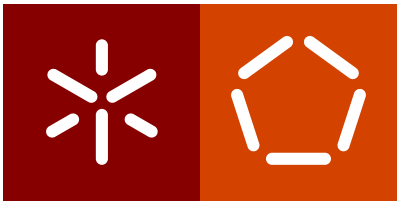


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Análise e Teste de Software - O Sistema de Software ”TrazAqui!”

Bruno Veloso (a78352)
João Mota (a80791)
Rafael Lourenço (a86266)
Grupo 4

Janeiro 2021

RESUMO

O presente relatório foi desenvolvido com o intuito de expor e explicar o processo de análise e teste de software aplicado a um conjunto de projectos reunidos pela equipa docente da unidade curricular de Análise e Teste de Software.

CONTEÚDO

1

INTRODUÇÃO

À medida que o tempo passa, a tecnologia ganha mais e mais importância em todo o mundo, sendo talvez o setor com mais impacto na economia mundial na atualidade. Com esta evolução, vem mais responsabilidade e perfecionismo para quem trabalha nesta área.

A tecnologia vem de braços dados com o do desenvolvimento de *software* evoluindo em conjunto com este. A exigência deste último obriga a que existam mecanismos que permitam a constante análise e teste do *software*. Tendo isto em conta, nesta unidade curricular procura-se que os alunos sejam capazes de estudar métodos e conheçam ferramentas para a análise e teste de sistemas de software. Estes métodos devem permitir a verificação de código e localização de falhas inerentes a este.

De forma a serem consolidados os conhecimentos obtidos nas aulas, a equipa docente apresentou um trabalho prático aos alunos no qual são atribuídos diversos projetos, previamente realizados por alunos do curso Mestrado Integrado em Engenharia Informática da Universidade do Minho na unidade curricular de Programação Orientada aos Objetos, como o tema "Traz Aqui!", que pretendia replicar um sistema de entregas ao domicilio.

Propôs-se então, para a unidade curricular de "Análise e Teste de Software", a preparação de projetos em *java* ,de forma a, analisar as métricas de *software* dos mesmos.De seguida, os projetos sofrerão de um *refactoring*, de forma a, melhorar essas métricas (diminuir os *smells*, *debt*, *bugs*, etc). Mais tarde será feito a geração de testes da aplicação, verificar o *coverage* dos testes e criar os ficheiros de *logs*. Por fim, irá ser feito uma análise do consumo de energia da aplicação, verificando se o consumo de energia é influenciado conforme a diminuição de *code smells*.

2

CONTEXTUALIZAÇÃO TEÓRICA

2.1 DEFINIÇÕES

Previamente à realização deste trabalho, foram estudados diversos tópicos imprescindíveis à elaboração do mesmo.

- ***Code Smells***: Porções de código que não representam erros, mas qualquer tipo de problema mais profundo no código. Estes problemas podem tornar o código desenvolvido difícil de entender, e consequentemente, mais difícil de progredir. Além disso, a existência de *code smells* aumenta a probabilidade de futuros erros.
- ***Program Refactoring***: Medidas tomadas para melhorar o código desenvolvido previamente, de forma a tornar este mais legível ao nível do *design* sem que se altere o comportamento externo.
- ***Technical Debt*** Define-se como o tempo necessário para corrigir *code smells* presentes no software desenvolvido.
- ***Java Red Code Smells*** *Smells* que afetam a eficiência energética.
- **Testes unitários** Técnica da área de teste de *software* que permite a análise de pequenas porções de código, facilitando a análise deste como um todo.
- **Complexidade Ciclomática**: Métrica de *software* usada para indicar a complexidade de um *software*. Enumera a quantidade de caminhos independentes no código.

2.2 FERRAMENTAS UTILIZADAS

- ***SonarQube***:Plataforma que permite a inspeção da qualidade de código. Esta plataforma analisa automaticamente o código fornecido, detetando *code smells*, *bugs* e vulnerabilidades em termos de segurança. Concretamente no projeto desenvolvido, são utilizados projetos em *Maven* por ser aceite pela plataforma [9].
- ***IntelliJ***: Ambiente de desenvolvimento de programas JAVA, recomendado pela equipa docente devido às suas ferramentas de análise, *refactoring* e teste de código.
- **Ferramenta de Refactor do IDEA IntelliJ**: Ferramenta do IDEA **IntelliJ** que reestrutura automaticamente o código pretendido, melhorando-o e tornando-o de mais fácil manutenção.
- **JUnit**: *Framework* que permite criar e executar testes unitários na linguagem de programação JAVA. Estes testes conseguem testar porções de código, permitindo assim, uma análise mais detalhada do código fonte de um programa. O IntelliJ, ambiente de desenvolvimento utilizado, suporta a criação destes testes unitários [2].
- ***Jupyter Notebook***: Aplicação *Web* de *open source* que permite a criação, no caso concreto da utilização para o projeto atual, de indicadores de forma a analisar alguns aspetos relevantes.
- **JSparrow** Ferramenta que faz *autorefactor* automático [5].
- **jRAPL-Running average power limit** *Framework* de *profiling* de programas JAVA a correr em CPUs com um suporte de limite de potência média de execução [3].
- **SonarLint** plugin do intellij para detetar smells [8].
- **Evo Suite** Extensão que permite encontrar *code smells* no código[4].
- **Auto Refactor** *Plugin* do Eclipse para fazer *auto refactor* de programas JAVA [6].

3

RESOLUÇÃO DAS TAREFAS

3.1 **tarefa 1**: QUALIDADE DO CÓDIGO FONTE DA APLICAÇÃO TRAZAQUI

Nesta tarefa, pretende-se analisar a qualidade do código das diversas aplicações fornecidas pela equipa docente. Para este fim, recorreu-se à utilização de ferramentas como o *SonarQube*, IDEA's do JAVA, entre outras. Métricas de software foram também utilizadas para encontrar *code smells*.

3.1.1 *Preparação dos projetos*

Numa primeira fase, foi necessária a preparação dos projetos de forma a ser possível aplicar as técnicas estudadas. Como tal, e dado que o *SonarQube* apenas aceita projetos *maven*, foi necessário fazer uma restruturação de todos os projetos para uma estrutura compatível com o *maven*. Deste modo, para cada projeto foram criadas as diretorias **src/main/java**, sendo, de seguida copiado todas as classes e os packages para essa mesma diretoria, uma vez que, caso isto não fosse feito, o código fonte não era detetado pelo *SonarQube*. Após este processo foi criada uma POM.XML, template base para configuração do maven. Foi então necessário detetar todas as dependências para todos os projetos, e após várias iterações de compilação, concluímos este processo. Por último, recorrendo a comandos *bash/shell*, foi possível inserir este *template*, de forma automática, em cada um dos projetos. Neste *template*, para além das dependências, encontram-se os dados de cada projeto, nomeadamente, a indicação do ficheiro que contém o método **main**. Para tal, foi necessário percorrer todas as classes para encontrar qual deles tinha o método main, sendo posteriormente colocada no template. Durante todo este processo existiram alguns casos exceção, que tiveram de ser tratados de forma individual.Todos os projetos processados estão localizados na pasta **Proj_sonar**.
Desde o primeiro momento, o grupo decidiu cumprir a tarefa extra proposta de automatizar estas tarefas de forma a simplificar a análise, contrariando a ideia de analisar cada projeto manualmente.

A existência de dois sistemas operativos distintos (Linux e MacOS) entre elementos do grupo gerou a necessidade de adaptar o código em *bash*, de forma a permitir a execução dos *scripts*.

