

UNIVERSIDADE DO MINHO
Mestrado Integrado em Engenharia
Informática
Análise e Teste de Software

Análise e Teste de Software

Trabalho Prático

Grupo 5

Alexandre Teixeira — A73547
Bruno Arieira — A70565
Duarte Freitas — A63129
João Palmeira — A73864

Braga, Dezembro de 2019

Resumo

Atualmente, cada vez mais se torna fundamental avaliar/analisar software desenvolvido com o principal objetivo de se obter uma performance de excelência. Para tal, é necessário ter em atenção diferentes indicadores importantes que possam comprometer o bom funcionamento de determinado sistema e procurar soluções por forma a otimizá-los.

Neste contexto, este trabalho prático foi realizado no âmbito da unidade curricular Análise e Teste de Software com a finalidade de analisar e melhorar a qualidade de trabalhos já elaborados por alunos de Programação Orientada a Objetos, através de técnicas de estudo e de teste conforme algumas métricas aprendidas durante as aulas práticas/teóricas.



Conteúdo

1	Introdução	4
2	Qualidade do Código Fonte da Aplicação	5
2.1	Projeto 1	6
2.1.1	<i>Bugs</i>	7
2.2	Projeto 2	10
2.2.1	<i>Bugs</i>	11
3	Refactoring da Aplicação	13
3.1	Projeto 1	15
3.2	Projeto 2	19
4	Teste da Aplicação	20
4.1	<i>JUnit</i>	20
4.1.1	<i>Demo1</i>	20
4.1.2	<i>Demo2</i>	21
4.2	<i>EvoSuite</i>	21
4.3	<i>JaCoCo</i>	22
4.3.1	<i>Demo1</i>	22
4.3.2	<i>Demo2</i>	23
4.4	Gerador de Inputs	24
5	Análise de Desempenho da Aplicação	26
5.1	<i>Demo1</i>	26
5.1.1	<i>logsPOO_carregamentoInicial</i>	27
5.1.2	<i>Large</i>	28
5.1.3	<i>Mini</i>	29
5.2	<i>Demo 2</i>	31
5.2.1	<i>logsPOO_carregamentoInicial</i>	31
5.2.2	<i>Large</i>	32
5.2.3	<i>Mini</i>	32
5.3	Análise de Resultados (Com Refactoring vs Sem Refactoring)	34
	Conclusão e Trabalho Futuro	35



Lista de Tabelas

1	Definição das métricas de código do 1º projeto	6
2	Definição das métricas de código do 2º projeto	10
3	Métricas do 1º projeto nas várias versões	18
4	Métricas do 2º projeto nas várias versões	19
5	<i>Carregamento ficheiro logs com logsPOO_carregamentoInicial</i>	27
6	<i>Melhores Clientes com logsPOO_carregamentoInicial</i>	27
7	<i>Registar com logsPOO_carregamentoInicial</i>	27
8	<i>Adicionar Carro com logsPOO_carregamentoInicial</i>	27
9	<i>Login com logsPOO_carregamentoInicial</i>	27
10	<i>Tempos de Melhores Clientes com logsPOO_carregamentoInicial</i>	28
11	<i>Viagens de Melhores Clientes com logsPOO_carregamentoInicial</i>	28
12	<i>Carregamento ficheiro logs com Large</i>	28
13	<i>Melhores Clientes com Large</i>	28
14	<i>Registar com Large</i>	28
15	<i>Adicionar Carro com Large</i>	29
16	<i>Login com Large</i>	29
17	<i>Tempos de Melhores Clientes com Large</i>	29
18	<i>Viagens de Melhores Clientes com Large</i>	29
19	<i>Carregamento ficheiro logs com Mini</i>	29
20	<i>Melhores Clientes com Mini</i>	30
21	<i>Registar com Mini</i>	30
22	<i>Adicionar Carro logs com Mini</i>	30
23	<i>Login com Mini</i>	30
24	<i>Tempos de Melhores Clientes com Mini</i>	30
25	<i>Viagens de Melhores Clientes com Mini</i>	30
26	<i>Carregamento ficheiro logs com logsPOO_carregamentoInicial</i>	31
27	<i>Registar com logsPOO_carregamentoInicial</i>	31
28	<i>Adicionar carro com logsPOO_carregamentoInicial</i>	31
29	<i>Login com logsPOO_carregamentoInicial</i>	32
30	<i>Melhores Clientes (km) com logsPOO_carregamentoInicial</i>	32
31	<i>Carregamento ficheiro logs com Large</i>	32
32	<i>Melhores Clientes (Alugueres) com Large</i>	32
33	<i>Registar com Large</i>	32
34	<i>Adicionar carro com Large</i>	33
35	<i>Login com Large</i>	33
36	<i>Tempos de Melhores Clientes (km) com Large</i>	33
37	<i>Carregamento ficheiro logs com Mini</i>	33
38	<i>Melhores Clientes (Alugueres) com Mini</i>	33
39	<i>Registar com Mini</i>	33
40	<i>Adicionar carro com Mini</i>	34
41	<i>Login com Mini</i>	34
42	<i>Melhores Clientes (km) com Mini</i>	34



Lista de Figuras

1	Análise dos dois projetos no <i>SonarQube</i>	5
2	<i>Overview</i> do 1º projeto	6
3	Classes mais complexas	7
4	<i>Bug 1</i> (Projeto 1)	8
5	<i>Bug 2</i> (Projeto 1)	8
6	<i>Bug 3</i> (Projeto 1)	8
7	<i>Bug 4</i> (Projeto 1)	9
8	<i>Bug 5</i> (Projeto 1)	9
9	<i>Overview</i> do 2º projeto	10
10	Classes mais complexas	11
11	<i>Bug 1</i> (Projeto 2)	12
12	<i>Bug 2</i> (Projeto 2)	12
13	<i>Bug 3</i> (Projeto 2)	12
14	Exemplo da aplicação de refactoring	13
15	<i>Code Smells</i>	14
16	Testes <i>refactoring</i> no SonarQube	15
17	1º exemplo <i>refactoring</i>	16
18	2º exemplo <i>refactoring</i>	16
19	3º exemplo <i>refactoring</i>	17
20	4º exemplo <i>refactoring</i>	18
21	Exemplo teste unitário Demo 1	20
22	Exemplo teste unitário Demo 2	21
23	Cobertura Demo 1	22
24	Cobertura Demo 2	23



1 Introdução

Este trabalho será realizado no âmbito da Unidade Curricular de **Análise e Teste de Software** com o objetivo principal de analisar dois projetos do ano lectivo anterior da UC de POO.

