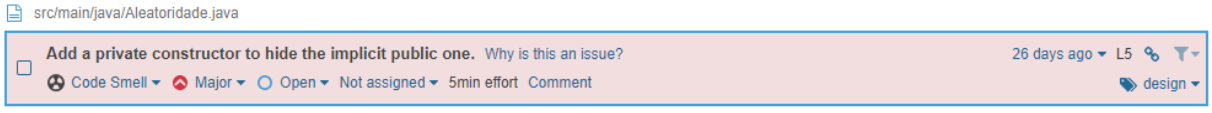


Figura 40: Code Smell 15

Para eliminar este smell é necessário criar um construtor privado à semelhança do público criado.

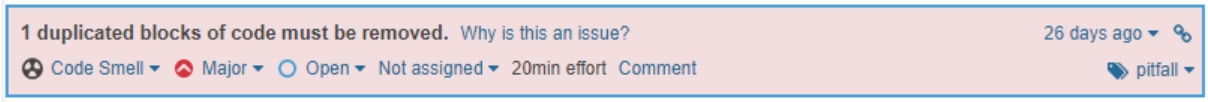


Figura 41: Code Smell 16

Este code smell ocorre quando existe código duplicado na classe. Para o eliminar, é necessário procurar qual o código que está duplicado e solucionar uma forma de não o duplicar. Como por exemplo, criar constantes, variáveis da classe, etc.

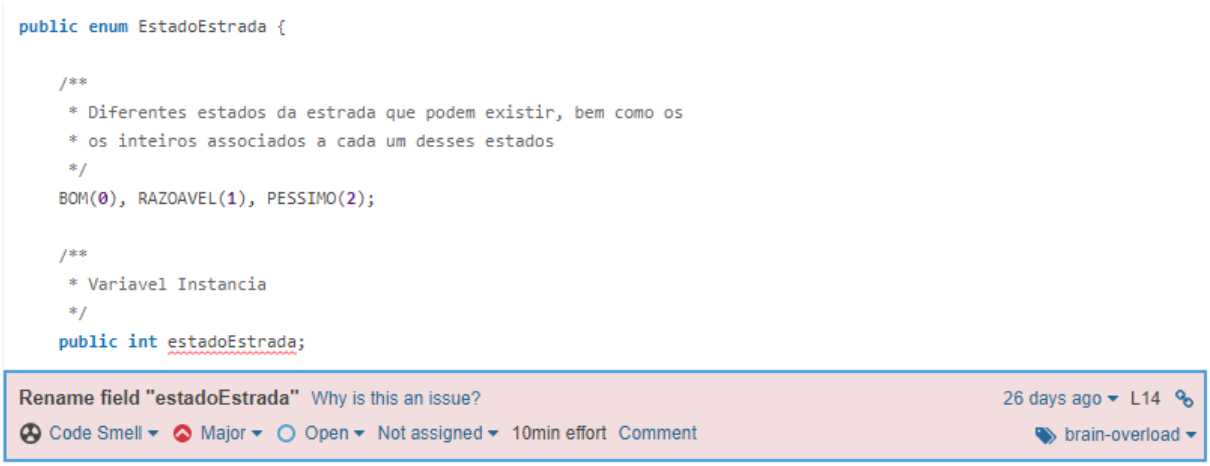


Figura 42: Code Smell 17

Este smell ocorre quando uma variável tem o mesmo nome que a classe visto que, não é uma boa prática uma classe ter variáveis com o mesmo nome que a classe. Apenas é necessário alterar o nome.

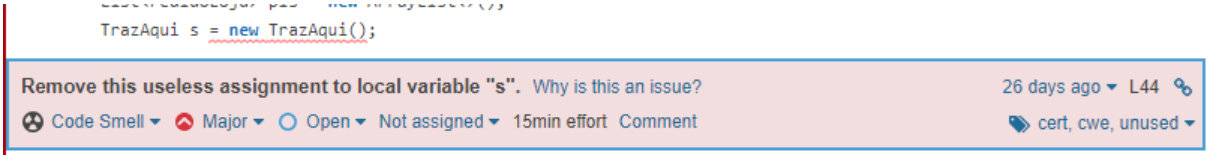


Figura 43: Code Smell 18

Este smell surge quando é criada uma variável que não é utilizada no método ou na classe e portanto, deve-se remover.

Figura 44: Code Smell 19

Este smell ocorre quando um método de uma classe, que é herdada de uma superclasse, é reescrito. Deve-se então adicionar então *@Override* acima do método.

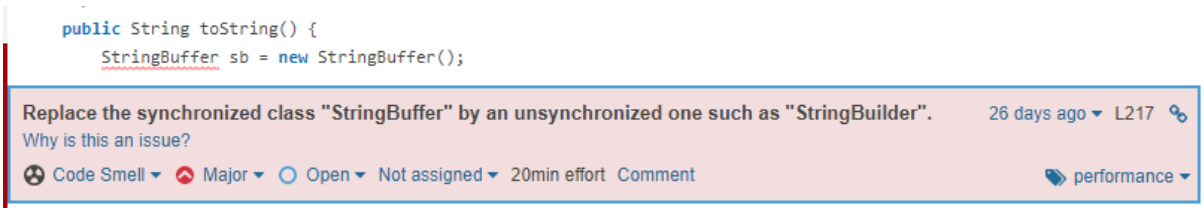


Figura 45: Code Smell 20

A utilização de StringBuffer tem um impacto negativo na performance e então, nessa medida é ´favorável a utilização de StringBuilder.

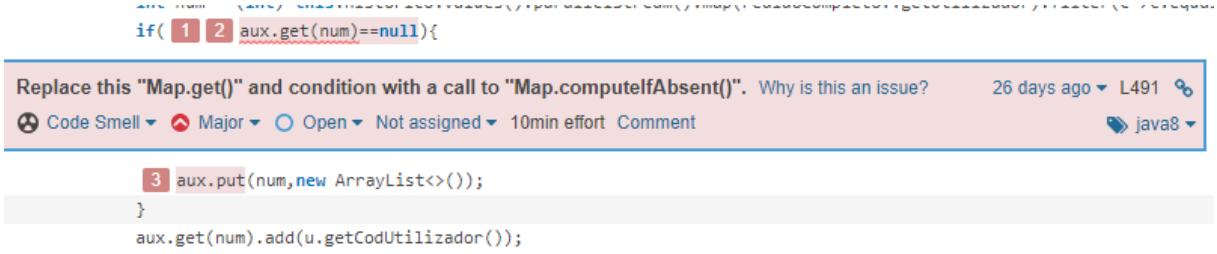


Figura 46: Code Smell 21

Quando se pretende inserir um valor numa estrutura Map, é possível testar se o valor já existe e inserir caso não existe apenas com uma operação utilizando Map.computeIfAbsent().