Sucintamente, nesta primeira **sub-fase** aplicou-se a cada um dos 99 projetos o seguinte fluxo de ações:

1. Criação das diretorias estruturadas para cada um.
2. Copiar tanto o projeto, como a pom.xml *template* criada para a respetiva diretoria criada previamente.
3. Encontrar o método **main** dentro de cada projeto, utilizando expressões regulares.
4. Adicionar o nome do ficheiro que contém a main, na respetiva pom.xml com a *tag* <**mainClass**>.
5. Tratamento de casos exceção.

Dado que foram disponibilizados todos os projetos que foram desenvolvidos pelos alunos de POO de 2019/2020, naturalmente surgiram alguns que tiveram de ser tratados de forma particular.

- Quatro dos projetos (38,62,75,80) encontravam-se sem qualquer ficheiro, sendo por isso impossível encontrar a classe contendo o método *main*.
- Alguns projetos continham vários métodos *main*. Nestes, foi analisado manualmente o método que faria mais sentido ser colocado como *mainClass* na pom.xml, sendo criado um caso para cada um desdes projetos no *script*.
- Dois projetos (26,96) encontravam-se com a denominação do ficheiro .java, diferente do nome da classe em si, sendo por isso, de forma automatizada, alterados os nomes destes ficheiros para os nomes das classes respetivas.
- O projeto 35 teve necessidade de várias alterações: Adicionar um *package* a cada classe; Remover um ficheiro existente relacionado com a compilação no ambiente de desenvolvimento *BlueJ*; Remover métodos chamados que eram inexistentes; Remover parâmetros de funções que na sua definição não receberam quaisqueres parâmetros; Alterar um método inexistente por uma mensagem; Adicionar um método "Set"necessário.
- O projeto 60 necessitou do *import* "view.InterfaceGeral",uma vez que, essa classe estava instanciada.
- O projeto 42 não apresentava um método "main".Após uma análise minuciosa ,descobrimos que o método "menu"deveria ser a "main", devido ao seu contexto.

Após estas **correções**, todos os projetos, exceto aqueles que não contêm qualquer classe java, são compiláveis pelo *maven*, estando assim prontos para ser adicionados ao *sonarqube*.

3.1.2 *SonarQube*

De seguida foi criado um novo *script* em *bash* com o intuito de automatizar as tarefas de iniciar o servidor do *SonarQube*, gerar um *token* necessário para a adição dos projetos *MAVEN*

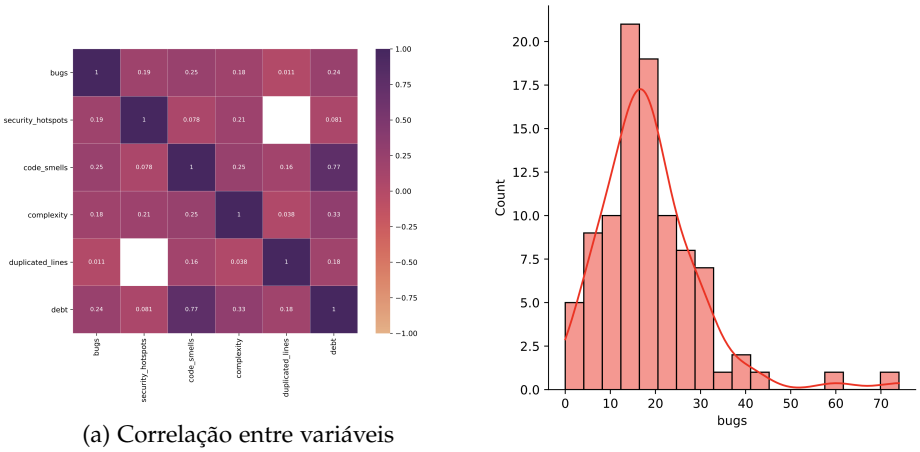
à plataforma e a compilação/execução dos mesmos. Após inicia o servidor,usando a *WEB API* do *SonarQube* e através do comando *curl*, executamos os pedidos *http* necessários para a geração do *token*.Para isso são feitos 2 passos, que consistem em eliminar o último *token* criado, e de seguida criar um *token* novo para a inserção dos projetos na plataforma. Caso o *token* eliminado não exista o resultado é ignorado.

Deste modo, para cada um dos 99 projetos foram executados os comandos *mvn compile* e *mvn package*.

Logo após ser efetuada a compilação, em cada iteração é criado um projeto no *SonarQube* relativo ao projeto em questão nessa iteração.

Com os projetos devidamente colocados no *SonarQube*, foi criado um novo *script bash* que, novamente, de forma automatizada, permitiu a extração dos dados dos campos considerados mais relevantes: **Número de *Bugs***; **Número de *Security hotspots***; **Número de *Code Smells***; **A percentagem de código duplicado**; **Débito**.

De forma a se ter uma visão global dos resultados obtidos, foram criados alguns indicadores, recorrendo à ferramenta *jupyter notebook*. Estes indicadores encontram-se presentes nas figuras 1a, 1b e 2;



(a) Correlação entre variáveis

(b) Nº de Bugs mais comum

Figura 1: Análise da qualidade do código

No gráfico 1a conseguimos visualizar a matriz de correlação , que permite relacionar as variáveis em análise. Uma correlação muito próxima de 1 significa que quanto maior for o crescimento de uma das variáveis maior vai ser o crescimento da outra (crescimento proporcional). Por outro lado, caso seja próxima de -1 significa um crescimento inversamente proporcional, ou seja, quanto maior o crescimento de uma componente maior vai ser o decréscimo da outra. Quando a correlação é 0 significa que não existe qualquer relação entre essas duas componentes. Sendo assim, é possível verificar que a componente que afeta mais o tempo necessário para a correção total do código é os *code smells*, sendo seguida da complexidade ciclomática(nº de caminhos de execução independentes no *control flow graph*) e os *bugs*.

O histograma 1b é uma relação entre o numero de bugs e quantos projetos se encontram nesse intervalo.É de realçar que o nº médio de *bugs* se encontra nos **17,5**.

Devido ao número de *code smels* ser elevado decidimos mostrar para cada projeto o nº correspondente de *code smells*,figura 2.Além disso é possível ver, através do realce feito pela cor atribuída a cada projeto, quantos dias seriam necessários para a correção dos mesmos, sendo a mais escura equivalente a 15 dias e a mais clara equivalente a 5 dias.

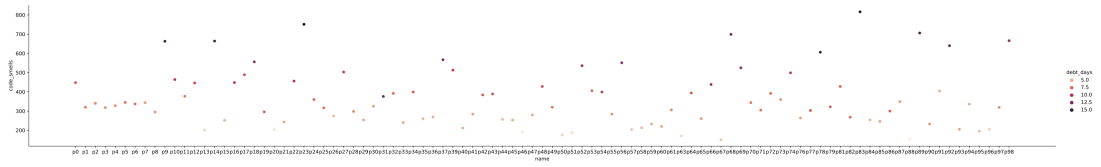


Figura 2: Nº de Code Smells por projeto

3.2 tarefa 2: refactoring da aplicação trazaqui

Nesta segunda tarefa, pretende-se a utilização de ferramentas de *autorefactor*, como o **Auto-Refactor**, ferramentas incluidas no IDE (intellij) para este propósito, ou o **jSparrow**. Estas ferramentas identificam *bad* e *red smells* permitindo a modificação de porções de código para resolver alguns destes problemas, bem como tornar o código mais legível. Importante notar, que esta tarefa não altera o comportamento externo do código. Gera exatamente o mesmo resultado que era obtido antes da tarefa de *refactoring*, apenas melhora a estrutura interna deste. Este processo tem muitas vantagens no que diz respeito ao desenvolvimento de qualquer *software*, pois possibilita uma leitura mais simples e rápida por parte de qualquer pessoa que queira reutilizar, ou continuar o desenvolvimento do *software*, o que naturalmente reduz o tempo de trabalho por parte de qualquer programador. *Refactoring* também ajuda a localizar possíveis *bugs* existentes no código, que poderão vir a dar problemas no futuro.

