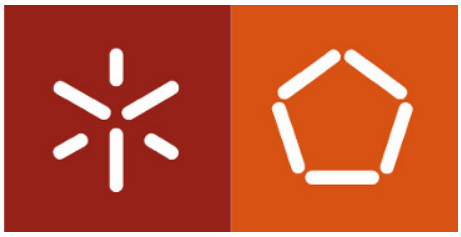
Análise e Teste de Software

Grupo nr.	10
a84577	José Pedro Silva
a85700	Pedro Costa
a84783	Pedro Rodrigues



Mestrado Integrado em Engenharia Informática
Universidade do Minho

Contents

1 Introdução

O presente relatório foi desenvolvido no âmbito da unidade curricular de Análise e Teste de Software. Este projeto consiste em avaliar, testar e refaturar trabalhos elaborados por alunos do 2º ano de Engenharia Informática no ano 2019/2020.

Este trabalho está dividido em 4 tarefas principais, sendo estas, avaliação da qualidade do código fonte dos trabalhos, refactoring das aplicações, teste das aplicações e, por fim, análise de desempenho das mesmas.

De modo a tornar o processo automático, é também pretendido realizar todas as operações mencionadas a cima em *batch*.

2 Estratégia

Nesta secção iremos apresentar brevemente a estratégia usada para abordar os principais problemas que nos surgiram ao longo do desenvolvimento do trabalho, estando estas explicadas com mais detalhe e rigor posteriormente no relatório.

Numa primeira fase, após ter sido feita uma análise superficial aos projetos, concluimos que os projetos não seguiam uma estrutura comum. Posto isto, decidimos que seria necessário rearranjar a estrutura dos projetos de modo a, mais tarde, facilitar a integração com o maven.

Após aplicarmos a estruturação referida, temos agora os projetos prontos para poderem serem integrados com o maven, e por isso, preparados para serem enviados para o sonarqube. Daqui conseguimos extrair as métricas necessárias para completar a primeira tarefa.

Estando agora completa a avaliação da qualidade de código dos projetos, podemos prosseguir para o refactoring do código. Aqui temos o objetivo de aplicar refactoring automático em massa aos projetos ao invés de aplicarmos manualmente a cada um. De modo a fazer tal coisa, após uma pesquisa extensa, recolhemos algumas aplicações que nos poderiam ser úteis, nomeadamente, AutoRefactorCli (*fork from cal101*), AutoRefactorCli (*fork from MarcoCouto*), JSparrow e, por fim, o autorefactor do editor de texto **eclipse**. É discutido na secção 2.2 a utilidade e os problemas de cada aplicação.

Na tarefa 3 temos como objetivo fazer testes às aplicações, tendo à nossa disposição aplicações como **EvoSuit** e **JaCoCo** para a geração automática de testes e **QuickCheck** para gerar os logs automáticos.

Numa fase final, resta-nos apenas realizar a análise de desempenho das aplicações. Para isso, o grupo considera que a melhor maneira de realizar esta tarefa será com recurso ao RAPL.

2.1 Estruturação de projetos

Como explicado na Estratégia, o ponto de entrada do projeto passou por uniformizar a estrutura de todos os projetos sendo o objetivo final a estrutura requirida por projetos maven.

Para isto escolhemos utilizar Haskell para gerar um script responsável por organizar cada projeto na estrutura desejada. De forma geral, a estrutura tem de ser alterada para cada projeto que não respeite a estrutura maven passa apenas por colocar todos os seus packages (e correspondentes ficheiros .java) dentro duma diretoria *src/main/java* e juntar a isto o ficheiro *pom.xml* requerido, sendo a maior parte deste trabalho manipulação de caminhos. Temos, então, duas fases:

- Haskell: No nosso programa Haskell, para cada projeto, filtramos os ficheiros .java. O próximo passo passa por criar a estrutura de packages desejada. Para este fim, basta-nos extrair o package a qual cada

ficheiro pertence (lendo a linha package do mesmo) e agrupá-los por package. A partir daqui, manipulamos strings para criar então um script bash que quando executado, replica todos os ficheiros java de cada projeto na estrutura maven, ainda sem o ficheiro *pom.xml*.

