

UNIVERSIDADE DO MINHO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

ANÁLISE E TESTE DE SOFTWARE

TrazAqui!

Etienne Costa (a76089)
Majka Medvidova (e9867)
Maurício Salgado (a71407)
Rui Azevedo (a80789)

25 de janeiro de 2021

Conteúdo

1	Introdução	3
2	Qualidade do Código Fonte	4
2.1	Java Code Analyser	4
2.2	Análise de resultados	9
3	Refactoring	11
3.1	Refactoring do Projecto 23	11
3.2	Refactoring do Projecto 88	15
4	Teste	18
4.1	Cobertura	18
4.2	Geração de logfiles	19
5	Análise de Desempenho	24
5.1	Desempenho do Projecto 23	25
5.2	Desempenho do Projecto 88	25
5.3	Análise dos resultados	26
6	Conclusão	27

Lista de Figuras

1	Página Inicial do <i>JCA</i>	4
2	Página <i>Projects</i> do <i>JCA</i>	5
3	Tabela de métricas da página <i>Dashboard</i> do <i>JCA</i>	6
4	<i>Pie Charts</i> da página <i>Dashboard</i> do <i>JCA</i>	8
5	<i>Boxplot</i> da página <i>Dashboard</i> do <i>JCA</i>	8
6	Basic layout	9
7	<i>Boxplot</i> das métricas <i>bugs</i> e <i>issues</i>	10
8	<i>Boxplot</i> da métrica <i>cyclomatic complexity</i>	10
9	Métricas do Projecto 23 no <i>SonarQube</i>	12
10	Métricas do Projecto 23 no <i>SonarQube</i>	14
11	Métricas do Projecto 88 antes do refactoring no <i>SonarQube</i>	15
12	Métricas do Projecto 88 após refactoring no <i>SonarQube</i>	17
13	Resultado dos testes gerados pelo Evosuite	18
14	Cobertura dos testes gerados	19

1 Introdução

O presente relatório é referente ao trabalho prático desenvolvido na Unidade Curricular **Análise e Teste de Software**, do 4º ano do Mestrado Integrado de Engenharia Informática, da Universidade do Minho.

É pretendido com este trabalho prático fazer a análise e teste de um conjunto de projectos escritos em *Java*, projectos esses desenvolvidos pelos alunos do 2º ano da Licenciatura de Engenharia Informática na Unidade Curricular de Programação Orientada a Objectos.

Numa primeira fase, irá ser analisada a qualidade do código fonte de cada projecto. Para esta fase foi criada uma pequena aplicação que utiliza o sistema *SonarQube* para recolher métricas de *software* e, consequentemente, *code smells*.

De seguida, foram escolhidos dois projectos para aplicar-se *refactoring* ao código fonte com o objectivo de eliminar o maior número de *code smells* e *red smells* presentes no código. Para isto, foram usados o *SonarQube*, para recolher informação específica de cada *code/red smell*, e o *IntelliJ IDEA* como auxiliar no processo de *refactoring*.

Uma vez feita a análise dos projectos, prossegui-se com a fase de testes aos projectos escolhidos. Para esse fim, foram usadas as ferramentas *EvoSuite* capaz de gerar testes unitários automaticamente, *IntelliJ IDEA* para analisar a cobertura dos testes, e por fim, *QuickCheck* para gerar ficheiros de *logs*, uma vez que a aplicação em estudo tem a possibilidade de ser povoado através deste ficheiro.

Por fim, será feita uma análise quanto ao desempenho da aplicação, tanto a nível do tempo de execução como consumo energético. A ferramenta *RAPL* permite fazer este tipo de análise uma vez fornecido o executável da aplicação.

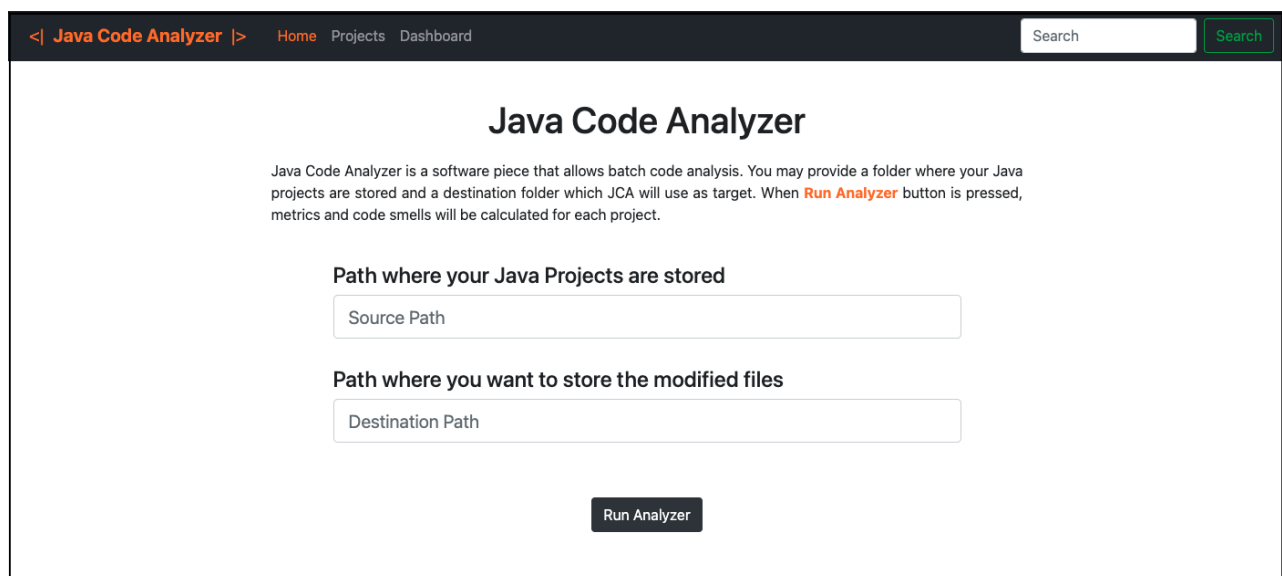
2 Qualidade do Código Fonte

Para a análise do código fonte de todos os projectos, o grupo desenvolveu uma pequena aplicação *web*, à qual se designou *Java Code Analyser (JCA)*, escrita em *Python* usando a *framework Django*, que permite visualizar as métricas de todos os projectos de uma maneira bastante apelativa e fácil.

Os dados apresentados no *JCA* foram cruciais na verificação de quais projectos com melhor e pior qualidade, o que tornou muito mais simples a decisão de escolha de quais os projectos a analisar nas tarefas seguintes.

2.1 Java Code Analyser

A ferramenta *SonarQube* obriga a que a estrutura do sistema de ficheiros esteja normalizada da seguinte maneira: `${project_name}/src/main/java/${project_content}`. Uma vez que a maior parte dos trabalhos não seguem esta estrutura, foi criada a funcionalidade de fazer a organização automática dos ficheiros. A página inicial da aplicação é a apresentada na figura abaixo.



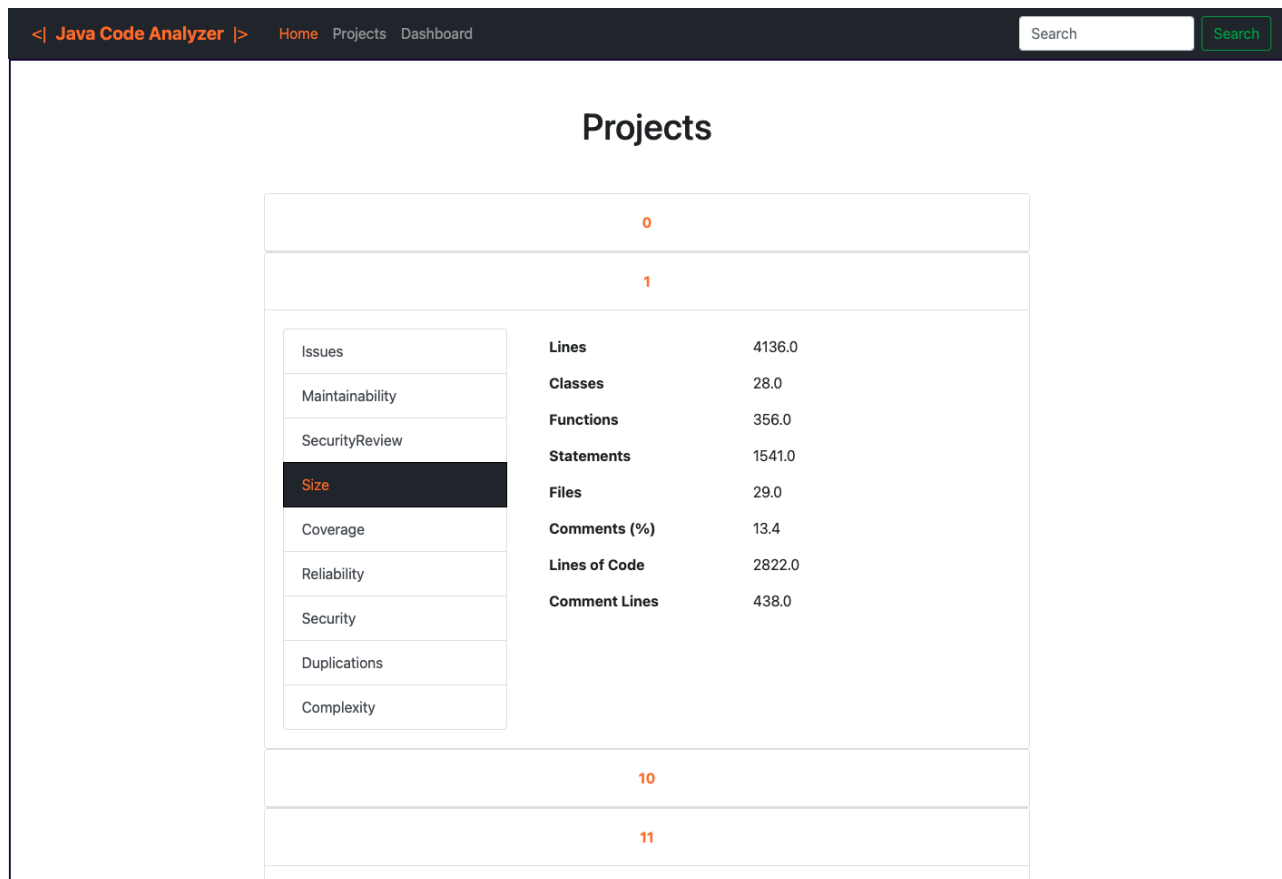
The screenshot shows the initial page of the Java Code Analyzer (JCA) web application. At the top, there is a dark navigation bar with the text "<| Java Code Analyzer |>" on the left, and "Home Projects Dashboard" in the center. On the right side of the bar is a search bar with the placeholder text "Search" and a green "Search" button. Below the navigation bar, the main content area has a white background. It features the title "Java Code Analyzer" in a large, bold, black font. Underneath the title is a paragraph of text: "Java Code Analyzer is a software piece that allows batch code analysis. You may provide a folder where your Java projects are stored and a destination folder which JCA will use as target. When **Run Analyzer** button is pressed, metrics and code smells will be calculated for each project." Below this text are two input fields. The first is labeled "Path where your Java Projects are stored" and contains the placeholder text "Source Path". The second is labeled "Path where you want to store the modified files" and contains the placeholder text "Destination Path". At the bottom center of the form is a dark button with the text "Run Analyzer" in white.