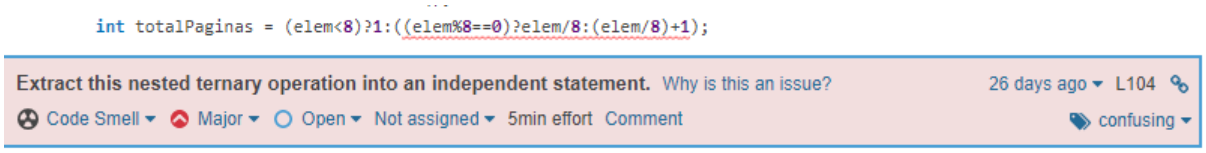


Figura 47: Code Smell 22

Este smell ocorre quando existe uma expressão demasiado complexa e difícil para alguém entender se não estiver "por dentro"do código produzido. Para simplificar, foi criada uma varíavel com esta expressão e substituída na operação.

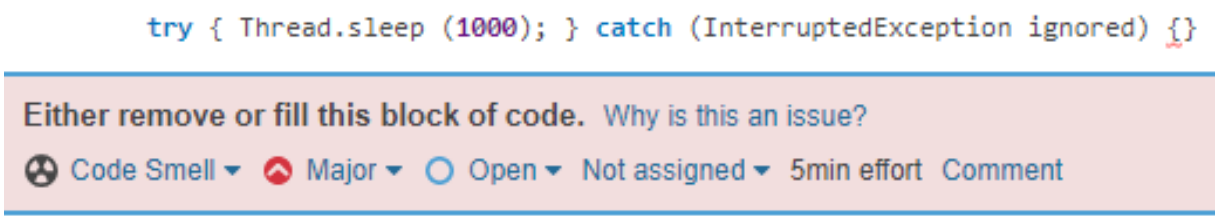


Figura 48: Code Smell 23

Quando um bloco de código não produz resultado, então deve ser eliminado ou adicionadas operações. Neste caso, era possível imprimir uma mensagem de erro caso apanhasse a exceção.

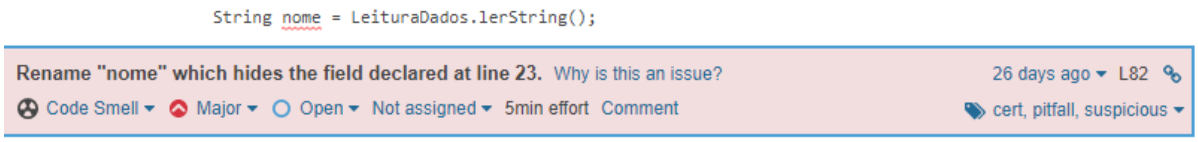


Figura 49: Code Smell 24

Variáveis criadas em métodos não devem ter o mesmo nome que variáveis da classe.

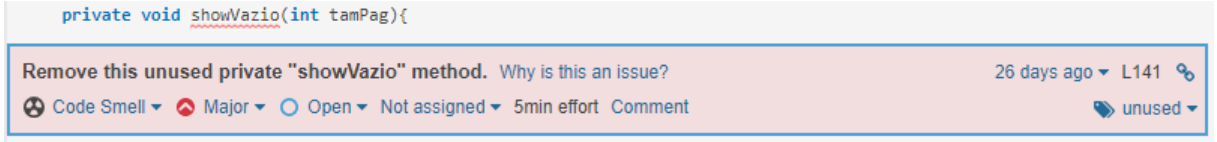


Figura 50: Code Smell 25

Este smell ocorre quando um método não é utilizado em lado nenhum no código e portanto, deve ser eliminado.

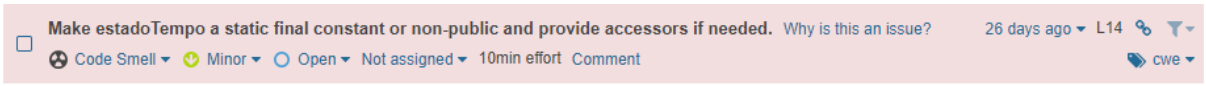


Figura 51: Code Smell 26

As variáveis públicas não respeitam o encapsulamento e portanto, devem tornar-se por exemplo, privadas e devem ser proporcionados os métodos get e set.

Figura 52: Code Smell 27

Este smell ocorre quando, por exemplo dentro de um switch case tem demasiado código que reduz a visibilidade das variáveis e portanto deve ser repartido por alguns métodos.

Figura 53: Code Smell 28

Utilizar métodos como toLowerCase() ou toUpperCase() é ineficiente porque requer a criação de objetos do tipo String temporários. Assim, deve-se utilizar um equalsIgnoreCase() de forma a evitar esta situação.

Figura 54: Code Smell 29

Quando uma exceção é a subclasse de outra, deve-se utilizar apenas a superclasse desta, caso contrário estaríamos a ser redundantes.

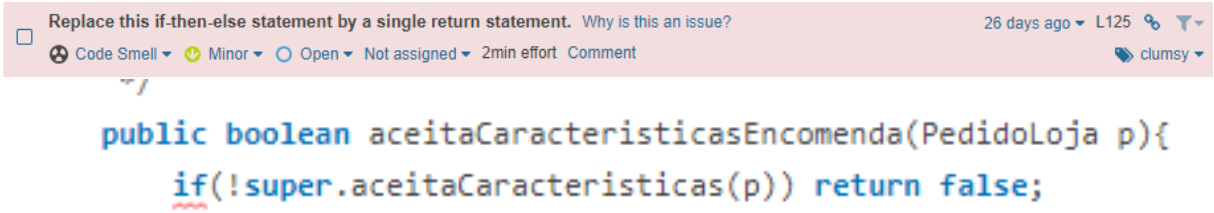


Figura 55: Code Smell 30

Quando se trata de uma expressão com valor verdadeiro ou falso, então deve se retornar essa mesma expressão (negada ou não) no caso de ser retornado um booleano.