Com o intuito de realizar algo automático nesta tarefa, fizemos um *refactor* que é aplicado a quase todos os projetos, pois existem alguns projetos que não eram possíveis de refractar devido às limitações dos comandos shell, nomeadamente o comando *grep* não aceita caminhos, armazenados em variáveis, para diretorias diretorias com espaços / carateres especiais, sendo que ficaria separado na ocorrência desses carateres. Para resolver estes casos tinha-se de tratar individualmente, dando um *refactor* em todas as diretorias com esse problema, sendo de seguida necessário refractar todos as linhas que contivessem o *package* em as classes java. Visto que o intuito desta tarefa não é esse, simplesmente ignoramos esses casos.
Este *refactor* consiste em substituir todos os "System.out.prints"por "logger.log", visto que, é um dos *code smells* mais **frequente**. Para esta finalidade, tivemos que procurar por todos os ficheiros onde existiam "System.out.prints"e para isso usamos mais um vez expressões regulares com os comandos **grep** e **awk**. De seguida foi feito o *import* necessário e criada uma variável de classe "logger"que permitia a execução desses "prints". Por último fomos a esses ficheiros e substituímos todas as ocorrências dos "prints"pelos "loggers", utilizando o comando **sed**.

Os resultados obtidos foram os seguintes, apresentados na figura 3.

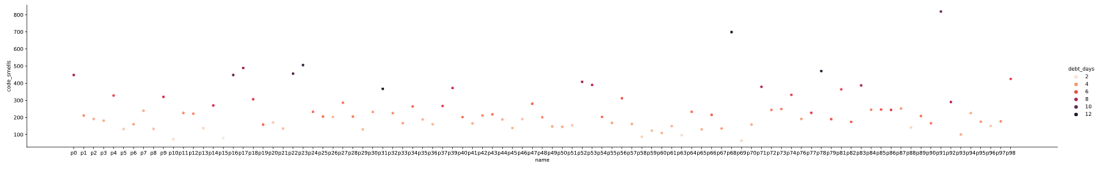


Figura 3: Nº de Code Smells após o *refactor* automático

Neste gráfico é possível verificar que a maioria dos projetos se encontra entre os **150** e **250** *codesmells*, levando assim a uma dispersão de menor magnitude comparado com o anterior, e para além disso, o *techinal debt* ficou praticamente entre os **2** dias a **5** dias, para os projetos

refractados.Uma vez que, na figura 2 a maioria dos projetos encontrava-se ente os **300** e os **500** *code smells*, torna-se óbvio que o *refactor* automático obteve resultados generosos.

De forma a ser feita uma análise mais minuciosa e aumentar a qualidade do *refactor* aplicado, com a indicação da equipa docente que se podia seleccionar um projeto dos noventa e nove fornecidos, o grupo acabou por decidir escolher dois dos projetos analisados na tarefa1, sendo estes os projetos 23 e 83. Esta escolha foi feita tendo por base os indicadores criados na tarefa anterior e apresentados na tabela que se segue.Estes mostraram que os projetos continham um elevado número de bugs (25 e 30 respetivamente), falhas de segurança (5 e 16 respetivamente), grande número de *code smells* (754 e 816 respetivamente) e sobretudo um grande valor de débito quando comparado com os outros projetos no geral (17 e 16 dias respetivamente) 2.1

Métricas de Software Utilizadas para a escolha		
Métrica	Projeto Nº23	Projeto Nº83
Bugs	25	30
Falhas de segurança	5	16
Code Smells	754	816
Complexidade Ciclomática	832	880
Linhas Duplicadas (%)	6.2	5.2
Débito (dias)	17	16

Como é visível pela tabela apresentada, a razão da escolha de estes dois projetos para a presente e consequentes tarefas deve-se à existência de um grande número de *bugs* e principalmente um elevado número de *code smells*. Dado isto, decidimos aplicar em primeiro lugar o **autorefactor**, um *plugin* do *eclipse* disponível para realizar o *refactor* automático, sendo apenas necessário selecionar o código que se pretende e aplicar o *refactoring*.Após este passo, executamos o **JSparrow** para estes projetos, uma vez que, tem um funcionamento similar ao do *autorefactor*. No entanto, o resultado não foi muito significativo, devido à versão *free* deste *plugin* apenas oferecer **15** regras de *refactoring*. Dado que, a versão paga tem **91** regras, provavelmente iríamos obter melhores resultados.
Deste modo, fomos a todas as classes e selecionamos todas as linhas, de forma a fazer um **refactor** total apresentando os resultados foram os seguintes:

Métricas após o autorefactor		
Métrica	Projeto Nº23	Projeto Nº83
Bugs	25	30
Falhas de segurança	5	16
Code Smells	**745**	**764**
Complexidade Ciclomática	832	**852**
Linhas Duplicadas (%)	6.2	**4.6**
Débito (dias)	17	**15.4**

Por último, utilizamos o *refactor* do *intelij IDEA* com o auxilio do *plugin-SonarLint*, que nos ajuda a detetar *smells* sem ter de recorrer ao sonarqube, porém, para fazer *refactor* tem de se ir especificamente à parte do código que se pretende fazer *refactor*, o que complica o uso em vários projetos. Por isso, apenas foi aplicado no projeto 23, de forma a, minimizar o valor obtido anteriormente em cada métrica. Os resultados foram os seguintes:

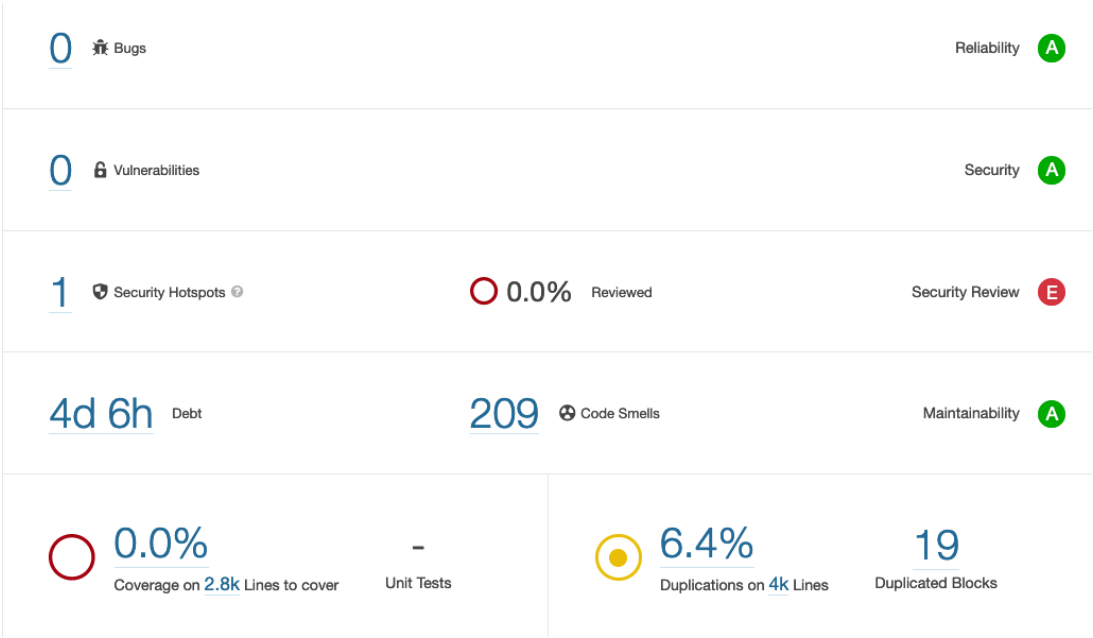


Figura 4: Overview final do projeto 23

Dado que, muitos dos *code smells* são repetidos, decidimos que não era necessário resolver-los todos, visto que , o processo de resolução é sempre idêntico.De seguida, apresentamos uma lista do que foi atualizado:

- Diretorias com letras maiúsculas passaram a minúsculas;
- Definir constantes em vez de usar a string com aspas;
- Usar *StringBuilder* em vez de concatenação pelo operador "+";

- Remoção/junção condições do *if-statements*;
- Remover de alguns *throws/try catch*;
- Remover expressões redundantes;
- Remover variáveis que nunca eram lidas;
- Adição de *finally* aos *try's* que não foram removidos;
- Usar o *SecureRandom* em vez do *Math.random*;
- Remover *imports* que não são usados;
- Resolver erros de possíveis valores nulos nas variáveis;
- Remover construtores de *strings*;
- Usar o método *equals* para comparações de strings;
- Usar métodos mais eficientes, como por exemplo, o *isEmpty* em vez do *size*.
- Diminuição da complexidade;
- Remoção de clones nos *get's* e *set's* ;

3.3 tarefa 3: teste das aplicações trazaqui

Uma das partes mais importantes no desenvolvimento de uma aplicação é o teste da mesma. Os testes são a melhor forma que o programador tem de testar o funcionamento da aplicação, bem como, verificar se os resultados dos métodos são os esperados. Para testar o bom funcionamento da mesma é necessário que hajam dados *(logs)*, que auxiliam o programador a verificar o seu bom funcionamento, tendo estes *logs* que ser gerados. Tendo isto em mente, foi desenvolvida a tarefa 3, sendo que, esta está dividida em três partes, que são:

- Gerar automaticamente testes unitários para a aplicação TrazAqui;
- Analisar a cobertura dos testes;
- Gerar automaticamente ficheiros *logs* para a aplicação TrazAqui.

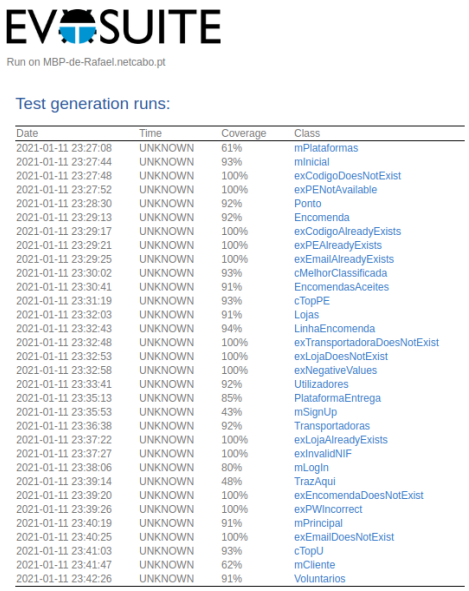
3.3.1 *Geração de testes*

De forma a ser possível gerar automaticamente testes unitários, foi usado o sistema *EvoSuite*.

Nesta parte, o *EvoSuite* foi aplicado, via terminal, aos projetos 23 e 83 em paralelo. Para ser aplicado a estes projetos, é necessário que o utilizador tenha uma pasta com os *jar's* do *EvoSuite*. Caso não os tenha, o *script* que executa a tarefa 3 irá fazer *download* dos mesmos. Para além disto, o *script* irá perguntar ao utilizador se quer que o *coverage* dos testes unitários, gerados pelo *EvoSuite*, sejam fornecidos em *csv* ou *html*. Após a execução do *script* da tarefa 3, é de esperar que dentro de cada projeto tenha uma pasta com todos os testes unitários gerados automaticamente, e outra pasta com o *coverage* total dos testes unitários, em que irá estar em *csv* ou *html*, conforme escolhido pelo utilizador.

```java
/*
 * This file was automatically generated by EvoSuite
 * Mon Jan 11 23:30:41 GMT 2021
 */


import org.junit.Test;
import static org.junit.Assert.*;
import static org.evosuite.runtime.EvoAssertions.*;
import org.evosuite.runtime.EvoRunner;
import org.evosuite.runtime.EvoRunnerParameters;
import org.junit.runner.RunWith;

@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true, useVFS = true, useVNET = true, resetStaticState = true, separateClassLoader = true)
public class EncomendasAceites_ESTest extends EncomendasAceites_ESTest_scaffolding {

  @Test(timeout = 4000)
  public void test00()  throws Throwable  {
      EncomendasAceites encomendasAceites0 = new EncomendasAceites("");
      EncomendasAceites encomendasAceites1 = new EncomendasAceites(encomendasAceites0);
      assertTrue(encomendasAceites1.equals((Object)encomendasAceites0));
  }

  @Test(timeout = 4000)
  public void test01()  throws Throwable  {
      EncomendasAceites encomendasAceites0 = new EncomendasAceites("V#;?m");
      String string0 = encomendasAceites0.getCodEncomenda();
      assertEquals("V#;?m", string0);
  }
```

Figura 5: Teste unitário gerado pelo *EvoSuite*.



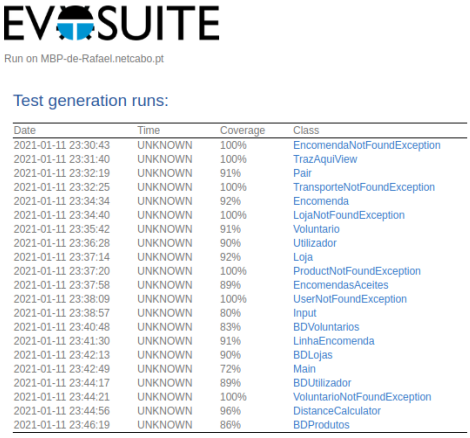
EVOSUITE

Run on MBP-de-Rafael.netcabo.pt

Test generation runs:

Date	Time	Coverage	Class
2021-01-11 23:27:08	UNKNOWN	61%	mPlataformas
2021-01-11 23:27:44	UNKNOWN	93%	mInicial
2021-01-11 23:27:48	UNKNOWN	100%	exCodigoDoesNotExist
2021-01-11 23:27:52	UNKNOWN	100%	exPENotAvailable
2021-01-11 23:28:30	UNKNOWN	92%	Ponto
2021-01-11 23:29:13	UNKNOWN	92%	Encomenda
2021-01-11 23:29:17	UNKNOWN	100%	exCodigoAlreadyExists
2021-01-11 23:29:21	UNKNOWN	100%	exPEAlreadyExists
2021-01-11 23:29:25	UNKNOWN	100%	exEmailAlreadyExists
2021-01-11 23:30:02	UNKNOWN	93%	cMelhorClassificada
2021-01-11 23:30:41	UNKNOWN	91%	EncomendasAceites
2021-01-11 23:31:19	UNKNOWN	93%	cTopPE
2021-01-11 23:32:03	UNKNOWN	91%	Lojas
2021-01-11 23:32:43	UNKNOWN	94%	LinhaEncomenda
2021-01-11 23:32:48	UNKNOWN	100%	exTransportadoraDoesNotExist
2021-01-11 23:32:53	UNKNOWN	100%	exLojaDoesNotExist
2021-01-11 23:32:58	UNKNOWN	100%	exNegativeValues
2021-01-11 23:33:41	UNKNOWN	92%	Utilizadores
2021-01-11 23:35:13	UNKNOWN	85%	PlataformaEntrega
2021-01-11 23:35:53	UNKNOWN	43%	mSignUp
2021-01-11 23:36:38	UNKNOWN	92%	Transportadoras
2021-01-11 23:37:22	UNKNOWN	100%	exLojaAlreadyExists
2021-01-11 23:37:27	UNKNOWN	100%	exInvalidNIF
2021-01-11 23:38:06	UNKNOWN	80%	mLogIn
2021-01-11 23:39:14	UNKNOWN	48%	TrazAqui
2021-01-11 23:39:20	UNKNOWN	100%	exEncomendaDoesNotExist
2021-01-11 23:39:26	UNKNOWN	100%	exPWIncorrect
2021-01-11 23:40:19	UNKNOWN	91%	mPrincipal
2021-01-11 23:40:25	UNKNOWN	100%	exEmailDoesNotExist
2021-01-11 23:41:03	UNKNOWN	93%	cTopU
2021-01-11 23:41:47	UNKNOWN	62%	mCliente
2021-01-11 23:42:26	UNKNOWN	91%	Voluntarios