Figura 1: Página Inicial do *JCA*

É possível, com esta funcionalidade, fornecer o caminho para a pasta onde se encontram todos os projectos e uma pasta destino onde serão guardados os projectos normalizados. Quando o botão *Run Analyzer* é pressionado, o *JCA* organiza, compila e corre o *SonarQube*

sobre todos os projectos contidos na pasta fonte. Caso só se forneça uma pasta destinado, significando que os projectos já estão estruturados correctamente, a parte de organização do sistema de ficheiros é descartada e os projectos são apenas enviados ao *SonarQube*.

Quando este processo é finalizado, é possível observar os resultados de duas maneiras diferentes. A página *Projects* dá acesso a todas as métricas calculadas para cada projecto. Após uma análise aos dados que são retornados pelo *SonarQube*, foi possível observar que cada métrica tem um domínio, por isso, as métricas apresentadas nesta página estão divididas pelos seus domínios.



Projects		
0		
1		
Issues	Lines	4136.0
Maintainability	Classes	28.0
SecurityReview	Functions	356.0
Size	Statements	1541.0
Coverage	Files	29.0
Reliability	Comments (%)	13.4
Security	Lines of Code	2822.0
Duplications	Comment Lines	438.0
Complexity		
10		
11		

Figura 2: Página *Projects* do *JCA*

Enquanto que na página anterior é possível ter acesso aos dados de uma forma particular, na página *Dashboard* é possível ver dados globais sobre os projectos. A primeira parte desta página apresenta uma tabela com algumas métricas escolhidas para análise onde é possível observar quais os projectos que tiveram valores mais altos e mais baixos numa determinada métrica.

<| Java Code Analyzer |>

Home Projects Dashboard

Search

Search

Dashboard

Overall

Metric	Highest	Lowest
Technical Debt	Project: 23 Value: 17d0h	Project: 88 Value: 2d6h
Code Smells	Project: 83 Value: 815	Project: 88 Value: 153
Bugs	Project: 39 Value: 74	Project: 5 Value: 0
Blocker Issues	Project: 39 Value: 44	Project: 5 Value: 0
Critical Issues	Project: 9 Value: 85	Project: 94 Value: 1
Issues	Project: 83 Value: 845	Project: 31 Value: 158
Lines of Code	Project: 73 Value: 6460	Project: 13 Value: 1117
Classes	Project: 78 Value: 66	Project: 7 Value: 9
Vulnerabilities	Project: 78 Value: 3	Project: 0 Value: 0
Cyclomatic Complexity	Project: 73 Value: 1355	Project: 13 Value: 169

Figura 3: Tabela de métricas da página *Dashboard* do *JCA*

De todas as métricas disponibilizadas pelo *SonarQube*, as escolhidas foram as que estão apresentadas na Figura 3 e têm o seguinte significado:

- ***Code Smells*** : mede o número de violações que existem no código tendo em conta os princípios fundamentais de *design* de uma certa linguagem, neste caso, o *Java*.
- ***Technical Debt*** : mede o esforço para reparar todos os *code smells*. O cálculo deste valor assume que um dia tem oito horas de trabalho.
- ***Bugs*** : conta o número de *bugs* que existem num determinado projecto.
- ***Issues*** : sempre que o *SonarQube* faz uma análise sobre um projecto, levanta um erro sempre que um pedaço de código viola uma regra. Essas regras estão associadas ao *Quality Profile* de cada linguagem. Os erros podem ser divididos em cinco categorias: *info*, *minor*, *critical* e *blocker*, onde cada nível mede um nível de impacto diferente do erro no código. Decidiu-se escolher os erros críticos e bloqueantes por serem os mais relevantes

num código, significando que o mesmo tem que ser imediatamente revisto e modificado. Os erros críticos podem representar erros de segurança, como por exemplo, blocos *try-catch* vazios e *SQL injections*. Por outro lado, os erros bloqueantes representam os *bugs* que têm grande probabilidade de ter um impacto alto no comportamento da aplicação em produção e podem ser, por exemplo, *memory leaks* e conexões *JDBC* não fechadas.

- ***Lines of Code*** : número de linhas de código. Embora esta métrica não seja indicativa da qualidade do código fonte, decidiu-se considerar esta métrica apenas para observar a diferença do seu valor entre os projectos.
- ***Classes*** : número total de classes. Tal como as linhas de código, o número de classes não é grande indicativo da qualidade do código, no entanto, considerou-se esta métrica para ter uma ideia geral dos valores.
- ***Vulnerabilities*** : é considerada uma vulnerabilidade qualquer ponto do código aberto para ataques.
- ***Cyclomatic Complexity*** : mede o número de caminhos linearmente independentes através do código fonte de um programa. Para calcular esta métrica, o *SonarQube* inicializa um contador com o valor de 1 e sempre que o controlo de fluxo divide-se, o contador é incrementado. As *keywords* que incrementam a complexidade ciclomática são as seguintes: *if, for, while, case, catch, throw, , ||, ?*.

Há excepção das métricas meramente quantitativas, particularmente as linhas de código e o número de classes, que apenas foram usadas simplesmente para observar a diferença entre os diferentes projectos, as outras métricas foram consideradas as mais interessantes e representativas da qualidade do código fonte dos projectos. Quanto maiores os valores destas métricas mais complexo é código, maiores as falhas, e, conseqüentemente, mais tempo demorará a sua correção.

Na mesma página, logo abaixo da tabela, é possível visualizar dois *Pie Charts* relativos aos projectos que tiveram maior e menor *Technical Debt*. Cada gráfico apresenta o valor de um sub-conjunto de métricas dos respetivos projectos. Decidiu-se salientar esta métrica por, possivelmente, ser a que apresenta maiores custos empresariais, uma vez que quanto mais tempo se demore a corrigir um programa, mais dinheiro uma empresa irá gastar em trabalho que, se o programa fosse bem escrito à partida, era desnecessário.

A Figura 4 apresenta os dois *Pie Charts* mencionados anteriormente. Os gráficos foram feitos utilizando a biblioteca *Pandas*, muito usada para análise de dados. É de notar que os gráficos são bastante interativos, onde é possível desativar métricas clicando nas caixas onde estão definidos os seus nomes e, quando se passa com o rato por cima de uma área do gráfico, é possível observar o valor da métrica, como se pode verificar na Figura 4, no caso das violações.

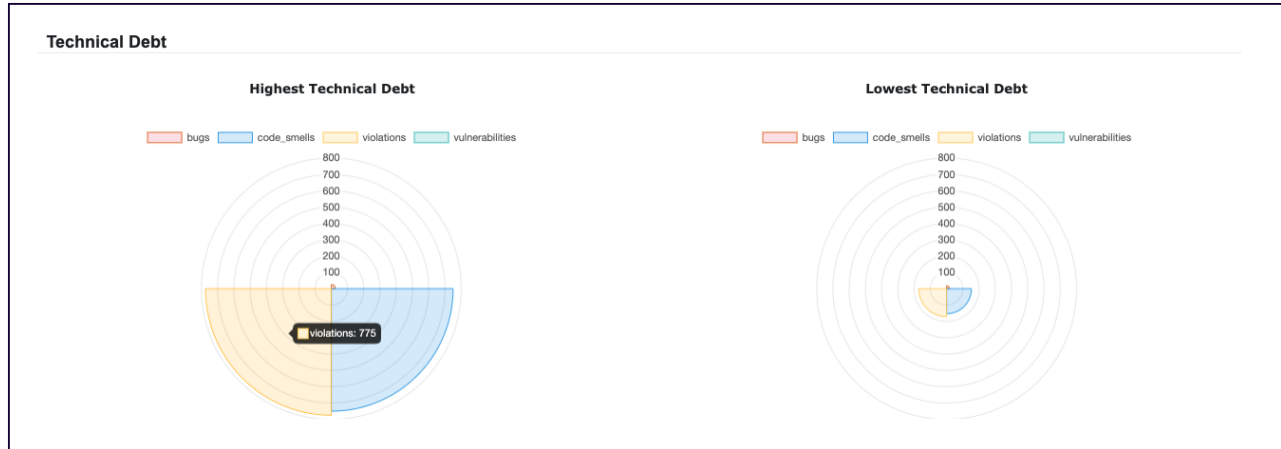


Figura 4: *Pie Charts* da página *Dashboard* do *JCA*

Por fim, ainda na mesma página, é possível observar dados globais de algumas métricas através de um diagrama *boxplot*. Estes gráfico permitem observar os dados divididos por quartis, onde $q1$ representa o número de valores abaixo de 25%, $q2$ abaixo de 50% e $q3$ abaixo de 75%. É também possível observar os *outliers* e o valor mediano. A Figura 5 apresenta o exemplo no caso das linhas de código. Como se pode observar, a maior parte dos projectos têm o número de linhas entre as 2061 e 3256.