3.4 Resultados Finais

Após o refactoring dos projetos, realizou-se novamente uma análise no **SonarQube** e a tabela das métricas finais e obtiveram-se os seguintes resultados:

- **Projeto 54**: neste projeto, concluí-se que todas as categorias ficaram classificadas com A.
  - 0 bugs;
  - 0 vulnerabilidades;
  - 1 hora e 20 minutos de *technical depth*;
  - 4 code smells;
  - 3.5% de linhas duplicadas;
  - 9 blocos de código duplicados;

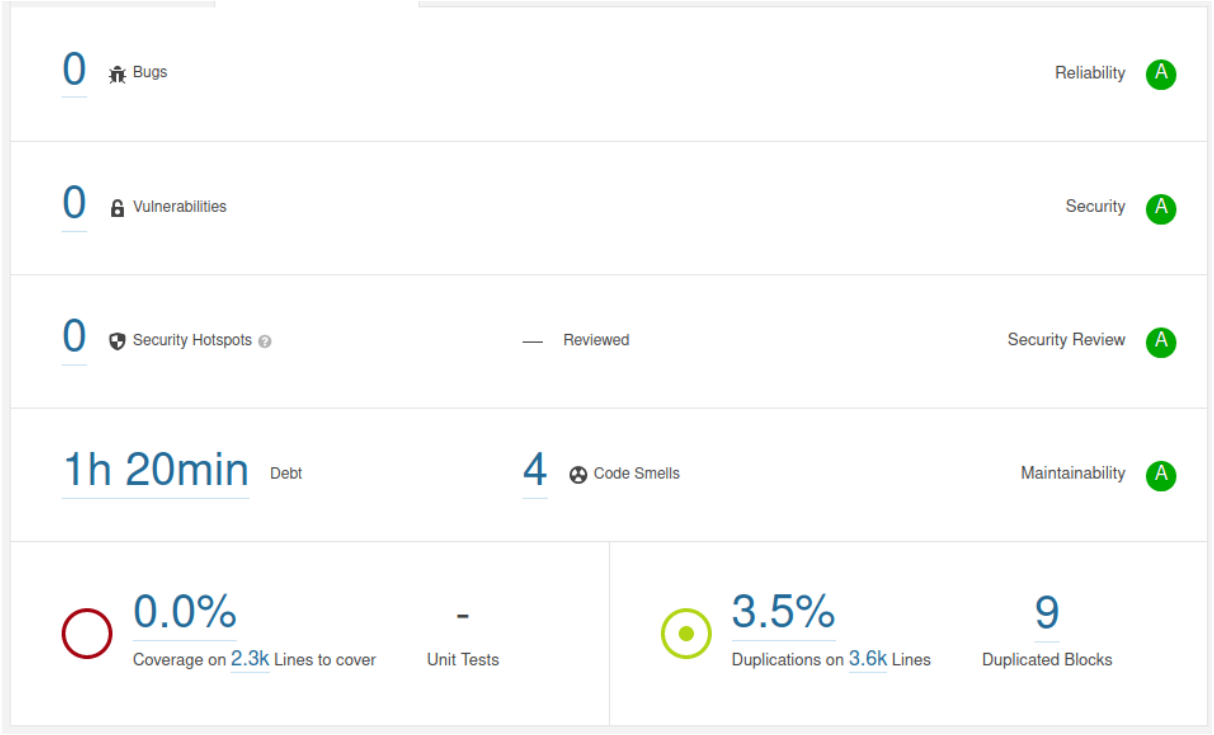


Figura 56: Projeto '54' - Depois do refactoring

- **Projeto 78**: neste projeto, concluí-se que todas as categorias ficaram classificadas com A exceto a *reliability* que ficou a C.
  - 2 bugs;
  - 0 vulnerabilidades;
  - 2 dias de *technical depth*;
  - 29 code smells;
  - 10.8% de linhas duplicadas;
  - 70 blocos de código duplicados;

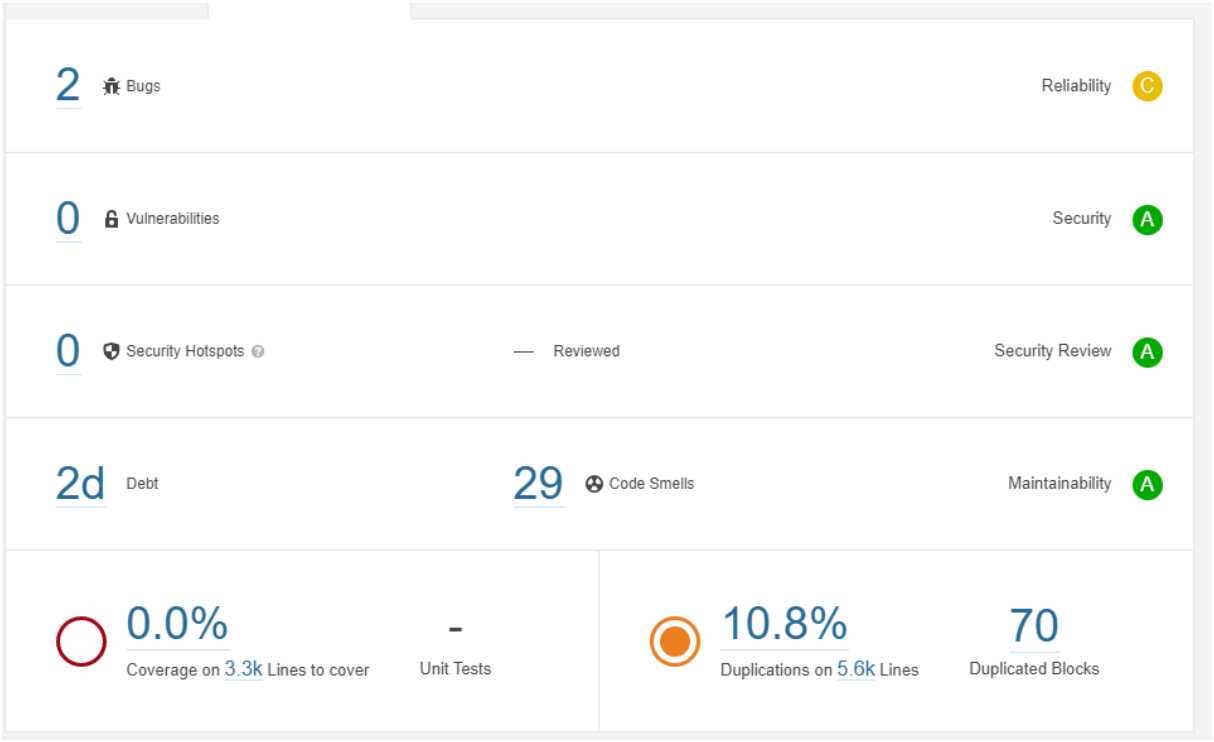


Figura 57: Projeto '78' - Depois do refactoring

Estes 29 smells são o resultado da diferença entre o números de smells final e o número de smells relativo aos packages e aos paths. Estes smells apenas aparecem quando colocamos os ficheiros java na pasta src/man/java e portanto são irrelevantes.