Numa primeira fase será necessário analisar a qualidade do código das duas aplicações através do **SonarQube**, avaliando quantitativamente e qualitativamente os seus **bugs**, **code smells**, **métricas**, entre outros parâmetros. Como complemento, irá ser elaborado algumas regras com o objetivo de encontrar alguns **red smells**.

A segunda tarefa do trabalho diz respeito ao **Refactoring** das duas aplicações tendo como objetivo analisar os diferentes tipos de *smells* e eliminá-los caso seja possível.

Para a terceira fase deste trabalho serão realizados testes para ambos os projetos, mais concretamente testes unitários em **JUnit** ou através do sistema **EvoSuite**.

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



2 Qualidade do Código Fonte da Aplicação

Teoricamente falando, qualidade de software e de código são duas disciplinas bem diferentes, cada uma delas está relacionada a diferentes aspectos e indicadores qualitativos do software.

Existem diversos recursos e metodologias que guiam o processo de desenvolvimento de software visando a construção de aplicações com maior qualidade. Medir a qualidade de um projeto de software é um grande desafio e envolve diversas métricas e indicadores. Este processo numa visão ampla, procura a conformidade a requisitos funcionais e de desempenho declarados explicitamente, padrões de desenvolvimento claramente documentados e critérios de qualidade. Existem cinco dos principais traços a serem medidos para obter maior qualidade:

- **Confiabilidade:** mede a probabilidade de um sistema ser executado sem falhas durante um período específico de operação. Está relacionado com o número de defeitos e com a disponibilidade do software.
- **Manutenção:** mede a facilidade com que o software pode ser mantido. Está relacionado ao tamanho, consistência, estrutura e complexidade da base de código.
- **Testabilidade:** mede quanto bem o software suporta os esforços de teste. A testabilidade pode ser medida com base na quantidade de casos de teste que se precisa para encontrar possíveis falhas no sistema (a complexidade ciclomática é uma das técnicas que entra neste fator).
- **Portabilidade:** mede a capacidade de utilização do mesmo software em ambientes diferentes. Está relacionado com a independência da plataforma.
- **Reutilização:** mede se os ativos existentes (como código fonte) podem ser usados novamente.

Nesta primeira secção iremos introduzir os dois projetos no **SonarQube** com o objetivo de analisá-los e testá-los segundo algumas métricas. Como podemos verificar pela figura 1, que corresponde a uma análise inicial onde o *demo2* é o projeto 1 e o *demo1* é o projeto 2, o *SonarQube* fornece-nos dados totais referentes a *bugs*, vulnerabilidades, *code smells*, *coverage* e duplicações.

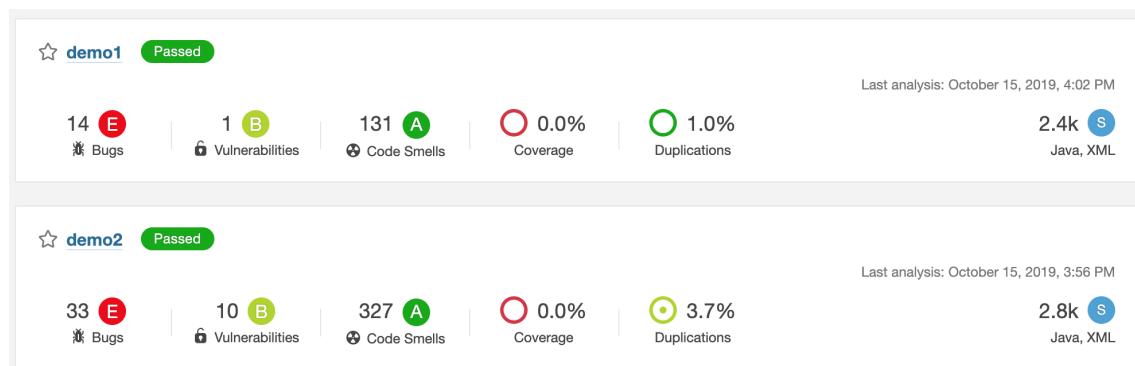


Figura 1: Análise dos dois projetos no *SonarQube*



2.1 Projeto 1

Começamos por avaliar o primeiro projeto. Este projeto passou no controlo do *SonarQube* e conta com 33 *bugs*, 10 vulnerabilidades, 327 *code smells* e uma percentagem de duplicados na ordem dos 3.7%.