Figura 5: *Boxplot* da página *Dashboard* do *JCA*

Uma vez que as ordens de grandeza das métricas são diferentes, a visualização dos dados não fica tão clara. Este problema é colmatado desactivando e activando as métricas que se pretende visualizar, tornando assim possível uma clarificação dos valores obtidos. Outra maneira de visualizar os dados é fazer *zoom* das caixas para melhor leitura dos dados. É possível também, caso seja necessário, fazer *download* dos gráficos em *SVG*.

Esta funcionalidade foi desenvolvida utilizando a biblioteca *plotly*, uma ferramenta para visualização de dados.

2.2 Análise de resultados

A ferramenta desenvolvida foi bastante útil na fase de análise dos resultados obtidos, tornando possível verificar quais os melhores e piores projectos. A análise apresentada a seguir tem como referências as imagens apresentadas anteriormente.

O projecto com maior *Technical Debt* é o 23 mas, no entanto, não é o projecto que tem mais *code smells*. Isto é indicativo que, apesar de o Projecto 83 ter mais *code smells*, os mesmos demoram menos tempo a resolver que o Projecto 23. É de notar que o número de *code smells* do Projecto 23 é na mesma muito elevado, tendo um valor de 750. O projecto que apresenta maior *bugs* é também o projecto que apresenta mais valores de erros bloqueantes, pelo que pode ser considerado um projecto com pouca qualidade de código. Um facto interessante que se pode observar na tabela é o de o projecto com mais linhas de código ser o projecto que apresenta também maior complexidade ciclomática e, analogamente, o projecto com menos linhas de código é o que apresenta menor complexidade ciclomática. Estas duas métricas estão relacionadas pelo que, quanto mais linhas de código um programa tem, maior é a probabilidade de acrescentar mais complexidade. Este facto não é necessariamente verdade em todos os casos, pelo que é possível adicionar linhas de código sem acrescentar complexidade.

Quanto aos valores mais baixos obtidos, verifica-se que o projecto que tem menos *Technical Debt* é por sua vez o projecto com menos *code smells*, sendo que estas métricas estão diretamente relacionadas uma vez que o débito é calculado através do tempo que demora a resolver *code smells*.

Como foi mencionado anteriormente, os gráficos *boxplot* permitem ver, de uma forma global, os valores de algumas das métricas calculadas. Isto permite observar entre que gamas de valores essas métricas se encontram e, conseqüentemente, ter uma visão geral da qualidade dos projectos.

Em relação à métrica *code smells*, verifica-se que o seu valor é bastante elevado, onde 75% dos projectos têm um valor superior a 242. Visto que os trabalhos foram realizados por alunos que estavam a ter o primeiro contacto com a linguagem, é normal os mesmos não terem muito conhecimento sobre as

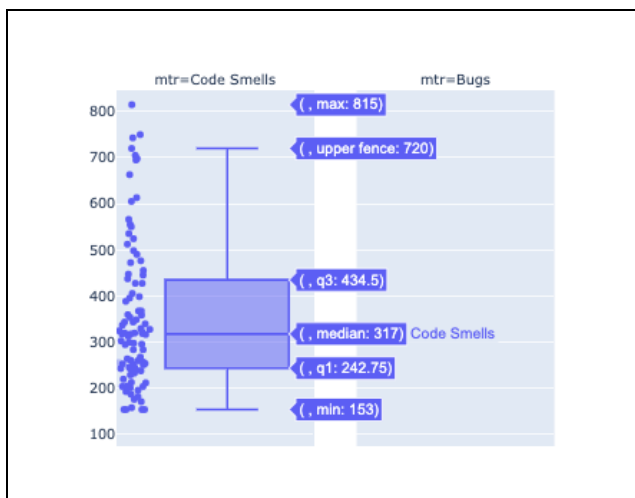
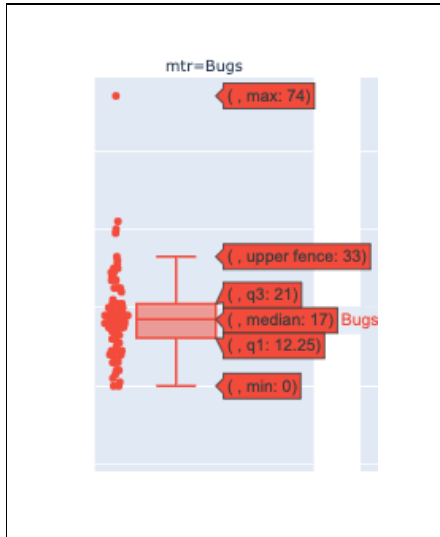


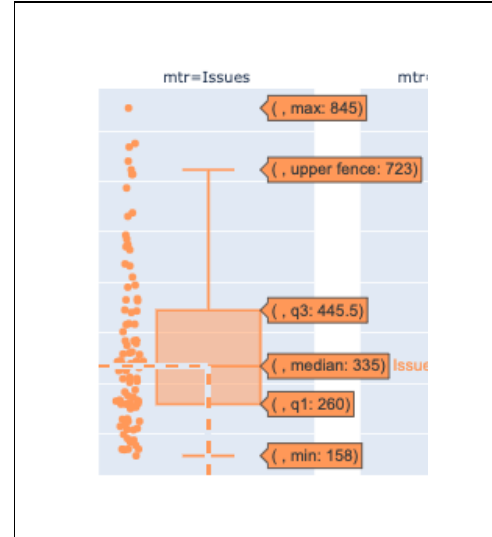
Figura 6: Basic layout

melhores práticas do paradigma nem sobre os princípios fundamentais de *design* da linguagem.

No que diz respeito ao número de *bugs*, observa-se que o seu valor é bastante baixo de uma forma geral. Há um projecto que chega a atingir os 74, nomeadamente o projecto 39, como se pode observar na Figura 3. Há excepção deste *outlier*, os valores encontram-se entre os 0 e 33, sendo a maior parte abaixo dos 21.



(a) *Boxplot* da métrica *bugs*



(b) *Boxplot* da métrica *issues*

Figura 7: *Boxplot* das métricas *bugs* e *issues*

A quantidade de *issues* calculados pelo *SonarQube* também é bastante alto de uma forma geral, estando este valor limitado entre 158 e 723, tirando os *outliers*, que atingem no máximo um valor de 845.

Analisando agora a complexidade ciclomática, pode-se constatar que todos os projectos têm um valor bastante alto. Segundo os valores originais propostos por *Thomas J. McCabe*, em 1976, que define que se os valores da complexidade estiverem acima de 51, pode-se considerar o código como *very high complexity*. Conclui-se, consequentemente, que todos os projectos apresentam uma complexidade enorme. Os seus valores variam entre os 169 e os 1091, sendo que os

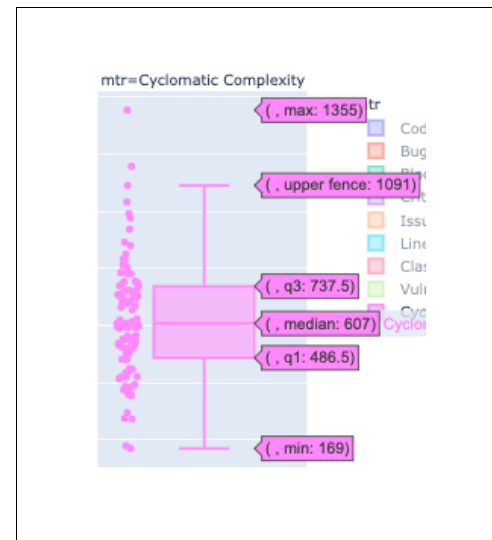


Figura 8: *Boxplot* da métrica *cyclomatic complexity*

outliers chegam a atingir o valor de 1355.

3 Refactoring

Refactoring é o processo de modificar o código fonte de uma sistema de *software* com o objectivo de melhorar a estrutura interna do código. É imperativo que esta modificação não altere o comportamento externo do sistema mas que apenas aprimore a concepção do código, melhore o seu entendimento e torne a sua manutenção mais fácil.