4 Tarefa 3

A tarefa seguinte está dividida em três partes principais:

1. Criação de testes unitários e de regressão;
2. Análise da cobertura dos testes efetuados;
3. Geração de ficheiros de *logs* (input da aplicação a analisar).

O foco desta tarefa é o teste do software. Testar um programa é importante por várias razões mas a principal é para verificar o correto funcionamento da mesma e que este não contém erros. Existem vários tipos de técnicas para testar o software em questão. Nas sub-secções seguintes apresentam-se os passos utilizados para tal assim como para a geração dos ficheiros de *logs*.

4.1 Testes unitários e de regressão com o sistema EvoSuite

Um **teste unitário** consiste em testar uma pequena porção do código em causa, pequenas *units* de código independentes umas das outras para determinar se estão corretas. Por outro lado, **testes de regressão** são testes utilizados para verificar se software já testado mantém o seu correto funcionamento após a realização de alterações neste.

Para a criação destes testes utilizou-se um sistema chamado **EvoSuite**, um plugin para o editor **IntelIJ** que permite criar testes unitários.

4.2 EvoSuite

O EvoSuite é uma ferramenta que gera automaticamente testes unitários para software desenvolvido em Java. O EvoSuite usa um algoritmo evolutivo para gerar testes JUnit e pode ser executado a partir

da linha de comandos ou também possui plugins para ser integrado no Maven, IntelliJ e Eclipse. Em contexto de projeto, o evosuite foi utilizado logo no início da parte da analise com testes unitários porque permite constroi esses mesmos testes.

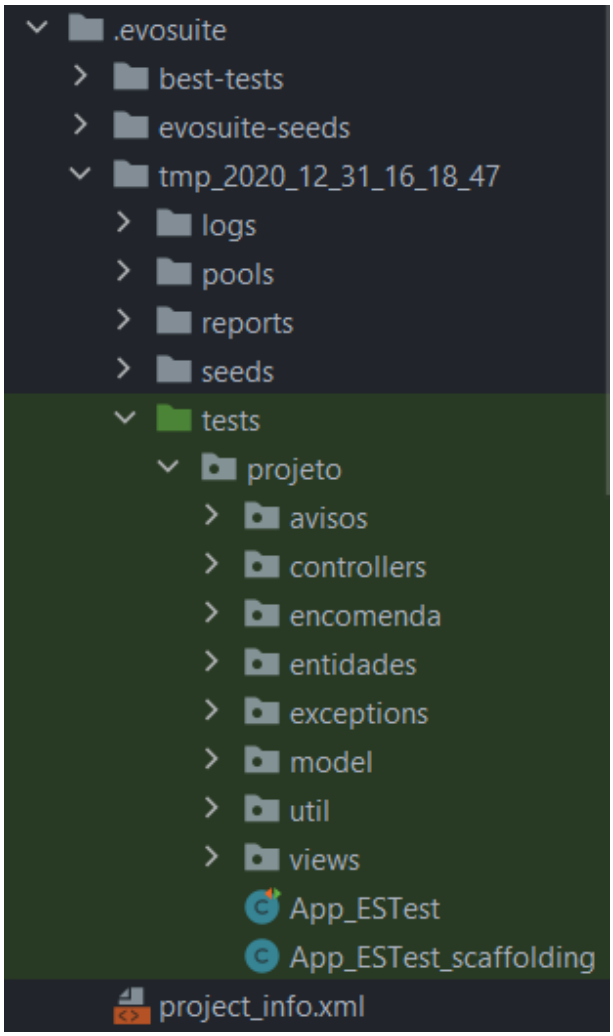


Figura 58: Geração de testes a partir do evosuite. (projeto 54)

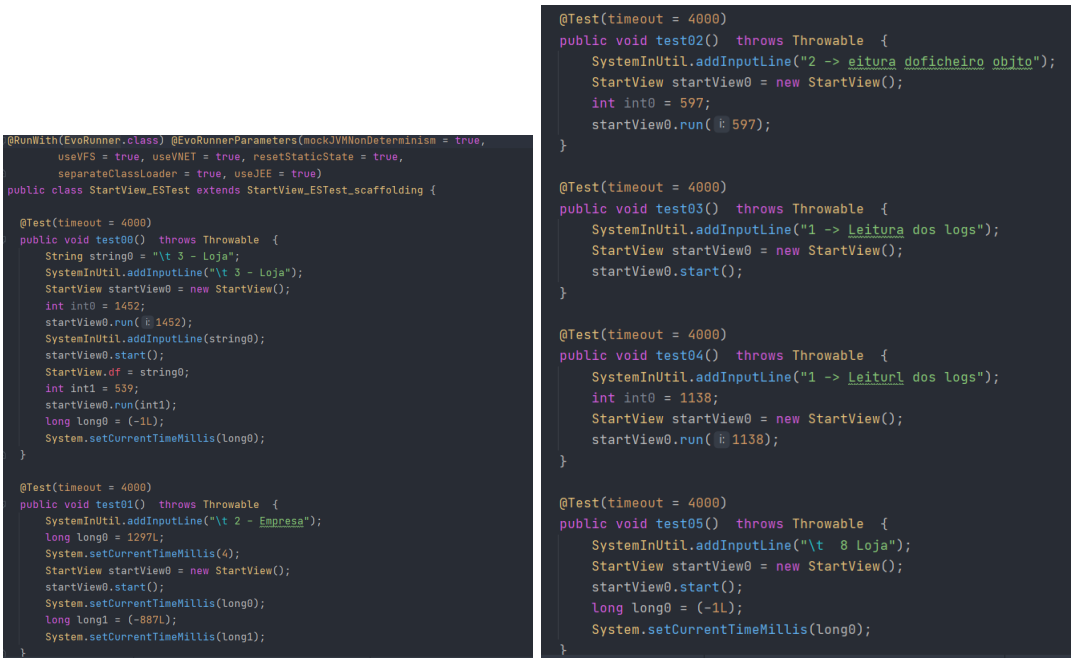
4.3 JUnit

É um framework open-source para construção de testes automatizados em Java, hospedado no Github, em que se verifica as funcionalidades de classes e seus métodos. Além disso, podemos automatizar também a execução de todos os testes de forma que quando há uma nova versão estável do sistema, o framework execute todos os testes para garantir a integridade e estabilidade do que foi desenvolvido.