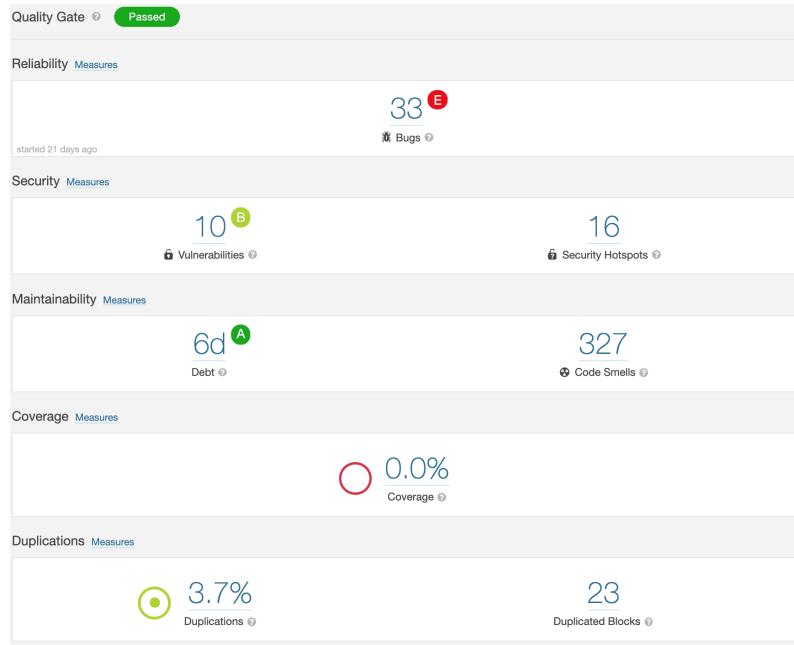


Figura 2: Overview do 1º projeto

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

Code Metrics	Values
Statements	1878
Functions	283
Classes	34
Lines	5151
Comment Lines	742
LOC	2838
NCLOC	2096
Files	35
Bugs	33
Code Smells	327
Duplicated Lines	192
Functions/Classes	8,32
Statements/Functions	6,5
Cyclomatic Complexity	619
Cognitive Complexity	479

Tabela 1: Definição das métricas de código do 1º projeto

Através destes valores obtidos pode-se inferir que existe bastante código elaborado como podes ver pela quantidade de linhas de código elaboradas (2838), classes (34) e



métodos (283), daí a existência de uma grande quantidade de *bugs* e *code smells* que iremos falar mais tarde. Analisando a *Cyclomatic Complexity* (mede o número mínimo de casos de teste necessários para a cobertura total do teste) e da *Cognitive Complexity* (medida de quanto difícil é o aplicativo entender) verifica-se que têm valores elevados, o que nos diz que a complexidade da aplicação, em geral, é um pouco elevada.

Como podemos visualizar na figura 3, existem 2 classes com a sua classificação de confiabilidade com valores baixos, pois a quantidade de maus cheiros e *bugs* nestas classes é considerável.

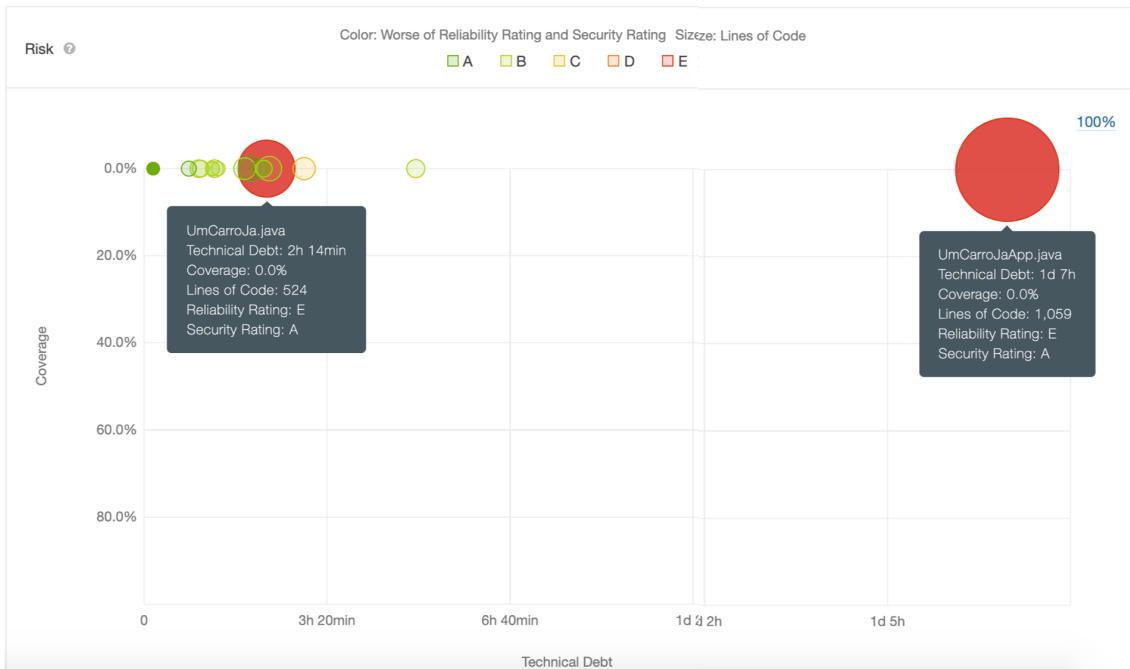


Figura 3: Classes mais complexas

2.1.1 Bugs

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



2 QUALIDADE DO CÓDIGO FONTE DA APLICAÇÃO

```
// calcula a nova latitude
double newLat = latitude + (kilometers / latitudeConstant());

return new Coordinate(new Double(newLat).doubleValue(), longitude);
```

Remove the boxing of "newLat". [See Rule](#) 28 days ago L98 🔍
Bug Minor Open Not assigned 5min effort Comment clumsy

Figura 4: Bug 1 (Projeto 1)

"Boxing" é o processo de colocar um valor primitivo num objeto análogo, como criar um número inteiro para armazenar um valor int. Como se pode reparar na figura 4, neste erro a variável "newLat" já foi inicializada acima com o tipo double, não havendo necessidade de a declarar novamente pois a variável pode ser reutilizada novamente.

```
Long c = Math.round(novaClassificacao);
int classifiFinal = Integer.valueOf(c.intValue());
```

Remove the boxing to "Integer". [See Rule](#) 28 days ago L281 🔍
Bug Minor Open Not assigned 5min effort Comment clumsy

Figura 5: Bug 2 (Projeto 1)

Situação idêntica á anterior, nesta figura 5 existe a variável "c" que ao ser retornada está a ser feito o cast da mesma para o tipo integer duas vezes, desnecessariamente.

```
// converte metros para km
double kilometers = distance / 1000;
```

Cast one of the operands of this division operation to a "double". [See Rule](#) 28 days ago L135 🔍
Bug Minor Open Not assigned 5min effort Comment based-on-misra, cert, cwe, overflow, s...