Inicialmente, o objectivo era automatizar este processo, tal como foi feito na análise anterior. No entanto, e após várias horas de pesquisa, o grupo não encontrou nenhuma ferramenta que permitisse usar uma linguagem de programação para automatizar o processo. Dado isto, procedeu-se à escolha de dois projectos para se aplicar *refactoring*, sendo o critério de escolha os projectos que tinham maior e menor *Technical Debt*. Esta escolha prendeu-se ao facto desta métrica ter impacto directo nos custos monetários de uma empresa de *software*. Quanto maior este valor, mais tempo uma empresa tem que dispensar a fazer *refactoring* ao código, tempo esse que poderia estar a ser usado para realizar tarefas ainda não concretizadas, tal como aconteceria se o código estivesse bem desenhado à partida. Para além disto, é espectável que um projecto com elevado *technical debt* também tenha um elevado número de *code smells* e outros problemas associados. Dado isto, pode-se afirmar que sempre que se embarca num processo demorado de *refactoring*, existe um custo monetário associado.

Desta forma, os projectos escolhidos para aplicar *refactoring* foram o **Projecto 23** cujo *technical debt* é de 17 dias e o **Projecto 88** cujo valor é de 2 dias e 6 horas.

3.1 Refactoring do Projecto 23

A Figura 9 apresenta a página *Overview* do *SonarQube* onde é possível observar as métricas consideradas mais relevantes em relação à qualidade do código.

Quando se começou a analisar a natureza dos erros do projecto, verificou-se que muitos deles eram relativos ao uso do método *System.out.println*. Uma das regras do *SonarQube* é usar um *Logger* ao contrário de *prints* para registar mensagens. No entanto, e uma vez que a *interface* do projecto é feita através da linha de comandos, o uso de *prints* é necessário. Dado isto, resolveu-se remover esta métrica, baixando o número de *code smells* de 750 para 350. Outra regra que se decidiu remover foi a complexidade ciclomática, não por não ser de elevada importância, mas porque iria demorar imenso tempo a resolver este problema e implicava uma análise muito mais detalhada do código. No entanto, foram observados quais os ficheiros que apresentam uma maior complexidade e, como era de esperar, o ficheiro que trata de criar a *interface* gráfica apresenta um valor bastante alto de 175 pois utiliza muitos blocos *switch*

para as opções dos menus, como também vários blockos *if-then-else*. Surpreendentemente, este ficheiro não é o que apresenta maior complexidade, sendo que a classe que representa o *facade* da aplicação apresenta uma complexidade de 238. Nesta classe, praticamente todos os métodos são baseados em ciclos *for* e blocos *if-then-else*, existindo métodos que contêm vários blocos deste tipo, chegando a haver um que contém seis ciclos *for*. Após uma breve análise a esta classe, pode-se afirmar que o código está demasiado complexo e o número de linhas muito alta (1083).

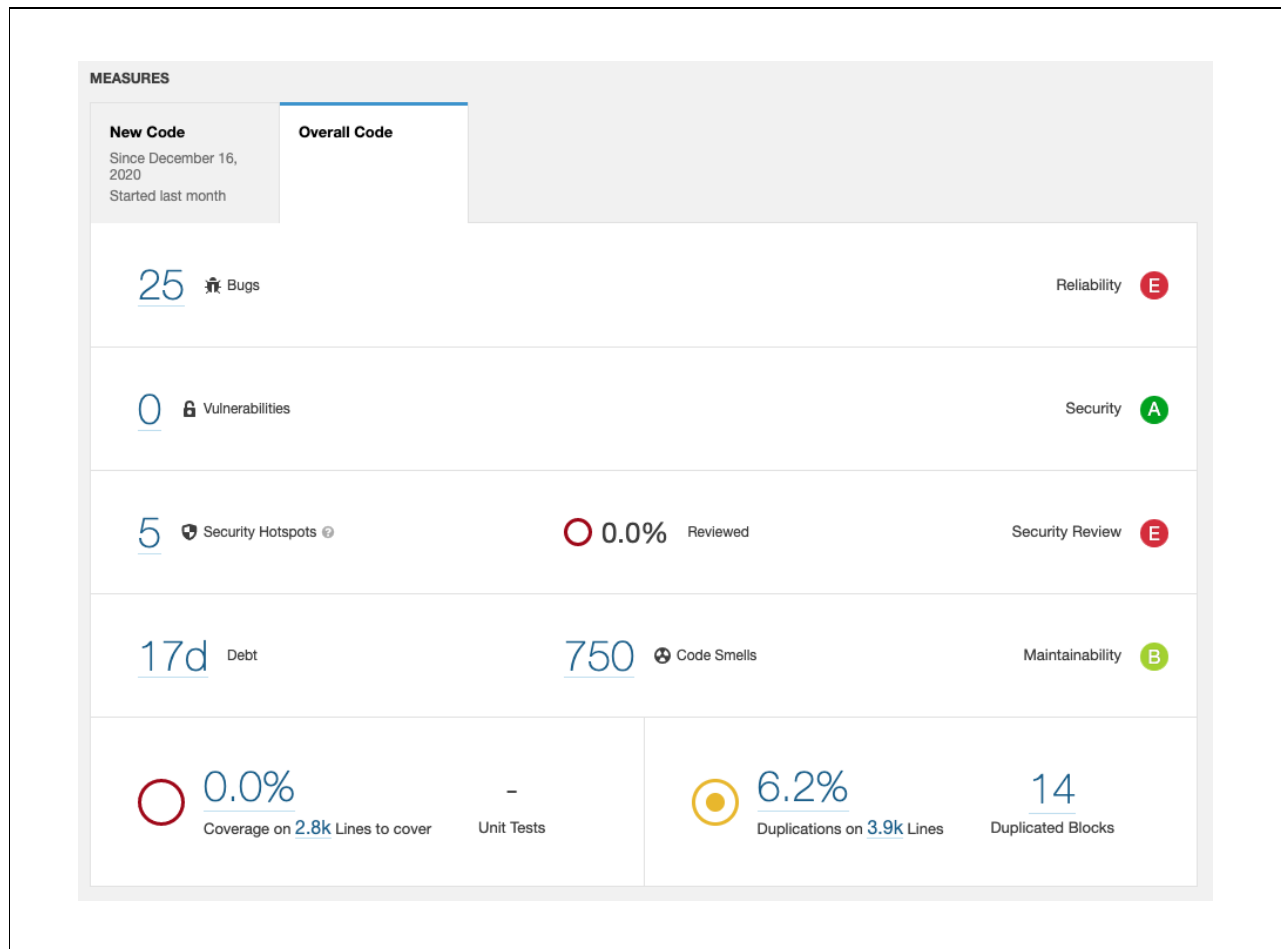


Figura 9: Métricas do Projecto 23 no *SonarQube*

A remoção destas duas regras diminuíram o número de *code smells* para 340 e o *technical debt* para 8 dias e 1 hora, e é com base nestes valores que foi feito o *refactoring* do código. A ferramenta usada para auxiliar este processo foi o *IntelliJ IDEA*, o qual contém já algumas funcionalidades de *auto refactor*.

O primeiro *code smell* tratado é referente à **declaração de variáveis não usadas**. Uma

vez removidas estas variáveis o número de *code smells* reduziu para 242 e o *technical debt* para 5 dias e 5 horas. De seguida, procedeu-se à **renomeação de variáveis** que não estavam de acordo com a expressão regular `^[a-z][a-zA-Z0-9]*$`, definida como boa prática para a linguagem *Java*. Isto baixou o número de *code smells* para 219 e o *technical debt* para 5 dias e 4 horas. Sucedeu-se à **criação de constantes para os literais duplicados** uma vez que isolar os literais torna mais fácil manter o código sendo que caso seja necessário modificar os seus valores, a alteração só é feita num sítio. Este *refactoring* diminui o número de *code smells* para 184 e o *technical debt* para 4 dias e 4 horas. A próxima alteração a ser feita foi **alterar os métodos que usavam a estrutura *Collection.size() == 0* para passar a usar o método *Collection.isEmpty()***, tornando o código mais legível. Os *code smells* baixaram para 171 e o *technical debt* manteve-se praticamente inalterado. Muitos métodos estavam a usar **construtores para *Strings***, pelo que foram removidos pois o *Java* já cria o objecto por omissão e o uso do construtor usa mais memória. Os *code smells* baixaram para 132 e o *technical debt* reduziu em 1 hora. Verificou-se que muitas classes apresentavam **código repetido**, daí usar-se a funcionalidade de extração de código do *IntelliJ IDEA* para isolar estes blocos apenas num método e alterar o bloco pela chamada do método. Os *code smells* passaram para 128 e o *technical debt* para 3 dias e 6h.

Uma das boas práticas do *Java* é não **declarar várias variáveis na mesma linha**, por exemplo, *String s1, s2, s3*, pelo que se procedeu ao *refactoring* destes casos fazendo com que os *code smells* reduzissem para 126 e o *technical debt* para 3 dias e 5 horas.

Nesta fase, deparou-se com dois tipos de *code smells* que se decidiram remover das regras. O primeiro, relativo ao uso de **clones**, removeu-se pois achou-se que não seria muito relevante, visto que há parte da comunidade que é a favor e outra contra o uso deste método. O segundo, relativo ao **número de *breaks* e *continues***, removeu-se pois era insuportável estar a fazer *refactoring* a esta métrica devido à complexidade do código e, para além disso, muitos destes operadores estão ser usados na criação da *interface* gráfica. Dado isto, o número de *code smells* diminui para 107 e o *technical debt* para 1 dia e 6 horas.

Verificou-se que os ficheiros não tinham um *package* associado, pelo que a introdução dos mesmo reduziu os *code smells* para 67 e o *technical debt* para 1 dia. De seguida, fez-se *refactoring* em relação a regras relacionadas com blocos *if-then-else*, particularmente, fazer com que tenham um ponto único de saída e dar indentação dentro dos blocos, reduzindo assim o número de *code smells* para 51 e o *technical debt* para 4 horas e 26 minutos.