Iniciou-se por desenvolver alguns testes unitários para as diferentes classes de cada projeto, verificando se esse teste resulta no estado esperado ou executa a sequência de eventos esperada (teste de comportamento). Nas dois tópicos seguintes são apresentados alguns exemplos de testes para os dois projetos.

4.3.1 Projeto 54

Para demonstrar os testes unitários gerados pelo evosuite no projeto 54 temos os seguintes exemplos:



```
@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true,
        useVFS = true, useVNET = true, resetStaticState = true,
        separateClassLoader = true, useJEE = true)
public class StartView_ESTest extends StartView_ESTest_scaffolding {

  @Test(timeout = 4000)
  public void test00()  throws Throwable  {
      String string0 = "\t 3 - Loja";
      SystemInUtil.addInputLine("\t 3 - Loja");
      StartView startView0 = new StartView();
      int int0 = 1452;
      startView0.run( r: 1452);
      SystemInUtil.addInputLine(string0);
      startView0.start();
      StartView.df = string0;
      int int1 = 539;
      startView0.run(int1);
      long long0 = (-1L);
      System.setCurrentTimeMillis(long0);
  }

  @Test(timeout = 4000)
  public void test01()  throws Throwable  {
      SystemInUtil.addInputLine("\t 2 - Empresa");
      long long0 = 1297L;
      System.setCurrentTimeMillis(4);
      StartView startView0 = new StartView();
      startView0.start();
      System.setCurrentTimeMillis(long0);
      long long1 = (-887L);
      System.setCurrentTimeMillis(long1);
  }
```

```
@Test(timeout = 4000)
public void test02()  throws Throwable  {
    SystemInUtil.addInputLine("2 -> eitura doficheiro objto");
    StartView startView0 = new StartView();
    int int0 = 597;
    startView0.run( r: 597);
}

@Test(timeout = 4000)
public void test03()  throws Throwable  {
    SystemInUtil.addInputLine("1 -> Leitura dos logs");
    StartView startView0 = new StartView();
    startView0.start();
}

@Test(timeout = 4000)
public void test04()  throws Throwable  {
    SystemInUtil.addInputLine("1 -> Leiturl dos logs");
    int int0 = 1138;
    StartView startView0 = new StartView();
    startView0.run( r: 1138);
}

@Test(timeout = 4000)
public void test05()  throws Throwable  {
    SystemInUtil.addInputLine("\t  8 Loja");
    StartView startView0 = new StartView();
    startView0.start();
    long long0 = (-1L);
    System.setCurrentTimeMillis(long0);
}
```

Figura 59: Testes JUnit

4.3.2 Projeto 78

Para demonstrar os testes unitários gerados pelo evosuite no projeto 78 temos os seguintes exemplos:

```
@Test(timeout = 4000)
public void test2()  throws Throwable  {
    TrazAqui trazAqui0 = new TrazAqui();
    ControllerLogin controllerLogin0 = new ControllerLogin(trazAqui0);
    // Undeclared exception!
    try {
      controllerLogin0.processa((List<String>) null);
      fail("Expecting exception: NullPointerException");

    } catch(NullPointerException e) {
       //
       // no message in exception (getMessage() returned null)
       //
       verifyException("controllers.ControllerLogin", e);
    }
}

@Test(timeout = 4000)
public void test3()  throws Throwable  {
    TrazAqui trazAqui0 = new TrazAqui();
    LinkedList<String> linkedList0 = new LinkedList<String>();
    linkedList0.add("Creat");
    linkedList0.add("Transuortadora");
    ControllerLogin controllerLogin0 = new ControllerLogin(trazAqui0);
    controllerLogin0.processa(linkedList0);
    assertEquals( expected: 2, linkedList0.size());
}
```

```
@Test(timeout = 4000)
public void test3()  throws Throwable  {
    TrazAqui trazAqui0 = new TrazAqui();
    LinkedList<String> linkedList0 = new LinkedList<>();
    linkedList0.add("");
    ControllerUtilizador controllerUtilizador0 = new ControllerUtilizador(trazAqui0, utilizador: "");
    controllerUtilizador0.processa(linkedList0);
    assertEquals( expected: 1, linkedList0.size());
}

@Test(timeout = 4000)
public void test4()  throws Throwable  {
    LinkedList<String> linkedList0 = new LinkedList<>();
    linkedList0.add("NumeroMensagens");
    ControllerUtilizador controllerUtilizador0 = new ControllerUtilizador((TrazAqui) null, utilizador: "NumeroMensagens");
    controllerUtilizador0.processa(linkedList0);
    assertTrue(linkedList0.contains("NumeroMensagens"));
}
```

Figura 60: Testes JUnit

4.4 Análise da cobertura dos testes com o sistema JaCoCo

Para utilizar os testes gerados, foi utilizado a ferramenta JaCoCo parar analisar a cobertura do código.

O JaCoCO permite avaliar em percentagem a quantidade de código que os testes unitários cobrem. Podemos observar de seguida os dados estatísticos da cobertura das classes anteriormente apresentadas.

Para demonstrar este sistema em uso, realizaram-se os seguintes testes para ambos os projetos que se têm vindo a analisar ao longo deste trabalho (54 e 78) antes e depois do refactoring. De seguida

apresentam-se os resultados da cobertura para ambos.

4.4.1 Projeto 54

Element	Class, %	Method, %	Line, %
StartView	100% (1/1)	30% (3/10)	20% (38/183)
ViewEmpresa	100% (1/1)	71% (10/14)	21% (57/268)
ViewLoja	100% (1/1)	62% (5/8)	22% (34/150)
ViewUtilizador	100% (1/1)	27% (3/11)	15% (30/194)
ViewVoluntario	100% (1/1)	75% (6/8)	19% (31/160)

Figura 61: Cobertura JaCoCo - antes do refactoring.

100% classes, 23% lines covered in package 'projeto.views'

Element	Class, %	Method, %	Line, %
StartView	100% (1/1)	33% (4/12)	26% (51/195)
ViewEmpresa	100% (1/1)	44% (8/18)	11% (31/261)
ViewLoja	100% (1/1)	63% (7/11)	27% (44/162)
ViewUtilizador	100% (1/1)	64% (9/14)	42% (85/202)
ViewVoluntario	100% (1/1)	30% (4/13)	14% (26/180)

100% classes, 67% lines covered in package 'projeto.controllers'

Element	Class, %	Method, %	Line, %
Controller	100% (1/1)	100% (7/7)	94% (83/88)
ControllerEmpresa	100% (1/1)	96% (31/32)	55% (38/69)
ControllerLoja	100% (1/1)	100% (15/15)	46% (24/52)
ControllerUtilizador	100% (1/1)	100% (19/19)	77% (80/103)
ControllerVoluntario	100% (1/1)	96% (25/26)	46% (36/77)

Figura 62: Cobertura JaCoCo - depois do refactoring.