Figura 6: Bug 3 (Projeto 1)

Quando a aritmética é realizada com números inteiros, o resultado será um inteiro. A atribuição desse resultado ao tipo long, double ou float com conversão automática de tipo, mas, como foi iniciado como int, o resultado provavelmente não será o esperado. Neste caso verifica-se que é criado uma variável do tipo double que recebe o resultado de uma operação. É considerado um bug devido aos operandos deste cálculo serem inteiros.



2 QUALIDADE DO CÓDIGO FONTE DA APLICAÇÃO

```

case "Hibrido":
    4 Veiculo ch = 3 parseCarroHibrido(linha);
    return 5 ch.clone();

```

A "NullPointerException" could be thrown; "ch" is nullable here. [See Rule](#)

28 days ago ▾ L76 🔍

Bug ▾ Major ▾ Open ▾ Not assigned ▾ 10min effort Comment cert, cwe ▾

Figura 7: Bug 4 (Projeto 1)

Uma referência a *null* nunca deve ser aceida, pois pode acontecer com que a exceção *NullPointerException* seja lançada. Na melhor das hipóteses, essa exceção pode causar o encerramento imediato do programa e na pior, pode permitir que um invasor ignore as medidas de segurança e tenha acesso a informações de depuração do código.

```

/* Data de início da aplicação */
private static GregorianCalendar dataInicioApp;

Make "dataInicioApp" an instance variable. See Rule
28 days ago ▾ L54 🔍
Bug ▾ Major ▾ Open ▾ Not assigned ▾ 15min effort Comment cert, multi-threading ▾
```

```

/** O construtor é privado, pois não queremos instâncias da mesma. */
private UmCarroJaApp() {}

/** Métodos de Classe */

/**
 * @brief Inicia a aplicação lendo o ficheiro objeto que contém
 * a data de início da aplicação e os dados.
 */
private static void initApp() {
    try {
        FileInputStream fis = new FileInputStream(ficheiroDados);

        Use try-with-resources or close this "FileInputStream" in a "finally" clause. See Rule
28 days ago ▾ L68 🔍
Bug ▾ Blocker ▾ Open ▾ Not assigned ▾ 5min effort Comment cert, cwe, denial-of-service, leak ▾

        ObjectInputStream ois = new ObjectInputStream(fis);

        Use try-with-resources or close this "ObjectInputStream" in a "finally" clause. See Rule
28 days ago ▾ L69 🔍
Bug ▾ Blocker ▾ Open ▾ Not assigned ▾ 5min effort Comment cert, cwe, denial-of-service, leak ▾

            dataInicioApp = (GregorianCalendar) ois.readObject();
            uci = (UmCarroJa) ois.readObject();
        } catch (FileNotFoundException e){
            dataInicioApp = new GregorianCalendar();
            uci = new UmCarroJa();
            System.out.println("Erro ao ler os dados!\nErro de leitura\n.");
        }
        catch (IOException e) {
            dataInicioApp = new GregorianCalendar();
            uci = new UmCarroJa();
            System.out.println("Erro ao ler os dados!\nErro de leitura\n.");
        }
        catch (ClassNotFoundException e){
            dataInicioApp = new GregorianCalendar();
            uci = new UmCarroJa();
            System.out.println("Erro ao ler os dados! Ficheiro com formato desconhecido!\n.");
        }
    }
}

```

Figura 8: Bug 5 (Projeto 1)

Nesta figura 8 estão representados **três bugs**. O **primeiro** está associado ao facto de se declarar uma variável estática do tipo "GregorianCalendar". Isto advém do facto de nem todas as classes na biblioteca Java padrão estarem gravadas para serem seguras no encadeamento, pois o seu uso de forma multi-encadeado pode levar a problemas ou exceções durante o tempo de execução. O **segundo** é devido à criação de ficheiros e a não serem fechados após a sua utilização. Esta chamada final é aconselhada que seja feita num bloco de código "*finally*", caso contrário uma exceção pode impedir que a chamada seja feita. Por fim, o **terceiro** bug foi detetado pela mesma razão que o anterior.



2.2 Projeto 2

Relativamente à avaliação do segundo projeto, averiguamos, segundo as diferentes métricas do Sonarqube, que apresenta 14 bugs, 1 vulnerabilidade, 131 code smells e exibe uma densidade de 1% de duplicados em relação ao código, tal como se encontra apresentado na figura a seguir.