- Bash: Neste script bash define-se todo o processo de estrutução. Primeiro, executa-se o programa Haskell, executando de seguida o script que o mesmo gera. Com os projetos uniformizados, faltando apenas pôr o ficheiro *pom.xml* com a *mainClass* certa. Para isso, utilizamos *awk* para descobrir qual é a classe principal de cada projeto, substituindo a linha correspondente no *pom.xml* através do comando *sed*.

2.2 Refactoring

Para uma melhor análise dos projetos, é necessário o processo de *refactoring*, permitindo uma comparação a nível de métricas e desempenho de cada projeto pré e pós *refactor*.

Posto isto, o grupo procurou ferramentas que permitissem a realização de *refactor* automático, de forma a ser possível realizar o mesmo para todos os projetos. Foi encontrado o plugin *autorefactor* do eclipse, que contém uma versão `cli`, a qual permite a execução deste plugin no terminal.

No entanto, houveram dificuldades iniciais no processo de utilização desta ferramenta e, apesar de, após a ajuda dos docentes, ter sido possível utilizar a ferramenta, o grupo reparou que esta não realizava o *refactor* de forma correta nem completa, acabando muitas vezes por lançar *exceptions* fora do controlo do grupo. Posto isto, o projeto inclui o processo de *autorefactor*, no entanto, não pode ser considerado viável.

2.2.1 Auto refactoring

Como mencionado acima, foi utilizada a ferramenta AutoRefactorCli, que corresponde a um plugin do eclipse possível de ser corrido numa interface *command line*.

Desta forma, o processo de autorefactor está presente no módulo `refactor` do projecto, onde contém a pasta `autorefactor` contendo a ferramenta e a versão do eclipse compatível, e o ficheiro `refactors.txt` com todo o tipo de *refactor* possível de realizar através da ferramenta. Posto isto, o *script* `autorefactor.sh` copia todos os projetos já estruturados e corre a ferramenta, realizando *refactor* automático em todos os projetos.

2.2.2 Manual refactoring

Tal como referido em Refactoring, o *refactor* automático não funcionou da maneira pretendida e, por isso, o grupo decidiu realizar *refactor* manual, de modo a obter uma análise viável.

Sendo assim, após a análise de métricas, foram escolhidos 3 projetos (melhor, pior e mediano), tendo sido estes três os escolhidos para a realização de *refactor* automático.

Neste processo, foi utilizado o *refactor* do `IntelliJ` e foram revistos os *code smells* fornecidos pelo *SonarQube*.

Os projetos escolhidos foram o 2, 67 e 83. A razão desta escolha será explicada em Análise de Resultados.

2.3 Análise de métricas

De forma semelhante ao que utilizamos nass aulas da unidade curricular, é utilizado o *SonarQube* como ferramenta de análise de métricas para cada projeto. De forma a ser extrair e interpretar os valores do *SonarQube* foi escrito um script em *python* e em *R*, respetivamente.

Primeiramente, o script cria uma pasta `results`, onde irá agregar todos os resultados da análise. Em seguida, todos o tipo de projetos (*refactor* automático, *refactor* manual e originais), são enviados para o *SonarQube* e é extraída a informação dos mesmos para um ficheiro *csv* através do script *python*. As métricas extraídas são:

- bugs
- code smells
- technicall debt
- duplicated blocks
- duplicated lines density
- cyclomatic complexity

Com a utilização do *R* é possível analisar a informação contida nos ficheiros *csv*. Assim, são criados histogramas para cada uma das métricas, bem como calculados o máximo, o mínimo e a média, apresentando no final o melhor e pior projeto. A definição de melhor e pior projeto consiste no máximo e mínimo da soma de todas as métricas, respetivamente.

Após uma análise mais detalhada ao tipo de *code smells* presentes nos projetos analisados para *refactor*, decidimos ignorar o *smell* que pedia para substituir o `System.out` por um *logger*, no entanto, achamos que isto não seria um smell e que aumentava bastante o número de smells desnecessariamente.

2.4 Geração e cobertura de testes

Após uma análise da qualidade do código de cada projeto, é necessário analisar o funcionamento do mesmo, para isso, foi utilizada a ferramenta apresentada pelos docentes *Evosuite*. Apesar da possibilidade de realizar esta operação para

todos os projetos, apenas é feita nos projetos escolhidos anteriormente, uma vez que o processo é demorado e que exige desempenho computacional.