Figura 6: *Coverage report* gerado para o projeto 23 através do *Evosuite*.



EVOSUITE

Run on MBP-de-Rafael.netcabo.pt

Test generation runs:

Date	Time	Coverage	Class
2021-01-11 23:30:43	UNKNOWN	100%	EncomendaNotFoundException
2021-01-11 23:31:40	UNKNOWN	100%	TrazAquiView
2021-01-11 23:32:19	UNKNOWN	91%	Pair
2021-01-11 23:32:25	UNKNOWN	100%	TransporteNotFoundException
2021-01-11 23:34:34	UNKNOWN	92%	Encomenda
2021-01-11 23:34:40	UNKNOWN	100%	LojaNotFoundException
2021-01-11 23:35:42	UNKNOWN	91%	Voluntario
2021-01-11 23:36:28	UNKNOWN	90%	Utilizador
2021-01-11 23:37:14	UNKNOWN	92%	Loja
2021-01-11 23:37:20	UNKNOWN	100%	ProductNotFoundException
2021-01-11 23:37:58	UNKNOWN	89%	EncomendasAceites
2021-01-11 23:38:09	UNKNOWN	100%	UserNotFoundException
2021-01-11 23:38:57	UNKNOWN	80%	Input
2021-01-11 23:40:48	UNKNOWN	83%	BDVoluntarios
2021-01-11 23:41:30	UNKNOWN	91%	LinhaEncomenda
2021-01-11 23:42:13	UNKNOWN	90%	BDLojas
2021-01-11 23:42:49	UNKNOWN	72%	Main
2021-01-11 23:44:17	UNKNOWN	89%	BDUtilizador
2021-01-11 23:44:21	UNKNOWN	100%	VoluntarioNotFoundException
2021-01-11 23:44:56	UNKNOWN	96%	DistanceCalculator
2021-01-11 23:46:19	UNKNOWN	86%	BDProdutos

Figura 7: *Coverage report* gerado para o projeto 83 através do *Evosuite*.

3.3.2 *Coverage dos testes*

Esta sub-tarefa tem como objetivo verificar a cobertura dos testes unitários criados pelo *EvoSuite*. Tal como na primeira sub-tarefa, serão usados os projetos 23 e 83, e para ser possível verificar a cobertura dos testes, é necessário importar o código dos projetos e os testes unitários para o *IntelliJ*. Após estes estarem no *IntelliJ*, e de forma a ser possível correr os mesmos, é necessário importar a *framework JUnit* e o *standalone jar file* do *EvoSuite*. Encontra-se um pdf na pasta do código chamado "EvoSuite_run_tests.pdf", em que explica o processo todo para correr os testes do *EvoSuite*.

Como forma de ser possível verificar a cobertura dos testes, foi usado a *coverage tool* do *IntelliJ*, que depois irá gerar um *html* com os resultados de cobertura, mostrando as percentagens que os testes cobrem das classes, métodos e linhas.

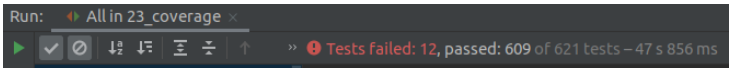


Figura 8: Testes unitários que passam e falham para o projeto 23.

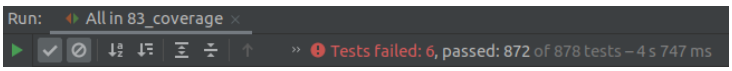


Figura 9: Testes unitários que passam e falham para o projeto 83.

Nas figuras seguintes serão apresentadas as percentagens de classes, métodos e linhas que são cobertos pelos testes:

Package	Class, %	Method, %	Line, %
<empty package>	100% (34/ 34)	100% (369/ 369)	52.9% (1477/ 2791)

Class	Class, %	Method, %	Line, %
App	100% (1/ 1)	100% (16/ 16)	9.1% (98/ 1074)
Encomenda	100% (1/ 1)	100% (23/ 23)	100% (82/ 82)
EncomendasAceites	100% (1/ 1)	100% (9/ 9)	100% (26/ 26)
LinhaEncomenda	100% (1/ 1)	100% (15/ 15)	100% (49/ 49)
Lojas	100% (1/ 1)	100% (22/ 22)	100% (74/ 74)
PlataformaEntrega	100% (1/ 1)	100% (28/ 28)	96.8% (90/ 93)
Ponto	100% (1/ 1)	100% (10/ 10)	100% (28/ 28)
Transportadoras	100% (1/ 1)	100% (14/ 14)	100% (50/ 50)
TrazAqui	100% (1/ 1)	100% (82/ 82)	67.8% (358/ 528)
Utilizadores	100% (1/ 1)	100% (21/ 21)	100% (69/ 69)
Voluntarios	100% (1/ 1)	100% (7/ 7)	100% (22/ 22)
cMelhorClassificada	100% (1/ 1)	100% (2/ 2)	100% (8/ 8)
cMelhorCusto	100% (1/ 1)	100% (2/ 2)	100% (10/ 10)
cTopPE	100% (1/ 1)	100% (2/ 2)	100% (8/ 8)
cTopU	100% (1/ 1)	100% (2/ 2)	100% (8/ 8)
exCodigoAlreadyExists	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exCodigoDoesNotExist	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exEmailAlreadyExists	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exEmailDoesNotExist	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exEncomendaDoesNotExist	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exInvalidNIF	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exLojaAlreadyExists	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exLojaDoesNotExist	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exNegativeValues	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exPEAlreadyExists	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exPENotAvailable	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exPWIncorrect	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
exTransportadoraDoesNotExist	100% (1/ 1)	100% (1/ 1)	100% (2/ 2)
mCliente	100% (1/ 1)	100% (30/ 30)	71.5% (143/ 200)
mInicial	100% (1/ 1)	100% (2/ 2)	100% (15/ 15)
mLogIn	100% (1/ 1)	100% (9/ 9)	93.3% (42/ 45)
mPlataformas	100% (1/ 1)	100% (18/ 18)	72.5% (116/ 160)
mPrincipal	100% (1/ 1)	100% (28/ 28)	98.2% (109/ 111)
mSignUp	100% (1/ 1)	100% (14/ 14)	43.8% (46/ 105)

Figura 10: Cobertura gerada pelo *IntelliJ* para o projeto 23.