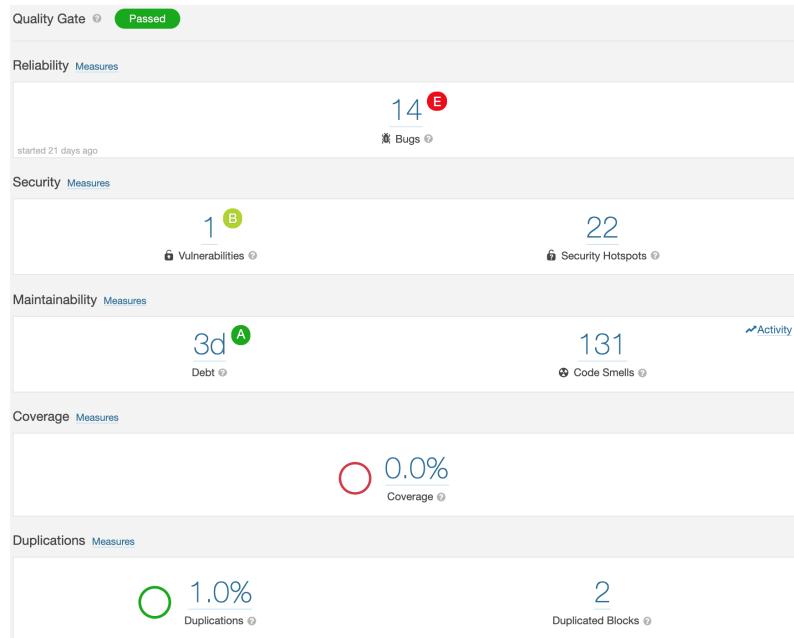


Figura 9: *Overview* do 2º projeto

Seguindo a organização da apresentação dos resultados do primeiro projeto, depois de efetuado todo o processamento para análise deste projeto, exibimos em formato tabular os valores para as diferentes métricas obtidas, diretamente relacionadas com o código fonte.

Code Metrics	Values
Statements	1042
Functions	247
Classes	43
Lines	2912
Comment Lines	34
LOC	2401
NCLOC	2367
Files	42
Bugs	14
Code Smells	131
Duplicated Lines	28
Functions/Classes	5,74
Statements/Functions	4.22
Cyclomatic Complexity	472
Cognitive Complexity	250

Tabela 2: Definição das métricas de código do 2º projeto

Depois de uma leitura atenta desta tabela, constatamos que este projeto relativa-



2 QUALIDADE DO CÓDIGO FONTE DA APLICAÇÃO

mente ao primeiro apresenta uma densidade menor em termos de linhas de código (2912), de linhas de comentários (34) e de funções/métodos (247), sendo que exibe um maior número de classes (43). Relativamente à complexidade ciclomática, que tem em atenção a quantidade de caminhos de execução independentes do código fonte, e à complexidade cognitiva, verificam-se resultados um pouco elevados. Existe também um aspeto bastante positivo que é o número reduzido de linhas duplicadas que baixa um pouco a probabilidade do software futuramente apresentar erros.

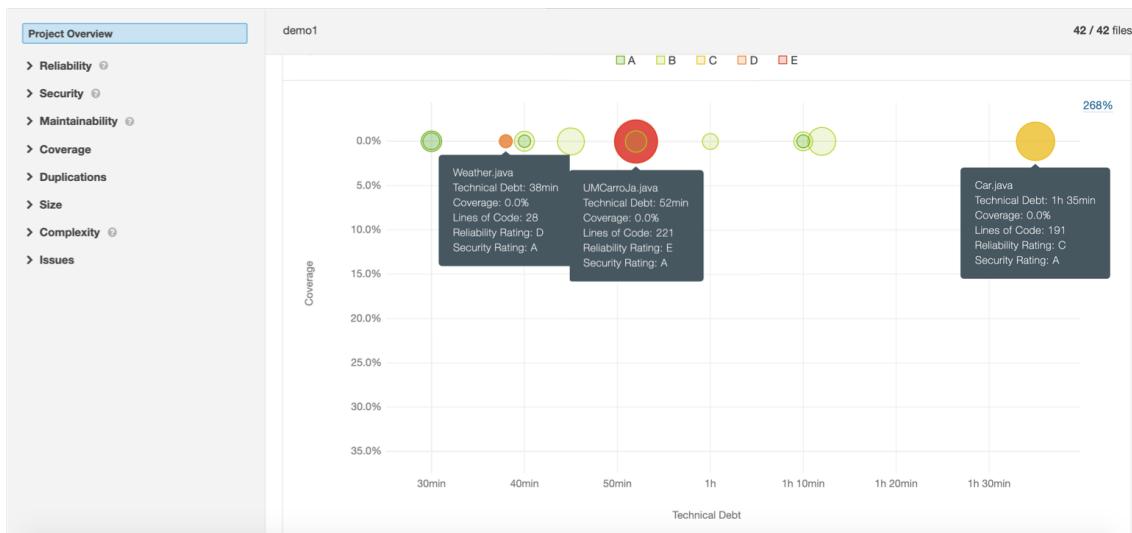


Figura 10: Classes mais complexas

Como se observar na figura anterior, existem 3 classes que se destacam e que apresentam a sua classificação de confiabilidade com valores baixos, nomeadamente a classe UMCarroJa.java, devido á quantidade de *bad smells* e *bugs* nestas classes ser relevante.

2.2.1 Bugs

Tal como foi visto anteriormente, este tipo de avaliação ao código-fonte é bastante relevante no sentido de promover confiabilidade e qualidade do mesmo a longo prazo por forma a que seja eficiente. Relativamente a este projeto, não foram encontrados bugs com grande relevância, segundo a ferramenta *Sonarqube*.



2 QUALIDADE DO CÓDIGO FONTE DA APLICAÇÃO

```
public void save(String fName) throws IOException {
    FileOutputStream a = new FileOutputStream(fName);
    ObjectOutputStream r = new ObjectOutputStream(a);

    Use try-with-resources or close this "ObjectOutputStream" in a "finally" clause. See Rule
    ↗ Bug 1 Blocker 0 Open Not assigned 5min effort
    last month ▾ L245 96
    cert, cwe, denial-of-service, leak

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

Figura 11: *Bug 1* (Projeto 2)

Como foi analisado anteriormente, os ficheiros necessitam de ser fechados ("*file.close()*") após a sua utilização e dentro de um bloco/cláusula "*finally*". Neste caso, o ficheiro "*a*" foi usado, mas no final não foi fechado, apenas o "*ObjectOutputStream r*".