Procedeu-se a um conjunto de *refactorings* que permitiram atingir zero *code smells* (assumindo que algumas regras foram desativadas por motivos já mencionados). Estes últimos *refactorings* realizados foram bastante rápidos de se fazer sendo que eram *code smells* com baixo *technical debt*, como por exemplo, renomear as classes para ir de encontro às boas práticas da linguagem, substituição de alguns métodos por outros mais eficientes, adicionar anotações de *Override* em alguns métodos, etc.

Os resultados obtidos foram bastante satisfatórios, uma vez que se conseguiram eliminar

praticamente todos os *code smells*, ficando apenas os que foram mencionados anteriormente. A Figura 10 apresenta o resultado final obtido e pode-se observar que se conseguiu diminuir em 718 o número de *code smells*.

Para concluir, pode-se constatar que o projecto escolhido tinha uma qualidade de código relativamente baixa. A sua interpretação não era muito directa e a complexidade criada com os mecanismos de controlo de fluxo tornam o código bastante confuso.

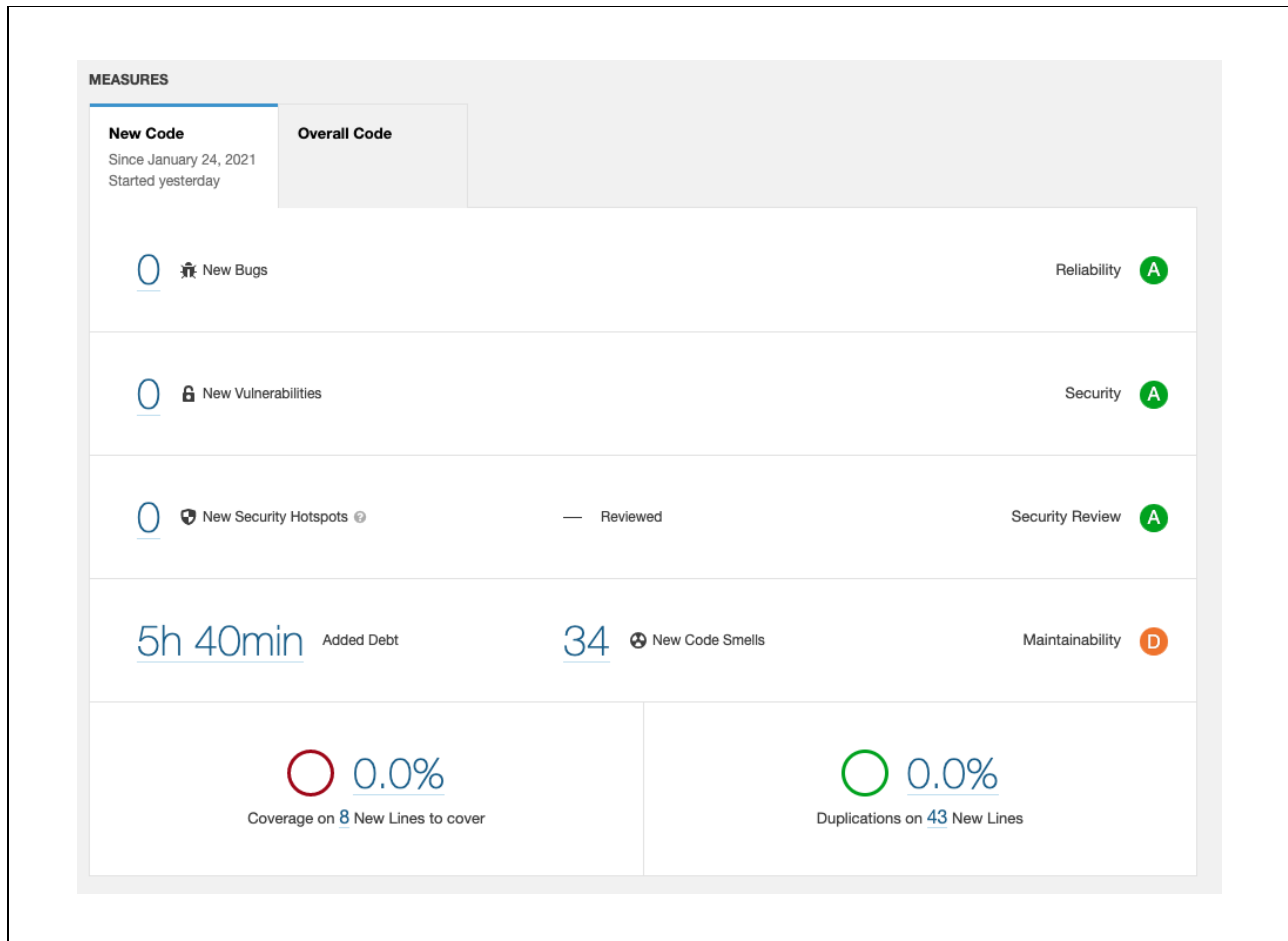


Figura 10: Métricas do Projecto 23 no *SonarQube*

3.2 Refactoring do Projecto 88

A Figura 11 apresenta a página *Overview* do *SonarQube* onde é possível observar as métricas consideradas mais relevantes em relação à qualidade do código.

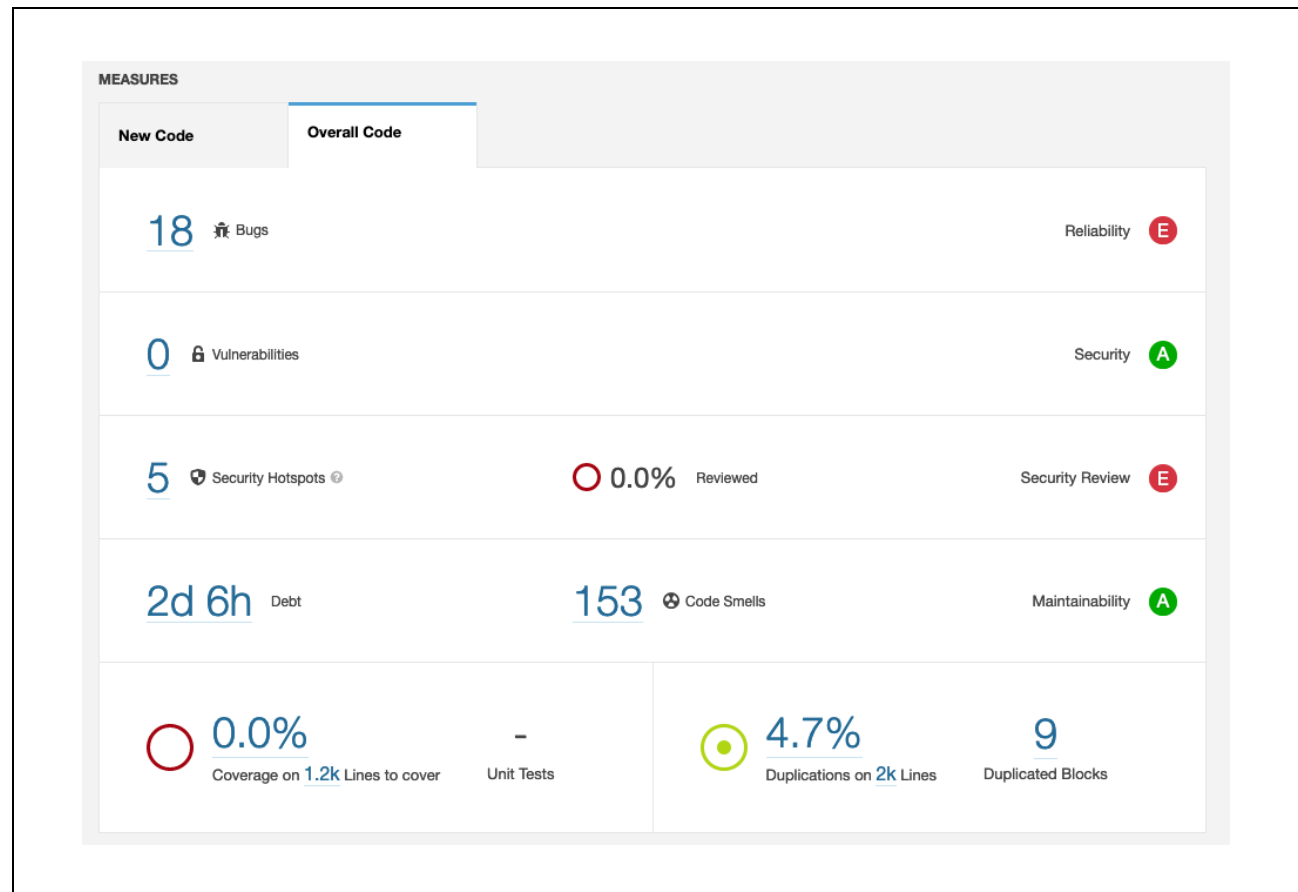


Figura 11: Métricas do Projecto 88 antes do refactoring no *SonarQube*

A abordagem adotada para o *refactoring* deste projecto foi muito similar a do projecto anterior, tendo sido desactivadas no *Sonarqube* as regras relacionadas com o uso do método *System.out.println* e complexidade ciclomática pelas razões mencionadas acima.