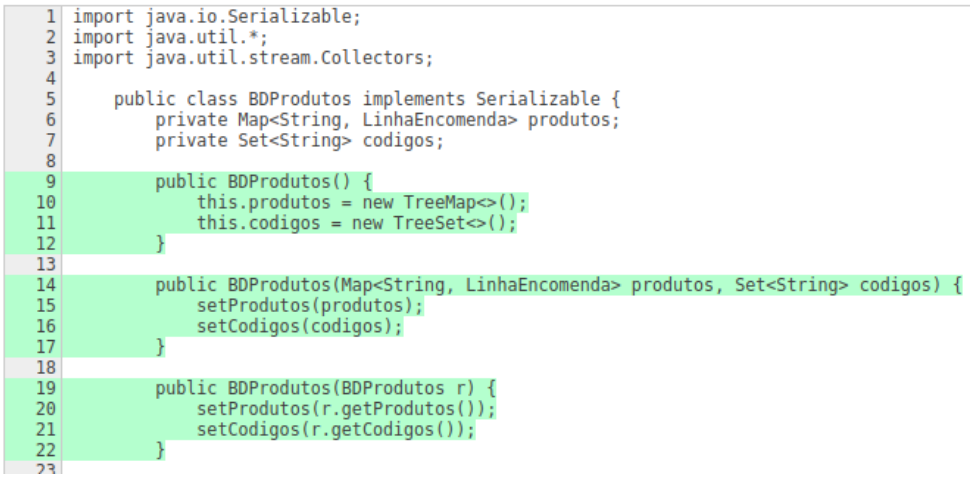
Package	Class, %	Method, %	Line, %
<empty package>	100% (28/ 28)	99.1% (438/ 442)	61.2% (1782/ 2914)

Class	Class, %	Method, %	Line, %
BDGeral	100% (1/ 1)	100% (39/ 39)	92.1% (139/ 151)
BDLojas	100% (1/ 1)	100% (20/ 20)	100% (66/ 66)
BDProdutos	100% (1/ 1)	100% (16/ 16)	100% (61/ 61)
BDTransportes	100% (1/ 1)	100% (24/ 24)	99.1% (113/ 114)
BDUtilizador	100% (1/ 1)	100% (18/ 18)	100% (55/ 55)
BDVoluntarios	100% (1/ 1)	100% (25/ 25)	96.3% (103/ 107)
ComparaQuantidadePair	100% (1/ 1)	100% (2/ 2)	100% (4/ 4)
DistanceCalculator	100% (1/ 1)	100% (2/ 2)	100% (2/ 2)
EmpresaTransportes	100% (1/ 1)	100% (50/ 50)	94.5% (257/ 272)
Encomenda	100% (1/ 1)	100% (30/ 30)	100% (112/ 112)
EncomendaNotFoundException	100% (1/ 1)	100% (1/ 1)	100% (1/ 1)
EncomendasAceites	100% (1/ 1)	100% (9/ 9)	100% (27/ 27)
Input	100% (1/ 1)	100% (7/ 7)	91.8% (67/ 73)
LinhaEncomenda	100% (1/ 1)	100% (15/ 15)	100% (52/ 52)
Loja	100% (1/ 1)	100% (21/ 21)	100% (82/ 82)
LojaNotFoundException	100% (1/ 1)	100% (1/ 1)	100% (1/ 1)
Main	100% (1/ 1)	100% (2/ 2)	77.8% (7/ 9)
Pair	100% (1/ 1)	100% (10/ 10)	100% (26/ 26)
Parse	100% (1/ 1)	100% (21/ 21)	42.4% (89/ 210)
ProductNotFoundException	100% (1/ 1)	100% (1/ 1)	100% (1/ 1)
TransporteNotFoundException	100% (1/ 1)	100% (1/ 1)	100% (1/ 1)
TrazAquiController	100% (1/ 1)	66.7% (8/ 12)	5.4% (55/ 1026)
TrazAquiView	100% (1/ 1)	100% (33/ 33)	100% (116/ 116)
UserNotFoundException	100% (1/ 1)	100% (1/ 1)	100% (1/ 1)
Utilizador	100% (1/ 1)	100% (19/ 19)	100% (78/ 78)
UtilizadorSistema	100% (1/ 1)	100% (19/ 19)	100% (61/ 61)
Voluntario	100% (1/ 1)	100% (42/ 42)	100% (204/ 204)
VoluntarioNotFoundException	100% (1/ 1)	100% (1/ 1)	100% (1/ 1)

Figura 11: Cobertura gerada pelo *IntelliJ* para o projeto 83.

É possível ver que os testes cobriram as *classes* todas e quase os métodos todos, tirando alguns métodos no projeto 83. No entanto nota-se também que cobrir as linhas através dos testes gerados se torna um maior desafio para o *EvoSuite*, ao qual este não conseguiu ultrapassar. No

report gerado pelo *IntelliJ* é possível entrar em cada *class* e ver as linhas que realmente foram **cobertas**, ficando estas sublinhadas a verde.



```
1  import java.io.Serializable;
2  import java.util.*;
3  import java.util.stream.Collectors;
4
5      public class BDProdutos implements Serializable {
6          private Map<String, LinhaEncomenda> produtos;
7          private Set<String> codigos;
8
9          public BDProdutos() {
10             this.produtos = new TreeMap<>();
11             this.codigos = new TreeSet<>();
12         }
13
14         public BDProdutos(Map<String, LinhaEncomenda> produtos, Set<String> codigos) {
15             setProdutos(produtos);
16             setCodigos(codigos);
17         }
18
19         public BDProdutos(BDProdutos r) {
20             setProdutos(r.getProdutos());
21             setCodigos(r.getCodigos());
22         }
23
```

Figura 12: Cobertura das linhas para uma *class*.

3.3.3 *Geração de ficheiros de logs automáticos*

Esta sub-tarefa corresponde à geração de ficheiros de *logs* automáticos. Para esta geração automática, foi usado para tal a biblioteca de *Haskell* chamada *QuickCheck* [7].

Segundo o ficheiro exemplo de *logs* fornecido, este contém uma nomenclatura predefinida em que esta é igual para todos os projetos (que foram observados). Os dados a serem gerados contém informação relativamente a: utilizadores da aplicação, voluntários, transportadoras de encomendas, lojas, encomendas, sendo que, estas são constituídas por produtos, em que estes estão vinculados a uma loja e utilizador existente e por fim, se a encomenda foi aceite. Após executar o gerador de dados, este irá pedir o nome do ficheiro para os armazenar (não sendo necessário a criação do ficheiro antes), bem como, o número de utilizadores, voluntários, transportadoras, lojas e encomendas, de forma a gerar a quantidade de dados que o utilizador quer.

Quanto à informação a ser gerada para a entidade ”utilizador”, foi possível ver que este precisava de um identificador, um nome e a sua posição com coordenadas x e y. Para todos os identificadores foi estabelecido um valor mínimo de 1 e máximo de 100000, em que este valor é concatenado com a letra ”u”, no caso dos utilizadores. Em relação aos nomes, este é composto por nome próprio e apelido, em que estes são escolhidos de forma aleatória,a partir de uma lista de nomes e apelidos predefinida. As figuras seguintes apresentam informação relativamente à geração de cada utilizador:

```
--gerar posicao X
genPosX :: Gen PosX
genPosX = choose(-99 :: Float, 99 :: Float)

--gerar posicao Y
genPosY :: Gen PosY
genPosY = choose(-99 :: Float, 99 :: Float)
```

Figura 13: Gerar posição do utilizador.

```
--gerar identificador
genIdentificador :: Gen Identificador
genIdentificador = do a <- choose(1 :: Int ,100000 :: Int)
                      return ("u"++show(a))
```

Figura 14: Gerar identificador do utilizador.

```
--gerar Nomes
genNome :: Gen Nome
genNome = do a <- elements nomes
             b <- elements apelidos
             return(a ++ " " ++ b)
```

Figura 15: Gerar nome do utilizador.

```
--gerar utilizadores
genUtilizador :: Gen Utilizador
genUtilizador = do a <- genIdentificador
                   b <- genNome
                   c <- genPosX
                   d <- genPosY
                   return (Utilizador a b c d)
```

Figura 16: Gerar utilizador.