Depois de testar a aplicação, é necessário criar um relatório de cobertura de testes. Apesar dos docentes terem recomendado o uso da ferramenta *JaCoCo*, o grupo não onseguiu colocá-la a funcionar corretamente, estando constantemente a reportar 0% de cobertura de testes. Sendo assim, foi encontrada outra ferramenta *cobertura*, utilizável a partir do maven e que funciona de forma expectável.

Então, são copiados os 3 projetos escolhidos e é colocada uma nova *pom.xml* que contém o *evosuite* e *cobertura*. Para cada projeto é corrido o *evosuite* com 4 cores e 2GB de memória, diminuindo para um terço o tempo de execução da ferramenta. À medida que é gerado o conjunto de testes, os mesmos são executados e é gerado um *report* para cada um deles. No final, é executada a ferramenta *cobertura* que gera o relatório de cobertura dos testes gerados, ficando tudo guardado num pasta `target/site`. No final de todo o processo, esta pasta é copiada para os *results*.

Os relatórios apresentados estão no formato de *website*, sendo que, para cada projeto, é fornecida uma pasta contendo o relatório do resultado dos testes `surefire-report.hmtl` e um relatório completo da cobertura dos mesmos `cobertura/index.html`

2.5 Análise de consumo de energia e tempo

Assim como as análises anteriores, este processo é executado apenas para os 3 projetos escolhidos, uma vez que houve necessidade de acrescentar a possibilidade dos programas de ler de um ficheiro de logs e seria um processo massivo para todos os projetos.

Assim, cada projeto é compilado e corrido através do *RAPL* onde são medidos o consumo de energia e desempenho de cada projeto pré e pós *refactor* para 3 ficheiros logs com diferentes tamanhos.

2.5.1 Geração automática de inputs

Tal como pedido na fase 3 recorremos ao QuickCheck, mais especificamente, à sua capacidade de gerar casos de teste de forma bastante trivial devido ao facto de Gen implementar um monad.

Inicialmente, começamos por modelar os diferentes tipos que queremos gerar, nomeadamente, utilizadores, voluntários, transportadoras, lojas e produtos. Estes tipos são ADT's compostos pelas várias componentes necessárias para cada um. Toda a informação sobre estruturas de dados foi colocada num ficheiro à parte chamado *Types*.

Tendo a modelação feita passamos então à geração de valores. As componentes básicas são geradas escolhendo um valor aleatório na gama desejada através da função *choose*. De seguida, criamos geradores para cada um dos ADT's, recorrendo ao package Control.Monad e às funções *liftM* para dar lift aos construtores puros para o *monad* Gen.

Falta-nos apenas garantir que não há identificadores repetidos. Para isso, fizemos exatamente o que aprendemos nas aulas, tornando o identificador externo a cada identidade, usando depois uma função para escolher um dentro de uma pool possível de valores que é reduzida a cada escolha. A geração de pedidos é o único caso que acaba por ser bastante diferente pois passa por selecionar entidades que já foram geradas (lojas, utilizadores e produtos) previamente.

Com vista a criar logs de dimensões arbitrárias de forma simples recorremos a dois aspetos dignos de mencionar:

- **Mockaroo**: Com isto fomos capazes de gerar "bases de dados" com nomes de pessoas, nomes de empresas e produtos de mercearia aleatórios em formato *csv*. Para os utilizar precisavamos que estes valores fossem puros pelo que criamos uma função que transforma um *csv* num ficheiro Haskell com essa lista de valores. Idealmente, deviamos ter recorrido ao package GenT para evitar esse fix desnecessário mas apenas nos apercebemos da existência deste monad transformer depois de terminar esta fase.

- Tipo **Dataset**: Criamos um tipo chamado Dataset que define para cada entidade a quantidade a ser gerada. Com isto conseguimos criar uma função independente destes valores, tornando-se extremamente fácil criar datasets de tamanho diferente.

A parte de geração encontra-se no ficheiro *Generator.hs*. Criamos essencialmente 3 logs diferentes, sendo um de tamanho pequeno, outro médio e um grande de forma a testar os projetos com cargas variadas.