```
public static UMCarroJa read(String fName) throws IOException, ClassNotFoundException {
    FileInputStream r = new FileInputStream(fName);
    ObjectInputStream a = new ObjectInputStream(r);

    Use try-with-resources or close this "ObjectInputStream" in a "finally" clause. See Rule
    ↗ Bug 1 Blocker 0 Open Not assigned 5min effort
    last month ▾ L253 96
    cert, cwe, denial-of-service, leak

    UMCarroJa u = (UMCarroJa) a.readObject();
    a.close();
    return u;
}
```

Figura 12: *Bug 2* (Projeto 2)

Esta situação é idêntica á da figura anterior, em que se inicializou o ficheiro "*r*", utilizou-se, e no final não se fechou devidamente.

```
public double getSeasonDelay() {
    Random a = new Random();

    Save and re-use this "Random". See Rule
    ↗ Bug 1 Critical 0 Open Not assigned 5min effort
    last month ▾ L20 96
    owasp-a6
```

Figura 13: *Bug 3* (Projeto 2)

O construtor "*Random()*" tenta definir sempre um valor diferente. No entanto, não existe garantia de que o valor seja aleatório ou mesmo distribuído uniformemente. Alguns JDK usam o tempo atual como género de algoritmo, o que torna os números gerados não aleatórios. Este *bug* foi localizado, porque a regra implícita, deteta casos em que um novo *Random* é criado toda vez que um método é chamado e atribuído a uma variável aleatória local.



3 Refactoring da Aplicação

Esta é uma técnica fundamental e que tem todo sentido em ser utilizada, pois nem sempre um sistema de software que apresente um bom funcionamento externamente tem uma boa performance, isto é, deve-se ter sempre em atenção alguns factores, como o código-fonte que deve ser limpo e de fácil compreensão. Considerando este um fator de máxima importância, a legibilidade do código e a falta de qualquer tipo de preocupação com este tema, quando o tempo escasseia os programadores dão mais relevância a finalizar o software do que propriamente ao código escrito, havendo a necessidade de se realizar um refactor do mesmo. Este processo passa por analisar todo software fazendo modificações a nível estrutural, ou seja, sem nunca mudar o seu comportamento externo garantindo sempre a sua perfeita execução.

Sendo assim, este processo apresenta vantagens sobre o código-fonte e consequentemente sobre a qualidade interna do software, das quais se destacam:

- Maior legibilidade;
- Facilita procura de *bugs*;
- Maior eficiência na programação;

O **refactoring** deve se aplicar quase sempre como forma de gerir e avaliar a estrutura interna de um sistema. No entanto, esta técnica deve-se empregar necessariamente quando:

- Ocorre a adição de uma nova função/método;
- Análise e pesquisa de *bugs*;

```
if(condição 1)
{
    if(condição 2) return a1;           Com refactoring
    else return a2;                   if((condição 1) && (condição 2)) return a1;
}
else return a2;
```

Figura 14: Exemplo da aplicação de refactoring

Uma forma de procurar os excertos de código que precisam de refactor passa inicialmente por verificar os *smells* do código. Como já foi analisado previamente existem vários tipos de *smells* e podem ser ultrapassados com o refactor, tais como:

- Duplicação do código;
- Desuso de código;
- Método/função demasiado extenso;
- Parametrização em excesso;
- Classes demasiado extensas;



Estes *smells* mencionados são os que menos interferem a nível de complexidade e os mais fáceis de detetar a "olho nu". O outro tipo de *smells* que se deve ter sensibilidade são conhecidos como *red smells* dos quais fazem parte:

- Copiar array manualmente;
- Travessias de matrizes pelas colunas;
- Concatenação de Strings com o operador '+';
- Operações aritméticas;
- Uso de *exception*;
- Uso de *gets* e *sets*;
- Criação de objetos;
- Uso de tipos de dados primitivos;

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

Figura 15: *Code Smells*

Após o estudo do tipo de *smells* e *refactor* foi então possível proceder ao *refactoring* das duas versões do projecto UMCarroJa. Em seguida, apresentamos alguns exemplos da aplicação de refatores relativamente aos dois projetos, utilizando os mesmos procedimentos usados nas aulas práticas.

Para proceder á realização desta tarefa, utilizamos uma funcionalidade do IDE IntelliJ, em que basta selecionar o código pretendido onde se pretende aplicar o refator e o tipo associado. Também utilizamos um *plugin* do mesmo IDE designado AutoRefactor, em que basta selecionar a classe onde se pretende empregar o refator e este é feito automaticamente.



3 REFACTORING DA APLICAÇÃO

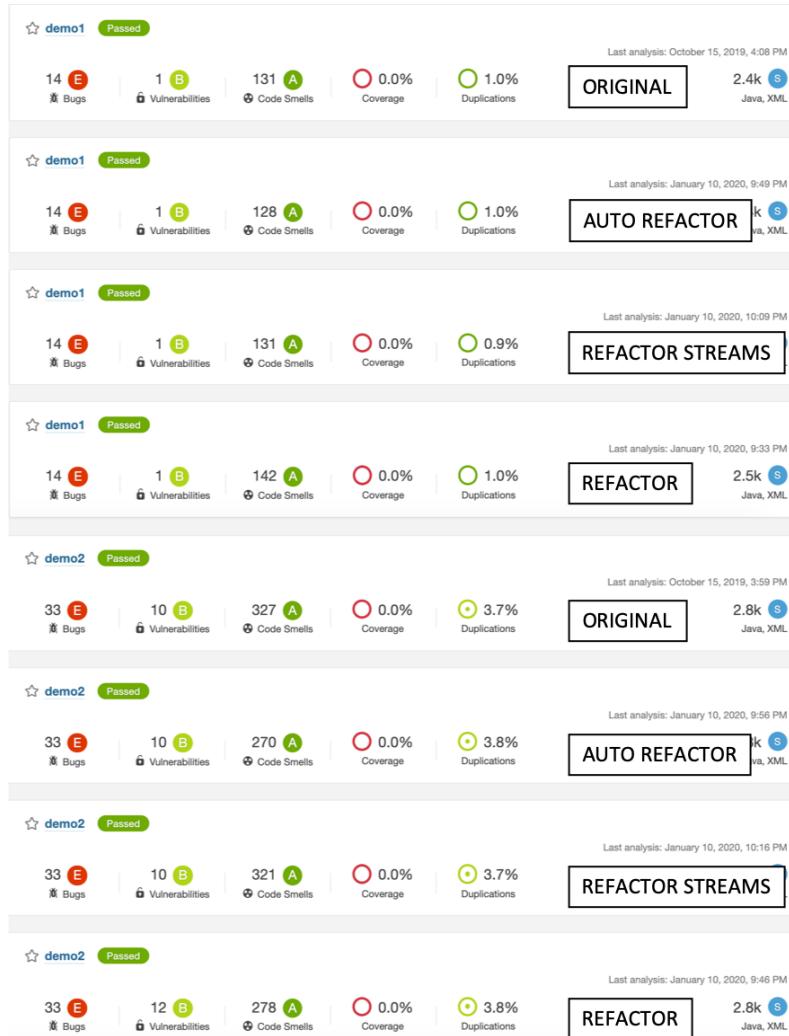


Figura 16: Testes *refactoring* no SonarQube

3.1 Projeto 1

Um dos exemplos onde aplicamos o refator foi numa estrutura *Switch/Case* do método *run* da classe *Controller.java*, que tinha um número elevado de casos. Por sua vez, cada caso era demasiado extenso em termos de linhas de código, o que tornava o método exacerbadamente grande, tornando-o num tipo de *smell*. Para tal, como podemos observar abaixo, decidimos dividir todo o código que se encontrava dentro dos *cases* em funções, tornando desta forma o método menos extenso e promove uma maior legibilidade.



3 REFACTORING DA APLICAÇÃO