Quanto a gerar voluntários, vimos que estes são acompanhados de um identificador, concatenado com a letra "v"para designar voluntário.Este tem um nome, como o utilizador normal, tem uma posição x e y e um raio onde este atua.
Nas figuras seguintes vemos a geração de um voluntário:

```
--gerar raio
genRaio :: Gen Raio
genRaio = choose(1:: Int, 300 :: Int)
```

Figura 17: Gerar raio.

```
--gerar voluntarios
genVoluntario :: Gen Voluntario
genVoluntario = do a <- genIdentificadorV
                   b <- genNome
                   c <- genPosX
                   d <- genPosY
                   e <- genRaio
                   return (Voluntario a b c d e)
```

Figura 18: Gerar voluntário.

No que toca a gerar transportadoras, estas são acompanhadas de um identificador, concatenado com a letra "t"para designar transportadora, um nome, a sua posição x e y, um número de identificação fiscal (NIF), o raio em que atua e o preço por km que este concebe, sendo que

este preço tem um mínimo de 1.0 e um máximo de 4.0 unidades monetárias.
Nas figuras seguintes vemos a geração de uma transportadora:

```
--gerar NIF
genNIF :: Gen NIF
genNIF = choose(100000000 :: Int,999999999 :: Int)
```

Figura 19: Gerar NIF.

```
--gerar preco por km
genPKm :: Gen PKm
genPKm = choose(0.1 :: Float,4.0 :: Float)
```

Figura 20: Gerar preço por km.

```
--gerar transportadoras
genTransportadora :: Gen Transportadora
genTransportadora = do a <- genIdentificadorT
                       b <- elements nomesTransportadoras
                       c <- genPosX
                       d <- genPosY
                       e <- genNIF
                       f <- genRaio
                       g <- genPKm
                       return (Transportadora a b c d e f g)
```

Figura 21: Geração de uma transportadora.

A entidade loja é composta por um identificador, concatenado à letra "l"que designa loja, um nome, em que este é escolhido aleatoriamente de uma lista predefinida, e uma posição x e y.
Na imagem seguinte vê-se a geração de uma loja:

```
--gerar lojas
genLoja :: Gen Loja
genLoja = do a <- genIdentificadorL
             b <- elements nomesLojas
             c <- genPosX
             d <- genPosY
             return (Loja a b c d)
```

Figura 22: Geração de uma loja.

Por fim, foi gerada a entidade encomenda, em que esta não poderia ter sido gerada sem ter em conta os utilizadores e lojas já existentes, pois uma encomenda tem associado um utilizador e uma loja. Para isto, é criado uma cópia do ficheiro gerado até agora (antes das encomendas),em que vamos ler o mesmo e guardar a seguinte informação.Uma lista com todos os identificadores de utilizadores gerados, outra lista que guarda todos os identificadores de lojas geradas, sendo que estas são passadas ao gerador de encomenda, para este escolher um elemento aleatoriamente da lista de utilizadores e lojas. Uma encomenda precisa de um peso total em kg (mínimo 0.1 e máximo de 500), um conjunto de produtos (para gerar estes produtos nós escolhemos um número aleatório entre 1 e 4 inclusivé) e o resultado irá representar o número de produtos associados à encomenda, sendo que cada produto já está predefinido numa lista. Cada produto tem associado um identificador, um nome, a quantidade e o preço. Nas figuras seguintes apresentamos toda a geração de uma encomenda: .

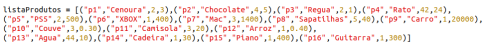


Figura 23: Lista de produtos predefinida.

```
copyFile file "copy"
fileHandle <- openFile "copy" ReadWriteMode
content <- hGetContents fileHandle
let linesOfFiles = lines content
let users = getUsers linesOfFiles
let lojas = getLojas linesOfFiles
ge <- gerador file (genEncomenda users lojas) ne
```

Figura 24: Obter utilizadores e lojas já geradas.

```
--gerar peso
genPeso :: Gen Peso
genPeso = choose(0.1 :: Float,500.0 :: Float)
```

Figura 25: Gerar peso de encomenda.

```
--gerar lista de compras
genLista :: Gen Lista
genLista = do n <- choose(1,4)
              (vectorOf n $ elements listaProdutos)
```

Figura 26: Gerar lista de compras para a encomenda.

```
--gerar encomendas
genEncomenda :: [String] -> [String] -> Gen Encomenda
genEncomenda u l = do a <- genIdentificadorE
                      b <- elements u
                      c <- elements l
                      d <- genPeso
                      p <- genLista
                      return (Encomenda a b c d p)
```

Figura 27: Gerar encomendas.

No que diz respeito à encomenda ser aceite, o grupo decidiu que ao criar uma encomenda esta irá ser imediatamente aceite.Logo, no *csv* mostrado em baixo, cada linha representa uma encomenda tem um ”Aceite:”com o identificador da encomenda.

3.4 tarefa 4: análise de desempenho da aplicação trazaqui

A tarefa 4 consiste em analisar o desempenho da aplicação. Para isto foi usada a plataforma *RAPL* que serve para analisar o consumo de uma aplicação (podendo analisar os métodos que quisermos). O grupo nesta tarefa focou-se exclusivamente no projeto 23,uma vez que, o projeto 83 não pôde ser usado, pois os *logs* gerados para este não são suficientes para a execução, dado que o projeto 83 para fazer *login* de um utilizador requer um email e este nunca foi gerado nem fornecido nos *logs* de exemplo.

O objetivo principal passa por analisar o consumo de energia da aplicação e seu tempo de execução. Para a execução serão usados os ficheiro *logs* criados na tarefa 3.

O consumo de energia será diferenciado em *power of consumption* (energia a dividir pelo tempo, em segundos) e energia (medida em *Joules*), ambos divididos em 3 categorias, em consumo de *DRAM*, *CPU* e *package*.

Foram usados dois ficheiros de *logs* distintos. Um com poucos dados, que contêm cerca de 100 utilizadores, 100 voluntários, 100 transportadoras, 20 lojas e 50 encomendas. O outro contêm muitos dados (grande), que inclui 5000 utilizadores, 5000 voluntários, 1000 transportadoras, 2000 lojas e 7000 encomendas.

Com a finalidade de observar o consumo da energia ,fizemos quatro análises distintas que serão apresentadas de seguida, em que para cada uma das análises foram usados tanto o *csv* com poucas entradas como o *csv* com muitas entradas, de forma a, ser possível comparar o *parsing* do projeto e o consumo do mesmo, em que tudo isto é feito para o projeto sem *refactor* e com *refactor* concluído.As quatro análises distintas são:

- Avaliar consumo e tempo do *parsing*;
- Consumo e tempo da aplicação com o utilizador a fazer *login* e ver a sua lista de encomendas que não existe (ele ainda não adicionou)
- Consumo e tempo da aplicação com o utilizador a fazer *login* e adicionar encomendas e a ver a lista das mesmas (vendo o valor total da mesma também)
- Consumo e tempo de execução do processo de adicionar encomendas.