2.5.2 RAPL

Para tirar partido do RAPL fizemos uma pequena alteração à versão utilizada nas aulas uma vez que esta está preparada para correr um executável e nós queremos correr um jar. Assim, alteramos o ficheiro *main.c* de forma a ter presente a alteração acima. Adicionamos ainda um argumento para poder escolher o nome do ficheiro de saída.

De forma a tornar os resultados obtidos pelo RAPL o mais realista possíveis, o grupo decidiu criar uma nova classe Main para os projetos que íamos testar contemplando apenas o carregamento dos logs para o sistema e a operação mais crítica requirida, nomeadamente, obter o top 10 de utilizadores que mais utilizam o sistema. Optamos por utilizar apenas esta pois era a única que dependia diretamente dos logs, permitindo-nos comparar como cada aplicação reage aos diferentes volumes de dados gerados.

3 Análise de Resultados

A execução completa da análise é toda automatizada e requer a execução do *SonarQube* em *background*.

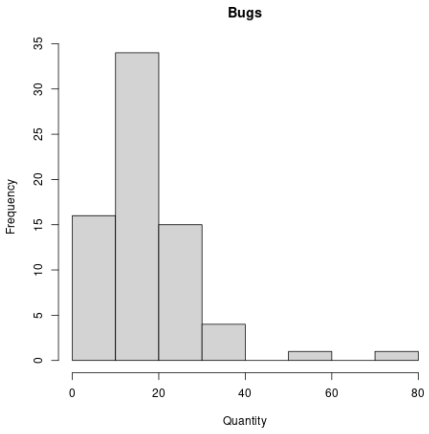
Esta execução é feita através do script `ats.sh` que apresenta duas possibilidades (com e sem *auto refactor*).

Para a obtenção de uma análise mais viável, o processo foi executado sem *autorefactor*, no entanto, para ativar esta funcionalidade basta acrescentar o argumento `refactor`, no entanto, a ferramenta responsável pelo processo de *refactor* não funciona corretamente e demora largos minutos.

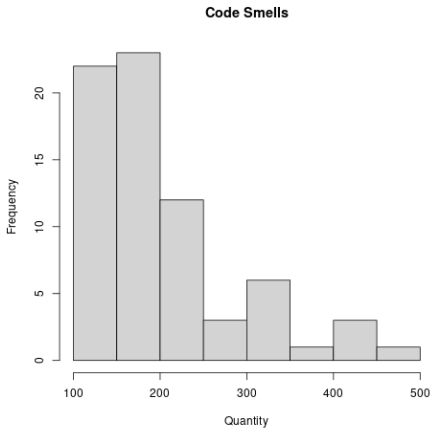
No final, todos os ficheiros necessários para a análise de resultados encontram-se na pasta `results`

3.1 Análise de métricas de projetos originais

Em `results/graphs`, podemos observar histogramas, relativos às métricas dos projetos originais, bem como, na diretoria anterior, o ficheiro `metric_analysis` que contém o máximo, mínimo e média para cada métrica, bem como o melhor e o pior projeto.



Bugs

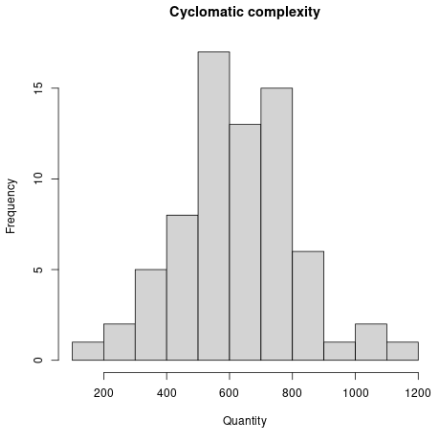


Code smells

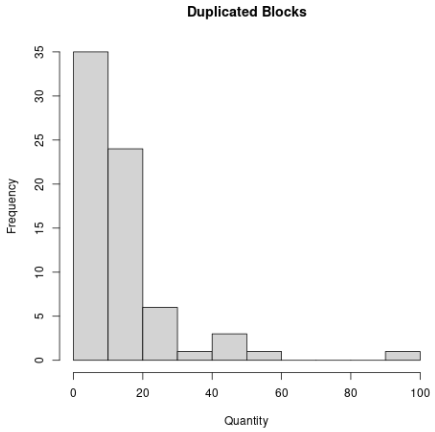
Ao observar a figura acima, podemos analisar que o número de bugs concentra-se, na maioria dos trabalhos, entre 10 e 20, sendo 18 o número médio. Pelo gráfico, podemos observar que existem alguns projetos distantes em relação aos outros tendo em conta o número de bugs, sendo um deles o projeto 39, que contém o maior número de bugs - 74. O menor número é 0, pertencente ao projeto 5.