4.4.2 Projeto 78

Loja	100% (3/3)	100% (6/6)	47% (28/59)
Transportadores	100% (4/4)	100% (8/8)	55% (51/92)
Utilizador	100% (5/5)	100% (10/10)	57% (47/82)
Controller_Login	100% (2/2)	100% (3/3)	18% (18/99)
Controller_SistemaTodo	100% (1/1)	100% (2/2)	84% (22/26)

Element	Class, %	Method, %	Line, %
Controller_UtilizadorAceitaPendentes	100% (1/1)	100% (2/2)	88% (8/9)
Controller_UtilizadorAvalia	100% (1/1)	100% (2/2)	45% (5/11)
Controller_UtilizadorGeraEncomenda	100% (1/1)	100% (2/2)	38% (5/13)
Controller_UtilizadorGeraLinhaEncomenda	100% (1/1)	100% (2/2)	45% (11/24)
ControllerUtilizador	100% (1/1)	100% (2/2)	72% (18/25)

Figura 63: Cobertura JaCoCo - antes do refactoring.

Element	Class, %	Method, %	Line, %
loja	100% (3/3)	78% (11/14)	52% (37/71)
transportadores	100% (4/4)	95% (22/23)	59% (72/121)
utilizador	100% (5/5)	94% (18/19)	52% (54/102)
ControllerLogin	100% (2/2)	57% (8/14)	21% (25/117)
ControllerSistemaTodo	100% (1/1)	100% (4/4)	85% (24/28)

Figura 64: Cobertura JaCoCo - depois do refactoring.

Como podemos observar, os refactorings realizados pouco alteram os resultados da cobertura dos testes, nuns casos melhorando e noutros dando valores ligeiramente abaixo. No geral estes apresenta a mesma cobertura.

4.5 Geração de ficheiros de *logs*

Para a geração de ficheiros *logs* foi criado um programa em **Haskell** que tira partido do sistema **QuickCheck** para gerar as entradas do ficheiro. Nesta secção abordam-se as decisões tomadas ao gerar estas e não uma análise extensa do código em si.

O ficheiro *logs* em si foi criado de acordo com o formato disponibilizado pelos docentes. Para tal, este apresenta o seguinte formato para cada entrada (por ordem em que aparecem no ficheiro):

- Utilizador: *Código Utilizador, Nome, Coordenada X, Coordenada Y*
- Voluntário: *Código Voluntário, Nome, Coordenada X, Coordenada Y, Raio*
- Transportadora: *Código Transportadora, Nome, Coordenada X, Coordenada Y, NIF, Raio, Preço por km*
- Loja: *Código Loja, Nome, Coordenada X, Coordenada Y*
- Encomenda: *Código Encomenda, Código do utilizador, Código da loja, Peso, Lista de produtos*
- Produto: *Código Produto, Quantidade, Preço*
- Aceite: *Código da encomenda aceite*

Primeiramente, foram criadas 5 listas nas quais se encontram nomes válidos para a geração de entradas:

- Lista de produtos com bens alimentícios;
- Lista de nomes próprios portugueses;
- Lista de apelidos portugueses;
- Lista de empresas de transporte portuguesas;
- Lista de algumas lojas do Nova Arcada.

Para gerar um nome, foi escolhido aleatoriamente um nome próprio e um apelido e concatenaram-se estes dois. As coordenadas X estão contidas no intervalo [-9.32,-6.32] e as coordenadas Y estão contidas no intervalo [37.0,42.0], em Portugal.

Foi definido um valor máximo do raio de 200 km para o voluntário. O preço das encomendas varia também entre 0 e 150€.

O peso dos produtos varia entre 0 e 20 kg e o preço varia entre 0 e 60€.

Os códigos das encomendas aceites são geradas a partir dos códigos das encomendas gerados.

Para a geração de inputs são pedidos 5 inteiros para definir o número de utilizadores, número de voluntários, número de transportadoras, número de lojas e número e encomendas, respetivamente. Desta forma, é possível gerar inputs diferentes no que toca à quantidade.

5 Tarefa 4

Nesta tarefa é proposto a análise do desempenho dos projetos. As aplicações foram testadas com a framework de geração de inputs da tarefa anterior.

Esta tarefa é importante no que toca a analisar a influência dos bad smells e red smells no desempenho do software.

5.1 Tarefa Extra 4

No nosso caso analisaram-se ambos os projetos 54 e 78 nas suas versões antes do refactoring. Para além disso, foram utilizados em cada teste 2 ficheiros de input: *logsSmall.txt* e *logsBig.txt*. Estes foram gerados com diferentes números de registos sendo que cada tipo de entrada nos ficheiro tem 10 (no Small) ou 10000 (no Big).

Os parâmetros analisados foram o **tempo de execução**, a **memória atual** e o ***CPU usage***. No entanto, uma vez que o grupo não possuía nenhuma máquina com Linux nativo nem conseguiu encontrar uma API ou software que realizasse o mesmo para Windows, o software RAPL não pode ser utilizado. Em vez disso, foram realizados testes especificamente para cada projeto explicados de seguida.

Para cada um dos projetos criou-se uma classe chamada **TestesPerformance** onde se inicializou um Model e se realizou vários métodos que o grupo considerou bastante ilustrativos e que não necessitavam de input direto do utilizador. Nestes métodos são medidos os parâmetros referidos. O primeiro método a ser avaliado é sempre o método de leitura do ficheiro de logs (um dos dois). Para escolher o ficheiro a ler é necessário verificar na classe onde está a ser lido o ficheiro, qual é o desejado, e confirmar uma vez que se põe a correr a main da classe **TestesPerformance**.

Para se medir o *CPU usage* utilizou-se um software Windows Performance Analyser. Este software permite efetuar uma grande variedade de análises e medições de processos, consumo de memória, storage e energia.