Relembro que todas estas análises foram feitas atráves de introduzir os dados via terminal (dar *login* e adicionar encomenda), por isso o tempo da aplicação a correr pode não ser o melhor indicador,dado que, o tempo é influenciado pela introdução do identificador ou a adição de encomendas (todos os utilizadores e encomendas tem de estar no ficheiro *logs*, senão falha seria de esperar).
Apresentamos agora os resultados obtidos:

Análise do RAPL **sem** *refactor* - Conjunto de dados **Pequeno**							
	E.dram (J)	E.cpu (J)	E.package (J)	P.dram (W)	P.cpu (W)	P.package (W)	Tempo (seg)
Parsing	0.1942	0.9544	1.7086	1.8906	9.2886	16.6283	0.10
Login + Ver Lista de encomenda vazia	32.5914	3.4460	115.4270	1.0711	0.1132	3.7935	30.90
Login + Encomendar +Ver Lista de encomenda	61.1965	4.4322	214.5396	1.0663	0.0772	3.7382	57.39
Encomendar + ver Lista de Encomendas	25.3940	1.1405	88.0658	1.0638	0.0477	3.6894	23.87

Análise do RAPL **com** *refactor* - Conjunto de dados **pequeno**							
	E.dram (J)	E.cpu (J)	E.package (J)	P.dram (W)	P.cpu (W)	P.package (W)	Tempo (seg)
Parsing	0.1998	0.9559	1.7161	1.9242	9.2075	16.5296	0.10
Login + Ver Lista de encomenda vazia	31.6075	3.8838	112.9759	1.0709	0.1316	3.8278	29.51
Login + Encomendar +Ver Lista de encomenda	51.7608	4.8932	183.5338	1.0681	0.1009	3.7872	48.46
Encomendar + ver Lista de Encomendas	28.9227	1.3891	101.0889	1.0624	0.0510	3.7133	27.22

No que toca ao conjunto de dados pequeno, podemos ver que o impacto que teve no consumo de energia não é notório, quando é comparado entre o sem *refactor* e com *refactor*. Uma vez que, os valores de energia podem ser ”falsos”, devido ao utilizador que introduz os dados no terminal poder ser mais lento ou mais rápida a escrever. A melhor forma de comparação é olhar para os valores de *power* (pode-se pensar como uma média tendo em conta o tempo) e estes encontram-se relativamente próximos tanto com ou sem *refactoring*.

Análise do RAPL **sem** *refactor* - Conjunto de dados **Grande**							
	E.dram (J)	E.cpu (J)	E.package (J)	P.dram (W)	P.cpu (W)	P.package (W)	Tempo (seg)
Parsing	0.9580	5.5848	8.8619	2.1642	12.6168	20.0202	0.44
Login + Ver Lista de encomenda vazia	31.4478	19.1494	127.7788	1.1269	0.6862	4.5788	27.90
Login + Encomendar +Ver Lista de encomenda	68.5652	19.8070	255.0977	1.0838	0.3130	4.0323	63.26
Encomendar + ver Lista de Encomendas	30.7603	1.1871	106.3566	1.0616	0.0409	3.6707	28.97

Análise do RAPL **com** *refactor* - Conjunto de dados **Grande**							
	E.dram (J)	E.cpu (J)	E.package (J)	P.dram (W)	P.cpu (W)	P.package (W)	Tempo (seg)
Parsing	0.8410	4.8430	7.8060	2.1107	12.1540	19.5901	0.39
Login + Ver Lista de encomenda vazia	26.9628	19.4017	113.1036	1.1287	0.8122	4.7348	23.88
Login + Encomendar +Ver Lista de encomenda	67.0715	19.5210	250.4786	1.0829	0.3152	4.0444	61.93
Encomendar + ver Lista de Encomendas	27.8649	1.4313	96.8485	1.0623	0.0545	3.6924	26.22

Já para o conjunto de dados grande, podemos ver que com *refactor* há uma redução no consumo de energia durante o *parsing*. Esta melhoria já não é tão significativa (para estes testes feitos pelo grupo) quando comparando às restantes três avaliações que o grupo fez, principalmente porque há o tempo que o utilizador demora a introduzir os valores que quer testar, como já foi dito anteriormente. Este processo poderia ter sido melhorado, se tivéssemos criado testes para o programa correr autonomamente e assim não incluir o erro humano na equação de verificação do consumo de energia. Posto isto, e uma vez que pode não ter sido aplicado *refactor* a métodos que realmente estivessem a prejudicar o consumo de energia, o grupo sabe que se tivesse aplicado um *refactor* melhor, este poderia ter valores mais significativos para avaliar.

4

CONCLUSÃO

Desde a atribuição do presente projeto, o grupo abraçou o mesmo com entusiasmo e curiosidade sobre o que as ferramentas e temas abordados durante as aulas poderiam trazer de benéfico a projetos de maior escala. A realização do projeto permitiu comprovar a enorme utilidade dessas ferramentas e métodos utilizados.

Durante todo o processo de elaboração deste trabalho, surgiram algumas dificuldades que acabaram por ser ultrapassadas com sucesso. Entre estas, destacam-se a dificuldade de tratamento de todos os projetos atribuídos de forma a tornar possível o processo de compilação; a utilização do RAPL; e a realização do *refactor* automático para todos os ficheiros.

Ultrapassadas as várias fases e dificuldades, foi possível observar todas as vantagens resultantes de *refactoring*, geração/utilização de testes unitários, geração de ficheiros *logs* e verificação da energia do programa. No que toca a *refactoring*, este torna-se útil para melhorar o programa, não só em termos de leitura do mesmo, como melhorar o seu consumo energético e um dos principais fatores que é evitar *bugs*. Quanto aos testes, estes acabam por ser bastantes úteis para verificar o funcionamento da aplicação, no entanto, estes apresentam uma desvantagem,pois não tem em conta valores realistas que são passados aos métodos, mas sim a sua estrutura usando dados "falsos". A geração de ficheiros *logs* auxilia o teste da aplicação, pois este já não usa os tais dados "falsos"mas sim dados que representem informação verdadeira para a aplicação, podendo assim verificar o bom comportamento da mesma aplicação.
Com isto tudo, torna-se importante avaliar o consumo de energia da aplicação, e espera-se que quantos menos *red code smells* existir,menor seja o consumo de energia, e nos dias que correm a energia é um fator determinante para a utilização de uma aplicação.

Por fim, o grupo ressalva que a realização deste trabalho ofereceu-nos uma excelente oportunidade para aprender novas técnicas de analisar o código produzido, que serão certamente bastante vantajosas numa perspectiva futura. No entanto, este projeto tornava-se mais completo se tivéssemos explorado mais a análise do consumo de energia da aplicação e criado outras técnicas de *refactor* automático.

BIBLIOGRAFIA

[1] Métodos Formais em Engenharia de Software - twiki2021
http://wiki.di.uminho.pt/twiki/bin/view/Education/MFES/ATS

[2] jUnit5-2021
https://junit.org/junit5/

[3] kliu20/jrapl_2021,
https://github.com/kliu20/jRAPL

[4] documentation — evosuite_2021,
https://www.evosuite.org/documentation/

[5] try free: automatic java code refactoring tool jsparrow_2021
https://jsparrow.eu

[6] automatically refactor entire code bases!_2021
https://autorefactor.org

[7] test.quickcheck.gen_2021
http://hackage.haskell.org/package/QuickCheck-2.14.2/docs/Test-QuickCheck-Gen.html

[8] sonarlint — fix issues before they exist_2021
https://www.sonarlint.org

[9] code quality and code security — sonarqube_2021
Sonarqube.org