Feito isso, passou-se para uma análise dos restantes code smells, sendo que um dos mais frequentes estava relacionado com os nomes associados aos *packages*, pois os mesmos não obedeciam a seguinte expressão regular $\sim [a-z_]+(\backslash.[a-z_][a-z0-9_]*)*\$$. De seguida foram removidos os *comentários*, porque apesar de serem úteis para acrescentar informação adicional ao código, os mesmos são considerados *code smells* pelo *Sonarqube*. Outro *code smell* que aparece com alguma frequência está fortemente relacionado com as preferências que um pro-

gramador tem a atribuir nome às variáveis, visto que não existe um padrão, o *Sonarqube* por sua vez tem uma padrão definido sendo que a definição das variáveis deve obedecer a seguinte expressão regular `^[a-z][a-zA-Z0-9]*$`. Face ao facto de o projecto necessitar uma interface gráfica, os desenvolvedores deste projecto recorreram muita das vezes a utilização do *switch*, mas a ausência de um *default statement* ocorreu algumas vezes gerando assim mais code smells ao projecto.

Relativamente às *constantes* declaradas no projecto, o *Sonarqube* considera como *code smells* todas as declarações que não obedecem a seguinte expressão regular `^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$`.

A declaração de múltiplos *returns* aos olhos de muitos programadores é visto como má prática e o *Sonarqube* não foge à regra no que concerne a essa avaliação, sendo que à custa do *Auto-Refactoring* podemos resolver facilmente essa situação.

```
1
2  /* Antes do Auto-Refactoring */
3  public boolean reviewEncomenda(double distancia, Encomenda enc) {
4      if (distancia > this.raio * 2 || enc.getPeso() > 6)
5          return false;
6      else
7          return true;
8  }
9
10
11
12 /* Depois do Auto-Refactoring */
13
14 public boolean reviewEncomenda(double distancia, Encomenda enc) {
15
16     return distancia > this.raio * 2 || enc.getPeso() > 6;
17
18 }
```

Outra situação peculiar que aparece com uma certa frequência, é a declaração de *métodos* e *variáveis* que nunca são utilizados em qualquer parte do programa, sendo os mesmos causadores de alguns code smells. A eliminação dessas ocorrências permitem de uma forma muito fácil resolver essa situação.

A resolução das situações acima especificadas, permitiram baixar significativamente o número de *code smells* passando a ter 20 *code smells* e um *technical debt* de 7 horas e 2 minutos. Após isso virou-se as atenções para a resolução dos *Bugs* e *Security Hotspots*.

Quanto aos *Security Hotspots* os mesmos estavam relacionados com a utilização da instância de *Random* para gerar valores aleatórios, pois a mesma tem vulnerabilidades associadas. A solução encontrada para essa situação, foi recorrer a uma instância do *SecureRandom*.

```

1  /* Antes do Refactoring */
2  public double randomVelocidade() {
3      Random r = new Random();
4      return minVelocidade + (maxVelocidade - minVelocidade) * r.nextDouble();
5  }
6
7
8  /* Depois do Refactoring */
9
10 /* Variavel de instancia da classe */
11 SecureRandom r = new SecureRandom();
12
13 public double randomVelocidade() {
14     return minVelocidade + (maxVelocidade - minVelocidade) * r.nextDouble();
15 }

```

Relativamente aos *bugs* boa parte dos mesmos consistia em implementar um *finally* de modo a fechar as *streams* que estavam a ser utilizadas para fazer a leitura dos ficheiros de *log*, e os restantes estavam relacionados com a necessidade de definir o *SecureRandom* como variável de instância por uma questão de eficiência .

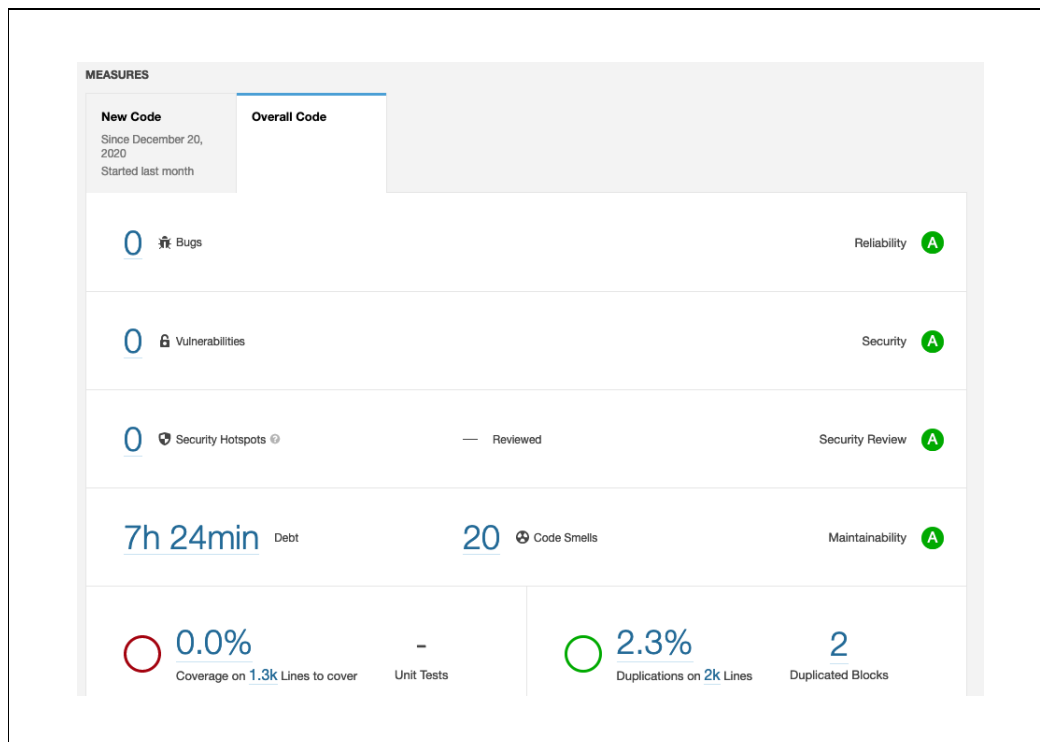


Figura 12: Métricas do Projecto 88 após refactoring no *SonarQube*

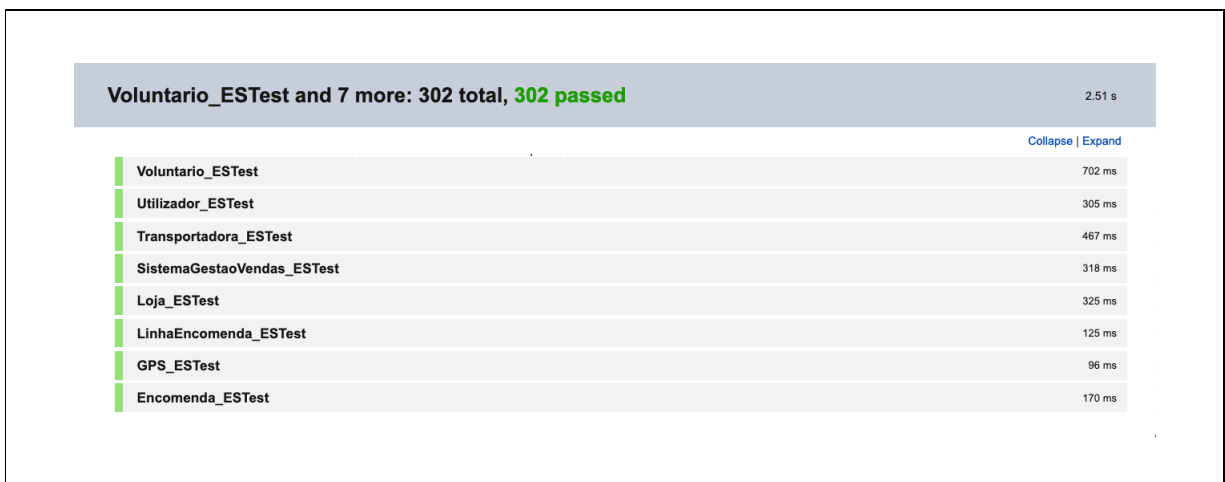
4 Teste

Não se pode garantir que todo *software* funcione correctamente, sem a presença de erros, visto que os mesmos muitas vezes possuem um grande número de estados com fórmulas, actividades e algoritmos complexos. O tamanho do projecto a ser desenvolvido e a quantidade de elementos envolvidos no processo aumentam ainda mais a complexidade. Idealmente, toda as modificações possíveis no *software* deveriam ser testadas. Entretanto, isso torna-se impossível para a ampla maioria dos casos devido à quantidade impraticável de possibilidades.

As falhas podem ser originadas por diversos motivos, tanto a nível de especificação ou pode conter simplesmente requisitos impossíveis de serem implementados devido a limitações associadas ao *hardware* ou *software*. A implementação também pode estar errada ou incompleta, como por exemplo, na ocorrência de um erro num algoritmo, sendo este o erro mais comum no desenvolvimento de *software*. Portanto, uma **falha** é o resultado de um ou mais defeitos em algum aspecto do sistema.

4.1 Cobertura

Partindo da premissa que a implementação do *refactoring* não modifica o comportamento externo do *software*, tirou-se partido do *Evosuite*, ferramenta que gera automaticamente testes unitários para *software* desenvolvido em *Java* e aplicou-se os resultados destes testes gerados a dois projectos, concretamente ao **Projecto 88 antes do *refactoring* e após o mesmo**. Em ambas as situações os testes passaram com 100% de sucesso, sendo esse o resultado esperado.



The screenshot displays the results of a test suite. At the top, a summary bar indicates 'Voluntario_ESTest and 7 more: 302 total, 302 passed' in green text, with a total time of '2.51 s'. Below this, a table lists individual test cases with their respective durations. Each test case is preceded by a green vertical bar, indicating a successful result. The table includes a 'Collapse | Expand' link on the right side.