No que toca a code smells, podemos afirmar que a maioria dos projetos contém entre 100 e 200 code smells, no entanto, há uma variedade muito grande no que toca a esta métrica. Sendo que o maior número é de 453 do projeto 98,

no entanto, não está isolado, uma vez que existem vários projetos entre com o número de code smells entre 300 e 500. O projeto 63 contém 101 code smells, representando o menor valor entre todos, sendo 203 o valor médio.



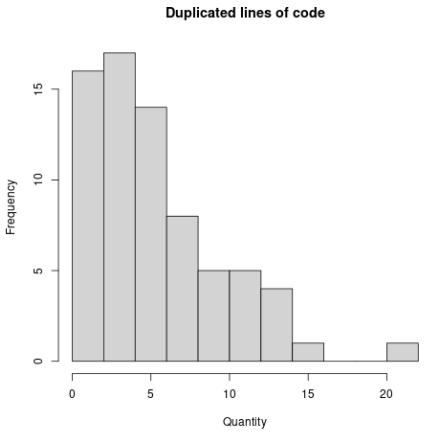
Complexidade ciclomática



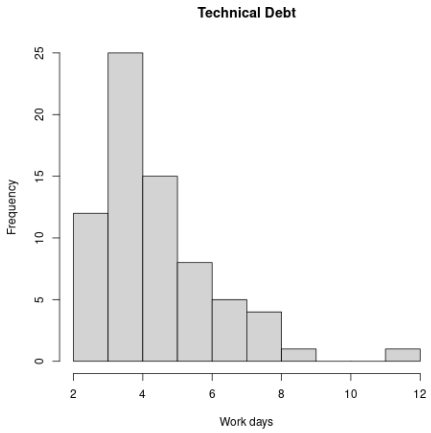
Blocos duplicados

Analisando o gráfico da complexidade ciclomática, podemos concluir que a maioria dos projetos contém entre 500 a 800 caminhos de execução diferentes. No entanto, tanto há mais projetos com valores mais baixos do que mais altos, sendo que, o maior número de complexidade ciclomática é do projeto 78 com 1158, contrastando com o menor do projeto 13 com 169. Observando estes números, podemos afirmar que o projeto 78 contém quase 7x mais complexidade ciclomática do que o 13. A média ronda os 626.

Na figura ao lado, é-nos apresentado o gráfico corresponde aos blocos de código duplicado e, podemos facilmente observar, que a maior parte dos projetos contém entre 0 a 20, sendo o valor médio 14. No entanto, há alguns projetos que fogem a esta média, sendo que o pior e de forma completamente isolada, é o projeto 78 com 100 blocos de código duplicado. Por outro lado, existem 5 projetos com 0 (3, 53, 57, 63 e 76).



Percentagem de linhas duplicadas



Technical debt

A percentagem de linhas duplicadas, apesar de haver uma grande distribuição, concentra-se maioritariamente entre 0 e 6, no entanto, o projeto 13 contém cerca de 22%, representando a maior percentagem entre todos, em contraste com os mesmos 5 de cima, com 0% (3, 53, 57, 63 e 76). O número médio ronda os 5% Por fim, o *technical debt* está concentrado entre 3 e 4 dias de trabalho (8 horas por dia), sendo que existe um projeto (78) que apresenta cerca de 11 dias, consituindo o número máximo, ao contrário dos projetos 13, 15 e 63 que seriam precisos cerca 2 dias, sendo este o valor minimo encontrado. O valor médio é cerca de 5 dias.

3.2 Comparação original vs refactored

Nas seguintes tabelas apresentam-se os valores das métricas medidas. De modo a facilitar a comparação entre sem refactor vs. com refactor, apresentamos os valores no formato *'sem refactor - com refactor'*.