```
switch (menu.getMenu()) {  
    case Login:  
        try {  
            NewLogin r = menu.newLogin(error);  
            user = model.logIn(r.getUser(), r.getPassword());  
            menu.selectOption((user instanceof Client)? Menu.MenuInd.Client : Menu.MenuInd.Owner);  
            error = "";  
        }  
        catch (InvalidUserException e){ error = "Invalid Username"; }  
        catch (WrongPasswordException e){ error = "Invalid Password"; }  
        break;  
    case RegisterClient:
```



```
switch (menu.getMenu()) {  
    case Login:  
        error = menuLogin(error);  
        break;  
    case RegisterClient:  
        error = menuRegisterClient(error);  
        break;
```

Figura 17: 1º exemplo *refactoring*

Este exemplo insere-se na descrição definida anteriormente, mas desta vez para uma estrutura condicional *If...Then...Else* no método *getCar* na classe *Cars.java*. Neste caso, quando a condição *if* é verificada existe um excerto de código que é bastante comprido e, mais uma vez, torna o método extenso. Tal como encima, para resolver este problema, decidimos dividir em funções.

```
if (compare.equals("MaisPerto")) {  
    return this.carBase  
        .values()  
        .stream()  
        .filter(e -> e.getType().equals(a)  
            && e.hasRange(dest)  
            && e.isAvailable())  
        .sorted(Comparator.comparingDouble(e ->  
            e.getPosition()  
            .distanceBetweenPoints(origin)))  
        .collect(Collectors.toList())  
        .get(0);  
}
```



```
if (compare.equals("MaisPerto")) {  
    return carMaisPerto(dest, origin, a);  
}
```

Figura 18: 2º exemplo *refactoring*



3 REFACTORYING DA APLICAÇÃO

O exemplo a seguir apresentado segue a mesma ideologia dos anteriores, no entanto, em vez de ser aplicado a estruturas condicionais, foi empregue no *return* do método *Equals* da classe *Car.java*.

```
@Override  
public boolean equals(Object o) {  
    if (this == o) return true;  
  
    if (o == null || this.getClass() != o.getClass()) return false;  
  
    Car car = (Car) o;  
    return this.avgSpeed == car.avgSpeed  
        && this.basePrice == car.basePrice  
        && this.gasMileage == car.gasMileage  
        && this.fullTankRange == car.fullTankRange  
        && this.isAvailable == car.isAvailable  
        && this.range == car.range  
        && this.rating == car.rating  
        && this.nRatings == car.nRatings  
        && this.numberPlate.equals(car.numberPlate)  
        && this.owner.equals(car.owner)  
        && this.brand.equals(car.brand)  
        && this.type == car.type  
        && this.position.equals(car.position)  
        && this.historic.equals(car.historic);  
}
```



```
@Override  
public boolean equals(Object o) {  
    if (this == o) return true;  
  
    if (o == null || this.getClass() != o.getClass()) return false;  
  
    Car car = (Car) o;  
    return equalAux(car);  
}
```

Figura 19: 3º exemplo *refactoring*

Neste caso, demonstramos o exemplo do refator automático em que transformou o código que estava compactado através do uso de streams, num código "extenso". Concluindo que o uso de streams gera *smells* no código.