Para permitir a medição de CPU foi então iniciada uma gravação com este software enquanto foram compilados os ficheiros logs (tanto o grande como o pequeno) o que permite efetuar a comparação entre o peso de cada um com os métodos utilizados em casa projeto. Selecionaram-se, após a gravação de cada um dos casos, os processos que eram relevantes e analisou-se o gráfico obtido após a captura. Antes da captura verificou-se que não haviam processos extra a correr , apenas os necessários.

De seguida encontra-se uma análise detalhada de cada projeto (antes e depois do refactoring) com cada um dos ficheiros de logs. (*Nota: os valores da DRAM são todos relativos à DRAM após a execução do método*)

5.1.1 Projeto 54

- Antes do refactoring:

Métodos	**DRAM (bytes)**	**Tempo de Execução (ms)**
parse	1655552	50
topNClientesEmpresa	1664752	5
topNEmpresasMaisUsadas	1666616	1
topNEmpresasDist	1671112	1
topNClientesMaisEncomendaram	1672568	1

Tabela 1: Métricas relativas ao ficheiro *logsSmall.txt* para o projeto '54'.

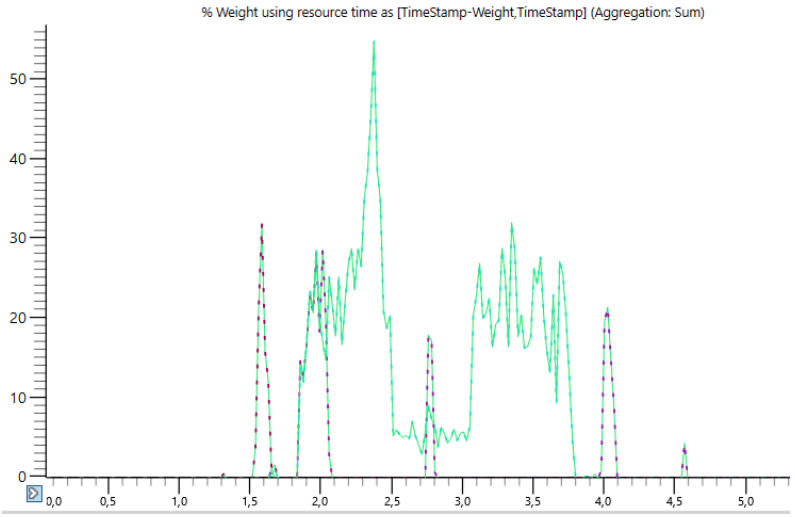


Figura 65: *CPU usage* relativas ao ficheiro *logsSmall.txt* para o projeto '54'

Métodos	DRAM (bytes)	Tempo de Execução (ms)
parse	13851144	634
topNClientesEmpresa	13862640	11
topNEmpresasMaisUsadas	13864960	10
topNEmpresasDist	13869704	7
topNClientesMaisEncomendaram	13872200	14

Tabela 2: Métricas relativas ao ficheiro *logsBig.txt* para o projeto '54'.

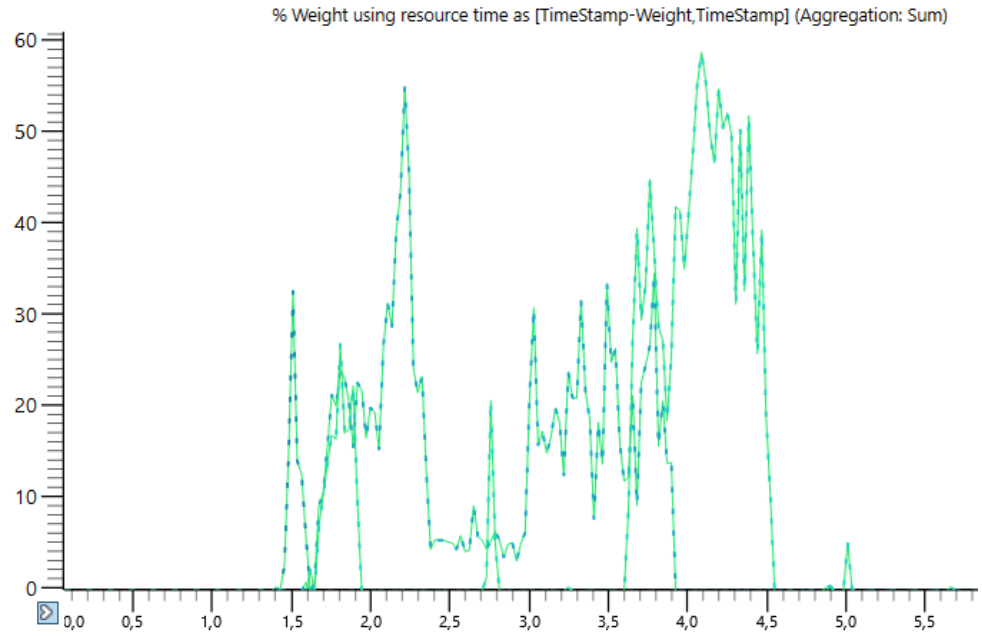


Figura 66: *CPU usage* relativas ao ficheiro *logsBig.txt* para o projeto '54'.

5.1.2 Projeto 78

- Antes do refactoring:

Métodos	DRAM (bytes)	Tempo de Execução (ms)
LeituraFicheiros	2233272	39
getHistoricoLoja	2241312	4
getListPedidosLoja	2243128	<0
getLojasDisponiveis	2245176	<0
getHistoricoUtilizador	2248032	1
getPedidosTransportadorasPendentes	2249864	1
getHistoricoTransportadoras	2252056	<0
totalFaturadoLoja	2259256	1
top10Transportadoras	2274936	2
top10Utilizadores	2277624	1

Tabela 3: Métricas relativas ao ficheiro *logsSmall.txt* para o projeto '78'.