Projeto	**Bugs**	**Code Smells**	**Technical Debt**
2	6 — 3	190 — 52	3.5 — 2.1
67	8 — 9	134 — 93	4.1 — 3.5
83	30 — 30	423 — 137	7.8 — 5.2

Projeto	**Duplicated Blocks**	**Duplicated line density**	**Complexity**
2	18 — 18	8.4 — 9.7	360 — 316
67	4 — 4	1.5 — 1.7	596 — 535
83	19 — 19	5.3 — 5.9	880 — 772

Analisando as tabelas acima representadas, podemos concluir que há uma melhoria na maior parte das métricas, no entanto, podemos também concluir que embora o refactor ajude bastante no melhoria do código, não consegue

transformar código mau em código bom, isto é, se a lógica implementada no programa estiver errada, não é possível melhorar a mesma através de refactoring. Com isto, é expectável que nem todas as métricas diminuam drasticamente de valor, no entanto é sempre esperado alguma melhoria.

3.3 Testes e cobertura de testes

3.3.1 Projeto 2

Tests	Errors	Failures	Skipped	Success Rate	Time
282	0	0	0	100%	2.446

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
(default)	17	46% 546/1177	23% 62/266	2.228
All Packages	17	46% 546/1177	23% 62/266	2.228

Classes in this Package	Line Coverage	Branch Coverage	Complexity
ComparatorUser	100% 4/4	100% 2/2	3
DataBase	95% 79/83	71% 10/14	0
Empresa	100% 35/35	N/A N/A	1
Encomenda	100% 125/125	100% 6/6	0
EncomendaJaExisteException	100% 4/4	N/A N/A	1
EncomendaNaoExisteException	100% 4/4	N/A N/A	1
LinhaEncomenda	98% 51/52	100% 12/12	1.389
Loja	100% 27/27	N/A N/A	1
Main	5% 31/553	4% 8/174	6.115
NaoSeEncontraRaioException	100% 4/4	N/A N/A	1
Parser	28% 37/129	3% 1/30	2.125
Perfil	97% 44/45	92% 13/14	1.533
Ponto2D	77% 35/45	71% 10/14	1.533
Transporte	96% 30/31	N/A N/A	1
UserNaoExisteException	100% 4/4	N/A N/A	1
Utilizador	100% 19/19	N/A N/A	1
Voluntario	100% 13/13	N/A N/A	1

3.3.2 Projeto 67

Tests	Errors	Failures	Skipped	Success Rate	Time
550	0	0	0	100%	2.335

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	31	62% 1049/1679	28% 173/605	2.076
MVC	3	86% 73/84	85% 12/14	2.444
MVC.Comparators	3	100% 23/23	100% 12/12	4
MVC.Controller	1	6% 31/491	2% 10/335	6.24
MVC.Exceptions	2	100% 10/10	N/A N/A	1
MVC.Models	2	59% 170/288	46% 66/142	3.189
MVC.Models.BaseModels	12	96% 497/514	97% 47/48	1.195
MVC.Models.Catalogs	5	89% 206/230	44% 22/50	0
MVC.Views	3	100% 39/39	100% 4/4	1.167

Report generated by Cobertura 2.1.1 on 25/01/21 21:09.

3.3.3 Projeto 83

Tests	Errors	Failures	Skipped	Success Rate	Time
592	0	1	0	99.831%	5.414

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
(default)	29	45% 1317/2901	46% 364/781	1.515
All Packages	29	45% 1317/2901	46% 364/781	1.515