Voluntario_ESTest and 7 more: 302 total, 302 passed		2.51 s
Voluntario_ESTest	702 ms	Collapse Expand
Utilizador_ESTest	305 ms	
Transportadora_ESTest	467 ms	
SistemaGestaoVendas_ESTest	318 ms	
Loja_ESTest	325 ms	
LinhaEncomenda_ESTest	125 ms	
GPS_ESTest	96 ms	
Encomenda_ESTest	170 ms	

Figura 13: Resultado dos testes gerados pelo Evosuite

O *Evosuite* trata por gerar testes que funcionem sempre, sendo que de seguida foi necessário fazer uma análise clínica dos testes gerados de modo a garantir a qualidade dos mesmos.

As principais medidas de um teste incluem a **cobertura** e a **qualidade**. A **cobertura** é a medida da integralidade do teste e baseia-se na cobertura de teste expressa pela cobertura de requisitos e casos de teste ou pela cobertura do código executado. A **qualidade** é uma medida de confiabilidade, estabilidade e desempenho do destino do teste. A qualidade baseia-se na avaliação dos resultados do teste e na análise dos controlos de mudanças identificados durante o mesmo.

As métricas de cobertura fornecem respostas à pergunta: "O quanto o teste é completo?". As medidas mais comuns de cobertura baseiam-se na cobertura dos requisitos de *software* e do código fonte. Basicamente, a cobertura de teste é qualquer medida de integralidade relacionada a um requisito ou aos critérios de *design* e implementação do código, como a verificação de casos de uso ou a execução de todas as linhas de código.

De seguida é apresentado a cobertura dos testes previamente definidos, sendo que podemos constatar uma cobertura ideal.

Package	Class, %	Method, %	Line, %
Model	100% (18/ 18)	100% (418/ 418)	96,9% (3746/ 3865)
Class ^			
Encomenda	100% (1/ 1)	100% (23/ 23)	88,2% (67/ 76)
Encomenda_ESTest	100% (1/ 1)	100% (43/ 43)	96,2% (353/ 367)
Encomenda_ESTest_scaffolding	100% (1/ 1)	100% (9/ 9)	100% (52/ 52)
GPS	100% (1/ 1)	100% (11/ 11)	100% (28/ 28)
GPS_ESTest	100% (1/ 1)	100% (23/ 23)	97,1% (132/ 136)
GPS_ESTest_scaffolding	100% (1/ 1)	100% (9/ 9)	100% (52/ 52)
LinhaEncomenda_ESTest	100% (1/ 1)	100% (33/ 33)	98,1% (210/ 214)
LinhaEncomenda_ESTest_scaffolding	100% (1/ 1)	100% (9/ 9)	100% (52/ 52)
Loja_ESTest	100% (1/ 1)	100% (41/ 41)	93,8% (303/ 323)
Loja_ESTest_scaffolding	100% (1/ 1)	100% (10/ 10)	100% (56/ 56)
SistemaGestaoVendas_ESTest	100% (1/ 1)	100% (19/ 19)	92% (115/ 125)
SistemaGestaoVendas_ESTest_scaffolding	100% (1/ 1)	100% (9/ 9)	100% (52/ 52)
Transportadora_ESTest	100% (1/ 1)	100% (59/ 59)	97,7% (603/ 617)
Transportadora_ESTest_scaffolding	100% (1/ 1)	100% (9/ 9)	100% (52/ 52)
Utilizador_ESTest	100% (1/ 1)	100% (34/ 34)	93,9% (186/ 198)
Utilizador_ESTest_scaffolding	100% (1/ 1)	100% (9/ 9)	100% (52/ 52)
Voluntario_ESTest	100% (1/ 1)	100% (58/ 58)	97,6% (1324/ 1356)
Voluntario_ESTest_scaffolding	100% (1/ 1)	100% (10/ 10)	100% (57/ 57)

Figura 14: Cobertura dos testes gerados

4.2 Geração de logfiles

A geração automática de ficheiros de *logs* da aplicação **TrazAqui**, foi feita à custa do *Quickcheck*. O *QuickCheck* é uma biblioteca de *software*, mais especificamente uma biblioteca

combinadora, originalmente escrita na linguagem de programação *Haskell*, projectada para auxiliar no teste de *software*, gerando casos de teste para *suites* de teste.

Os ficheiros gerados obedecem à seguinte estrutura :

1. Utilizador:

- Código do utilizador, sendo este um valor único.
- Nome do utilizador.
- Coordenadas do utilizador.

2. Voluntário:

- Código do voluntário, sendo este um valor único.
- Nome do voluntário.
- Coordenadas do voluntário.
- Raio geográfico

3. Transportadora :

- Código da transportadora, sendo este um valor único.
- Nome da transportadora.
- Coordenadas da transportadora.
- NIF da transportadora
- Raio geográfico
- Preço do transporte

4. Loja:

- Código da loja, sendo este um valor único.
- Nome da loja.
- Coordenadas da loja.

5. Encomenda:

- Código da encomenda, sendo este um valor único.
- Código do utilizador, sendo este um valor único.
- Código da loja, sendo este um valor único.
- Peso da encomenda.

- Código do produto, sendo este um valor único.
- Descrição do produto.
- Quantidade do produto.
- Preço unitário do produto.

6. Aceite:

- Código da encomenda aceite.

Feita a estrutura de dados , a etapa seguinte na realização desta tarefa consistiu em definir funções capazes de gerar valores aleatórios para a aplicação *TrazAqui* para futuras simulações reais.

O primeiro bloco de código apresentado abaixo permite criar os campos necessários para gerar a informação sobre um utilizador, onde **genName** é responsável pela criação do primeiro e último nome de um determinado utilizador, sendo que ambos valores são selecionados de duas listas previamente definidas.

O **genCodes** é responsável por gerar uma lista de valores não repetidos, que posteriormente será usada para fazer a atribuição do código de utilizadores. Relativamente à **genCoords**, é responsável por gerar um par de coordenadas sendo que a primeira componente está compreendida entre -90 e 90 (latitude) e a segunda componente compreendida entre -180 e 180. Por fim, a função **genUsers** é responsável pela geração de utilizadores do sistema.

O segundo bloco de código permite criar uma estrutura intermédia de modo a combinar o conjunto de valores gerados das entidades do sistema, sendo isto feito à custa da função **genLogFile**. Por sua vez, **genLogFile** é uma função auxiliar que é utilizada à posterior na função **genLogFiles** de modo a ter-se o controlo sobre a quantidade de ficheiros a serem gerados.

Feito isso, usa-se a função **writeLogs** para gerar concretamente os ficheiros e guardá-los num *path* previamente definido no *scope* do programa.

1. Geração de um utilizador:

```
1 firstNames :: [Name]
2 firstNames = ["Etienne", "Rui", "Mauricio", "Maria"]
3
4
5 genLastName :: Gen Name
6 genLastName = frequency [(321, return "Costa"), (596, return "Azevedo"), (777,
7     return "Salgado"), (381, return "Medvidova")]
8
9 genName :: Gen Name
10 genName = do
11     fn <- elements firstNames
12     ln <- genLastName
13     return (fn++" "+ln)
14
15 genCoords :: Gen Coords
16 genCoords = do
17     latitude <- choose(-90,90)
18     longitude <- choose(-180,180)
19     return (latitude, longitude)
20
21
22
23
24 genCodes :: [Code] -> Int -> Gen [Code]
25 genCodes codes 0 = return codes
26 genCodes [] n = do
27     code <- elements [1..10000]
28     genCodes [code] (n-1)
29 genCodes codes n = do
30     code <- elements [1..10000]
31     if elem code codes then genCodes codes n else
32     genCodes (code:codes) (n-1)
33
34 genUser :: Code -> Gen User
35 genUser code = do
36     name <- genName
37     coords <- genCoords
38     return (User code name coords)
39
40 genUsers :: [Code] -> Gen [User]
41 genUsers [] = return []
42 genUsers (h:t) = do
43     x <- genUser h
44     xs <- genUsers t
45     return (x:xs)
```

A geração das restantes entidades é bastante análoga .