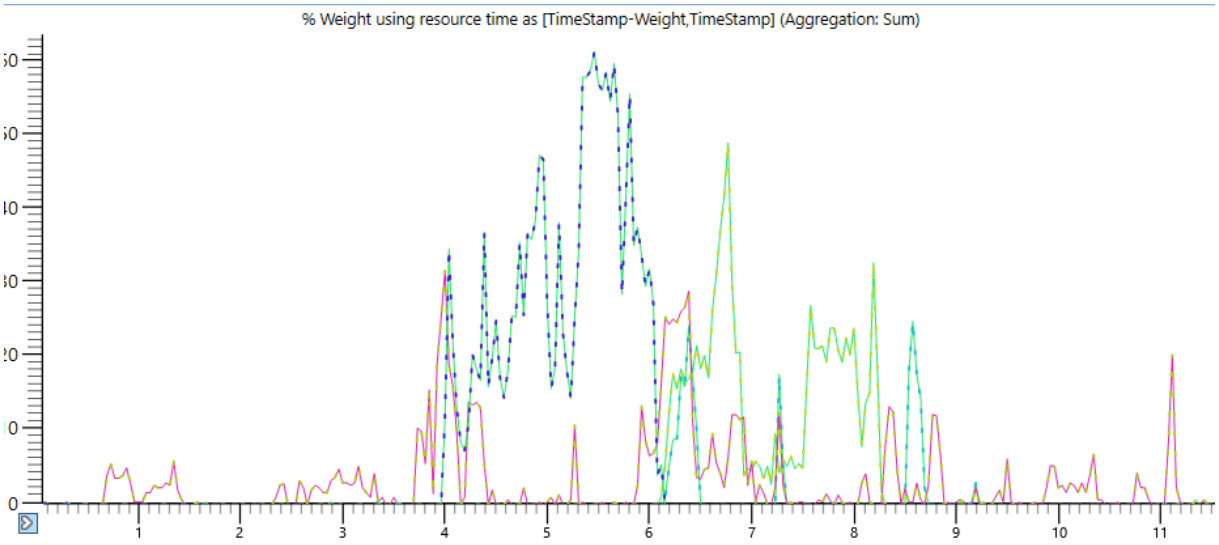


Figura 67: *CPU usage* relativas ao ficheiro *logsSmall.txt* para o projeto '78'.

Métodos	**DRAM (bytes)**	**Tempo de Execução (ms)**
LeituraFicheiros	59085280	1151
getHistoricoLoja	59096824	5
getListPedidosLoja	59098856	<0
getLojasDisponiveis	59406208	10
getHistoricoUtilizador	59409184	1
getPedidosTransportadorasPendentes	59411016	1
getHistoricoTransportadoras	59413208	<0
totalFaturadoLoja	59420336	2
top10Transportadoras	59436208	38
top10Utilizadores	59439024	15

Tabela 4: Métricas relativas ao ficheiro *logsBig.txt* para o projeto '78'.

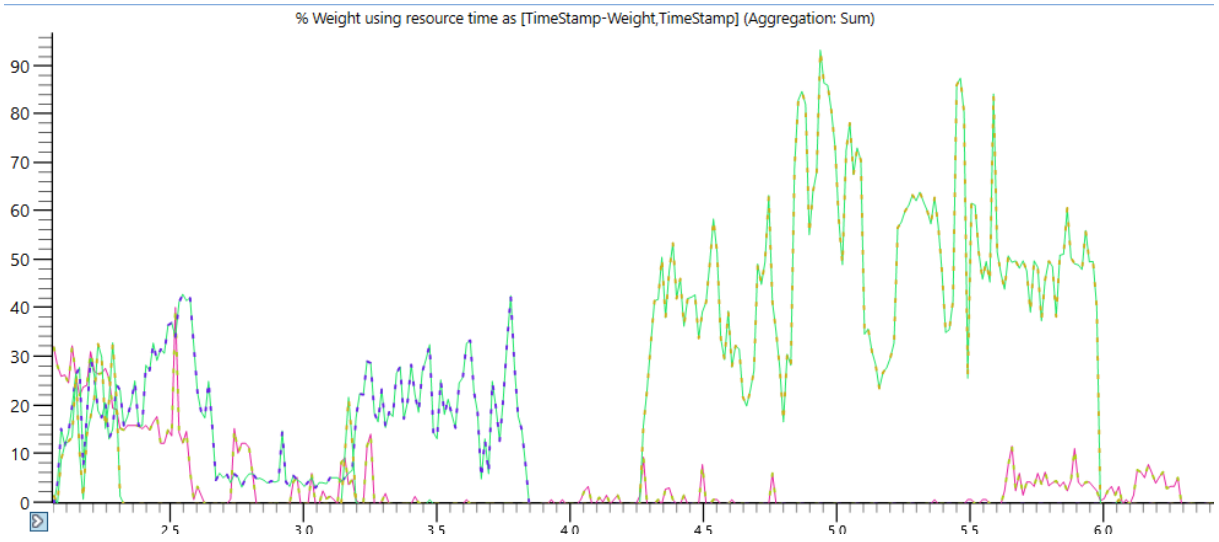


Figura 68: *CPU usage* relativas ao ficheiro *logsBig.txt* para o projeto '78'.

Em relação aos valores obtidos podem se tirar várias conclusões. Começando pelas tabelas apresentadas, podemos primeiramente observar que em ambas as versões do código, para os ficheiros *logsSmall.txt*

e *logsBig.txt* há uma óbvia e previsível descrepância entre os valores do tempo de execução e DRAM. Claro que os valores do ficheiro maior serão também maiores.

Em relação aos gráficos do *CPU usage* é também óbvio que este é menor para os ficheiros *logsSmall.txt* em comparação com os ficheiros maiores.

6 Conclusão

Ao longo deste relatório foram analisados vários métodos que permitem uma analise de qualidade de código em várias aplicações.

Numa primeira parte, foi feita uma analise inicial a partir da ferramenta **SonarQube** que permite analisar métricas relevantes tais como número de bugs, code smells, número de métodos e classes, NLOC, entre outros. Numa segunda instância e após a verificação da existência de code smells, foram escolhidos dois projectos para efetuar **refactoring** e correção dos mesmos. Nesta fase foi definida a diferença entre code smells e red smells o que permitiu uma nova análise da qualidade do código, após o refactoring. Na terceira fase deste projeto foi desenvolvido um gerador de logs, desenvolvido em Haskell, no qual foi implementado de modo a gerar casos os mais distintos e aleatórios o possível. Nesta mesma fase foram gerados teste unitários a partir da ferramenta **evosuite** e de seguida foi analisada a cobertura destes mesmos testes a utilizando o **JaCoCo**. Com a quarta e última tarefa, ainda que não se tenha utilizado o software RAPL, o grupo conseguiu adaptar-se e tirar o melhor da situação revelando esforço e um espírito pro-ativo para ultrapassar os obstáculos que surgiram. Para além disso, tiraram-se conclusões interessantes da análise dos resultados.

Após estas etapas, concluímos que os projetos têm sempre hipótese de melhorias com mais análises e um refactoring mais detalhado em conjunto com análises de consumo de energia e tempo de execução. Ao longo deste projeto obtivemos e desenvolvemos conhecimentos úteis e necessários para efetuar a devida analise e teste de software o que, abre portas a uma nova área de grande importância no desenvolvimento de todos os projetos.