Classes in this Package	Line Coverage	Branch Coverage	Complexity
BDGeral	23% 34/147	3% 1/30	1.487
BDGeralInterface	N/A N/A	N/A N/A	1
BDLojas	43% 28/64	38% 7/18	0
BDProdutos	98% 58/59	94% 32/34	0
BDTransportes	89% 96/107	77% 48/62	0
BDUtilizador	73% 38/52	62% 10/16	0
BDVoluntarios	97% 99/102	71% 40/56	0
ComparaQuantidadePair	100% 3/3	100% 2/2	3
DistanceCalculator	100% 2/2	N/A N/A	1
EmpresaTransportes	32% 88/269	18% 18/100	0
Encomenda	99% 110/111	95% 21/22	0
EncomendaNotFoundException	100% 2/2	N/A N/A	1
EncomendasAceites	96% 25/26	100% 8/8	0
Input	89% 65/73	83% 10/12	3
LinhaEncomenda	98% 51/52	100% 22/22	1.867
Loja	98% 86/87	100% 30/30	0
LojaNotFoundException	100% 2/2	N/A N/A	1
Main	0% 0/8	N/A N/A	1
Pair	96% 26/27	100% 6/6	1.3
Parse	34% 71/207	8% 6/70	0
ProductNotFoundException	100% 2/2	N/A N/A	1
TransporteNotFoundException	100% 2/2	N/A N/A	1
TrazAquiController	0% 0/1034	0% 0/169	0
TrazAquiView	100% 116/116	N/A N/A	1
UserNotFoundException	100% 2/2	N/A N/A	1
Utilizador	63% 52/82	23% 6/26	0
UtilizadorSistema	98% 59/60	91% 11/12	1.35
Voluntario	99% 198/200	95% 82/86	2.087
VoluntarioNotFoundException	100% 2/2	N/A N/A	1

3.4 Consumo de energia e tempo - original vs refactored

3.4.1 Projeto 2

Para o caso dos logs *small e medium*, como se pode ver abaixo, o refactor acaba por não ter qualquer impacto positivo. Isto deve-se, potencialmente, tanto

ao facto de o volume de dados ser reduzido como ao facto de os refactorings aplicados não terem em vista melhorar o consumo de energia diretamente.

2 — 2_refactored	Energia (J)	% CPU	Time (sec)
small	2.65 — 2.94	1.9 — 2.1	0.263 — 0.291
medium	4.04 — 4.32	3.44 — 3.65	0.234 — 0.258
big	8.11 — 7.89	5.88 — 5.75	0.8 — 0.78

3.4.2 Projeto 67

67 — 67_refactored	Energia (J)	% CPU	Time (sec)
small	2.31 — 1.99	1.66 — 1.44	0.228 — 0.197
medium	6.28 — 6.04	5.19 — 4.94	0.420 — 0.424
big	24.48 — 23.24	17.8 — 16.86	2.427 — 2.304

3.4.3 Projeto 83

No projeto 83 tivemos alguns problemas uma vez que este gera erros ao ler o ficheiro de logs. O formato está correto, no entanto, pelo que analisamos, este projeto forçava condições que não deviam ser forçadas, pelo que o único caso em que obtemos resultados fieis foi no log *small*. Nos *medium* foram lançadas exceções pelo que os resultados acabam por não representar nada muito realista. Este projeto relevou-se bastante fraco, tanto que nem sequer conseguimos terminar a execução do RAPL para o log *big*, daí não ser apresentado na tabela.

83 — 83_refactored	Energia (J)	% CPU	Time (sec)
small	29.22 — 26.24	21.07 — 18.96	2.896 — 2.599
medium	389.01 — 422.53	293.78 — 311.18	34.977 — 39.975

4 Conclusão

Concluído o trabalho, o grupo considera que atingiu os objetivos e extras indicados pelos docentes, tendo criado um sistema de análise de projetos java, de forma completamente automática e independente, capaz de apresentar resultados em forma de gráficos e tabelas.

Atentando aos resultados das análises e em conformidade com a unidade curricular, confirmamos que o processo de *refactoring*, melhora a qualidade do código e o consumo de energia dos projetos. Apesar do *refactor* exercido neste projeto ter sido simples, fácil e com pouco investimento de tempo, podemos verificar essas melhorias, principalmente em computações mais pesadas.

Apesar do grupo ter atingido os objetivos, sentiu dificuldade em certos momentos e tarefas proposta. Uma delas é a falta de ferramentas de *autorefactor*, tendo sido necessário realizar *refactor* manualmente. Outra dificuldade foi a capacidade de testar projetos de forma automática, ou seja, sem interação humana, uma vez que os projetos propostos estão unicamente preparados para utilização numa interface gráfica.

Em suma, o grupo conclui que nem sempre o facto de um projeto funcionar implica que este execute a sua função da melhor maneira, e não implica ainda que este possa ser melhorado através de refactoring e testing, pois estas operações não alteram o comportamento do programa.