2. Geração da estrutura :

```
1
2 data Structure = Structure [User] [Volunteer] [Transporter] [Shop] [Order]
   [Product] [Accepted]
3     deriving (Show)
4
5 genLogFile :: (Int,Int,Int,Int,Int,Int,Int) -> Gen Structure
6 genLogFile (nrUsers,nrVolunteers,nrTransporters,nrShops,nrOrders,nrProducts
   ,nrAcceptedds)= do
7
8     usersCodes      <- genCodes [] nrUsers
9     volunteersCodes <- genCodes [] nrVolunteers
10    transportersCodes <- genCodes [] nrTransporters
11    shopsCodes       <- genCodes [] nrShops
12    productsCodes    <- genCodes [] nrProducts
13    nifs             <- genNifs [] nrTransporters
14    ordersCodes      <- genCodes [] nrOrders
15    products         <- genProducts productsCodes
16    users            <- genUsers usersCodes
17    volunteers       <- genVolunteers volunteersCodes
18    transporters     <- genTransporters nifs
19    transportersCodes shops <- genShops shopsCodes
20    orders           <- genOrders ordersCodes usersCodes shopsCodes
21    acceptedds       <- genAcceptedds (take nrAcceptedds ordersCodes)
22    return (Structure users volunteers transporters shops orders products
   acceptedds)
23
24 genLogFiles :: Int -> (Int,Int,Int,Int,Int,Int,Int) -> Gen [Structure]
25 genLogFiles 0 _ = return []
26 genLogFiles n (nrUsers,nrVolunteers,nrTransporters,nrShops,nrOrders,
   nrProducts,nrAcceptedds) = do
27
28     h <- genLogFile (nrUsers,nrVolunteers,nrTransporters,nrShops,nrOrders,
   nrProducts,nrAcceptedds)
29
30     t <- genLogFiles (n-1) (nrUsers,nrVolunteers,nrTransporters,nrShops,
   nrOrders,nrProducts,nrAcceptedds)
31     return (h:t)
32
33 writeLogs :: Int -> [Structure] -> IO()
34 writeLogs 0 [] = return ()
35 writeLogs n (h:t) = do
36     let content' = (structureToString h)
37     let log = concat content'
38     writeFile (path++"log"++show(n)++".txt") log
39     writeLogs (n-1) t
```


3. Função principal:

```
1
2 main :: IO ()
3 main = do
4     putStrLn "Indique o numero de Logs"
5     n <- getLine
6     putStrLn "Indique o numero de Utilizadores"
7     nrUsers <- getLine
8     putStrLn "Indique o numero de Volunt rios"
9     nrVolunteers <- getLine
10    putStrLn "Indique o numero de Transportadoras"
11    nrTransporters <- getLine
12    putStrLn "Indique o numero de Lojas"
13    nrShops <- getLine
14    putStrLn "Indique o numero de Encomendas"
15    nrOrders <- getLine
16    putStrLn "Indique o numero de Produtos"
17    nrProducts <- getLine
18    putStrLn "Indique o numero de Encomendas Aceites"
19    nrAcceptedds <- getLine
20
21    let number = read n :: Int
22    let nu = read nrUsers :: Int
23    let nv = read nrVolunteers :: Int
24    let nt = read nrTransporters :: Int
25    let ns = read nrShops :: Int
26    let no = read nrOrders :: Int
27    let np = read nrProducts :: Int
28    let na = read nrAcceptedds :: Int
29
30    structures <- generate (genLogFiles number (nu,nv,nt,ns,no,np,na)
31    )
32    writeLogs number structures
33    putStrLn ("Log files successfully created !")
```

Combinando todas as funções previamente especificadas , conseguiu-se produzir uma conjunto de ficheiros *logs* com valores aleatórios.

5 Análise de Desempenho

Nesta secção serão analisados os projectos em estudo quanto ao consumo energético e tempo de execução. Para este efeito foi usado o *software RAPL*, capaz de medir estas duas métricas anteriormente mencionadas.

Visto que a única interação com os projectos é através da navegação por menus, decidiu-se

criar uma classe à parte que trata de simular um possível comportamento do sistema. A classe é responsável por ler os ficheiros de *logs* gerados na fase anterior e executar um conjunto de métodos do *facade* da aplicação.

5.1 Desempenho do Projecto 23

Tirando partido dos métodos já definidos no projecto, criou-se uma classe, designada *RAPL*, responsável por definir o seguinte comportamento:

$$\boxed{\begin{array}{l} \textit{criar utilizadores/lojas/encomendas} \rightarrow \textit{adicionar encomendas} \\ \rightarrow \textit{calcular top 10 utilizadores} \rightarrow \textit{recomendar transportadoras} \end{array}} \quad (1)$$

A funcionalidade mais complexa presente no comportamento acima é a de calcular o *top 10* de utilizadores visto que precisa de agregar dados de todos os utilizadores do sistema para proceder ao cálculo.

A tabela abaixo apresenta os valores calculados pelo *RAPL*. Em cada célula, o valor mais acima é o valor obtido antes de se aplicar *refactoring* e o valor abaixo foi obtido após aplicar o mesmo.

Tabela 1: Consumo de energia e Desempenho do Projecto 23

Refactoring	Core 1 (J)	Core 2 (J)	Tempo (s)
Antes	8.2727	7.3636	0.2185
Depois	9.4545	8.6363	0.2422

5.2 Desempenho do Projecto 88

Tirando partido dos métodos já definidos no projecto, criou-se uma classe, designada *Teste*, responsável por definir o seguinte comportamento:

$$\boxed{\begin{array}{l} \textit{carregar ficheiro} \rightarrow \textit{aceitar encomenda} \rightarrow \textit{calcular top 10 utilizadores} \\ \rightarrow \textit{calcular top 10 transportadoras} \rightarrow \textit{listar transportadoras} \end{array}} \quad (2)$$

As funcionalidades que apresentam maior custo computacional são as de calcular o *top 10* de utilizadores e transportadoras por, mais uma vez, precisarem de agregar dados de todos os utilizadores/transportadoras.

A tabela abaixo apresenta os valores calculados pelo *RAPL*. Em cada célula, o valor mais acima é o valor obtido antes de se aplicar *refactoring* e o valor abaixo foi obtido após aplicar o mesmo.

Tabela 2: Consumo de energia e Desempenho do Projecto 88

Refactoring	Core 1 (J)	Core 2 (J)	Tempo (s)
Antes	2.1818	1.8181	0.0958
Depois	2.6363	2.4376	0.0958

5.3 Análise dos resultados

Quando se observaram os resultados obtidos pelo *RAPL*, suspeitou-se que deveria haver uma anomalia algures nos métodos escolhidos para teste. Isto porque o ficheiro usado para carregar as estruturas ainda contém bastantes registos e, para além disso, nos testes, estão-se a inserir cerca de 10 000 utilizadores novos para além daqueles que já existem no ficheiro. Isto devia degradar o desempenho das operações de carregamento e cálculo do *top 10*.

Dado isto, resolveu-se inspecionar os métodos que estavam a ser usados nos testes. Após uma análise ao método que calcula o *top 10*, verificou-se que, independentemente do número de utilizadores, o resultado devolvido era sempre nulo, o mesmo acontece com os métodos que calculam os *top 10* de Voluntários e Transportadoras. Analisando o fluxo de execução do carregamento das estruturas, constatou-se que não havia nenhuma informação de encomendas a ser guardada nos utilizadores nem nas transportadoras, pelo que estes métodos que têm maior carga computacional não puderam ser testados.

Verificou-se que os valores antes e depois do *refactoring* não apresentam grande variação, pelo que se pode afirmar que neste caso, os *refactorings* feitos aos métodos usados nos testes não tiveram grande impacto no consumo de energia nem no tempo de execução dos programas.

6 Conclusão

Numa primeira fase do trabalho foram analisadas um conjunto de métricas sobre todos os projectos a analisar. Para isto foi desenvolvida uma pequena aplicação *web* que permitiu ter uma análise mais detalhada sobre o estado da qualidade de cada um dos projectos, assim como observar a qualidade de uma forma global. Isto permitiu definir um critério de escolha sobre quais projectos aplicar *refactoring*. Para além do estudo das métricas do trabalho, foi possível aprender um pouco sobre desenvolvimento *web* em *python*, bem como análise e apresentação de dados.

Dada a análise feita na primeira parte do trabalho, procedeu-se ao processo de *refactoring* dos Projectos 23 e 88, projectos que apresentam maior e menor *technical debt*, respetivamente. Nesta fase, e com a ajuda do *SonarQube* foi possível observar qual a natureza dos *code smells*, *bugs* e vulnerabilidades dos projectos. Esta fase permitiu com que se ficasse com um melhor conhecimento sobre o correcto *design* de código em *Java*. Feita a análise aos dois projectos, e comparando um com o outro, conclui-se que o projecto 23 está muito mais complexo que o projecto 88, como se esperava. Observa-se que o projecto 23 não está muito bem modelado, existindo classes bastante compridas, com métodos com elevada complexidade ciclomática. Já o projecto 88 apresenta modelação através da arquitectura *Model-View-Controller* e os métodos são mais legíveis, sem erros muito críticos.

A terceira fase do projecto consistiu em fazer a análise da cobertura dos testes gerados automaticamente. A geração de testes unitários foi feita através do *EvoSuite* e a cobertura feita através das funcionalidades do *IntelliJ IDEA*. Esta análise foi feita sobre o Projecto 88, sobre as classes da camada lógica do projecto e verificou-se que os testes cobriram 100% das classes.

Por último, foi feita a análise do consumo energético e do tempo de execução do programa. Nesta fase, observou-se que os métodos mais importantes para esta análise não estavam a funcionar, pelo que não foi possível tê-los em conta na análise do consumo energético. Com os resultados obtidos pelo *RAPL* verificou-se que o processo de *refactoring* não alterou praticamente nada o consumo energético dos projectos. Um aspecto que impediu o grupo de uma melhor análise energética dos projectos foi o facto do *RAPL* apenas correr em máquinas com o sistema *Linux* nativo. Uma vez que nenhum elemento do grupo disponha de uma máquina com este sistema, foi necessário pedir a um outro grupo para correr os testes feitos, impedindo assim de se fazer uma análise aprofundada. Seria interessante comparar os consumos energéticos dos programas alterando as estruturas de dados, isto é, uma estrutura que esteja definida como *HashMap* trocar para *TreeMap* e verificar o impacto. Outra análise interessante seria ver o impacto que alguns *red smells* têm no consumo energético.

Pode-se concluir que todas as tarefas foram concluídas com sucesso, à excepção da última devido ao entrave causado pela dependência do *RAPL* quanto ao *Linux*. Foi possível aumentar os conhecimentos sobre o *design* do código *Java* bem como *refactorings* que proporcionam

uma melhor legibilidade sobre o código. Os conhecimentos adquiridos visam ser bastante úteis para trabalhos futuros.