```
public List<Entry<String, Double>> getBestClients() {
    return this
        .users
        .getClientIDS()
        .stream()
        .collect(Collectors
            .toMap(Function.identity(),
                (e) -> rentals.getRentalListClient(e)
                    .stream()
                    .map(Rental::getDistance)
                    .reduce(0.0, Double::sum)))
        .entrySet()
        .stream()
        .sorted(Collections
            .reverseOrder(Comparator
                .comparingDouble(Entry::getValue)))
        .collect(Collectors.toList());
}
```



```
public List<Entry<String, Double>> getBestClients() {
    Map<String, Double> map = new HashMap<>();
    for (String e : this
        .users
        .getClientIDS()) {
        Double acc = 0.0;
        for (Rental rental : rentals.getRentalListClient(e)) {
            Double distance = rental.getDistance();
            acc = acc + distance;
        }
        if (map.put(e, acc) != null) {
            throw new IllegalStateException("Duplicate key");
        }
    }
    List<Entry<String, Double>> list = new ArrayList<>();
    for (Entry<String, Double> stringDoubleEntry : map
        .entrySet()) {
        list.add(stringDoubleEntry);
    }
    list.sort(Collections
        .reverseOrder(Comparator
            .comparingDouble(Entry::getValue)));
    return list;
}
```

Figura 20: 4º exemplo *refactoring*

Decidimos também recolher algumas métricas através do *SonarQube* de modo a perceber a influência que os tipos de *refactoring* têm na execução do mesmo. Neste caso, o melhor é quando se realiza o Auto Refactor, pois consegue-se baixar os *code smells* e, pelo menos, manter tanto a *Cyclomatic Complexity* como a *Cognitive Complexity*.

	Bugs	Code Smells	Duplications	Cyclomatic Complexity	Cognitive Complexity
ORIGINAL	14	131	1.0%	472	250
AUTO REFACTOR	14	128	1.0%	472	250
REFACTOR STREAMS	14	131	0.9%	503	356
REFACTOR	14	142	1.0%	492	177

Tabela 3: Métricas do 1º projeto nas várias versões



3.2 Projeto 2

Para este projeto os refatores foram semelhantes, não havendo necessidade de repetir a forma de como estes foram aplicados, sendo que está explícito de forma clara anteriormente.

Tal como já foi referido, para este projeto foi também feita uma análise conforme algumas métricas, mediante os diferentes tipos de refactores aplicados. Desta forma, podemos observar que para os indicadores mais relevantes (*Code Smells* e *Cyclomatic Complexity*) os resultados são menores relativamente aos tipos restantes, o que significa que pelo Auto Refactor obtém-se melhor desempenho em termos de análise do código fonte.

	Bugs	Code Smells	Duplications	Cyclomatic Complexity	Cognitive Complexity
ORIGINAL	33	327	3.7%	612	479
AUTO REFACTOR	33	270	3.8%	608	464
REFACTOR STREAMS	33	321	3.7%	636	544
REFACTOR	33	278	3.8%	607	449

Tabela 4: Métricas do 2º projeto nas várias versões



4 Teste da Aplicação

Nesta secção são apresentados várias técnicas/ferramentas de teste de *software* bem como vários tipos de testes unitários realizados aos diferentes projetos. Para além dos testes realizados, foi elaborado um gerador de *inputs* que permite gerar automaticamente ficheiros de *logs* de ambos os projetos fornecidos.

4.1 JUnit

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

4.1.1 Demo1

```
class CarsTest {  
  
    UMCarroJa m = new UMCarroJa();  
    Cars carBase = new Cars();  
    Point p = new Point( x: 1.0, y: 2.1);  
    Point p1 = new Point( x: 5.0, y: 1.1);  
    Owner o = new Owner( email: "d@gmail.com", name: "D", address: "rua", nif: 99999999, passwd: "pass");  
  
    @Test  
    void addCarTest() throws CarExistsException, InvalidUserException, InvalidCarException {  
  
        Car a = new Car( numberPlate: "00-AA-00", o, gas , avgSpeed: 100.0, basePrice: 1.50, gasMileage: 9.6, range: 120, p,  
        carBase.addCar(a);  
        assertEquals( expected: "d@gmail.com", carBase.searchCar( numberPlate: "00-AA-00").getOwnerID());  
    }  
  
    @Test  
    void addCarTest2() throws CarExistsException, InvalidUserException, InvalidCarException {  
  
        Car a = new Car( numberPlate: "00-AA-00", o, gas , avgSpeed: 100.0, basePrice: 1.50, gasMileage: 9.6, range: 120, p,  
        carBase.addCar(a);  
        assertEquals( expected: "d@gmail.com", carBase.searchCar( numberPlate: "00-AA-00").getOwnerID());  
    }  
  
    @Test  
    void searchCarTest() throws CarExistsException, InvalidCarException {  
        Car a = new Car( numberPlate: "00-AA-00", o, gas , avgSpeed: 100.0, basePrice: 1.50, gasMileage: 9.6, range: 120, p,  
        carBase.addCar(a);  
        assertSame(a, carBase.searchCar( numberPlate: "00-AA-00"));  
    }  
}
```

Figura 21: Exemplo teste unitário Demo 1



4.1.2 Demo2

```
public class AluguerTest {

    GregorianCalendar in = new GregorianCalendar();
    GregorianCalendar fim = new GregorianCalendar();
    Coordinate c = new Coordinate( latitude: 10, longitude: 10);
    Aluguer a = new Aluguer( mail: "d@gmail.com", matricula: "AA-00-AA", in, fim, precoViagem: 10.2, tempoAPe: 5, tempoViagem: 7, c);

    @Test
    public void getEmail() { assertEquals( expected: "d@gmail.com", a.getEmail()); }

    @Test
    public void getMatricula() {
        assertEquals( expected: "AA-00-AA", a.getMatricula());
    }

    @Test
    public void getDataInicio() {
        assertEquals(in, a.getDataInicio());
    }

    @Test
    public void getDataFim() {
        assertEquals(fim, a.getDataFim());
    }

    @Test
    public void getCustoViagem() {
        assertEquals( expected: 10.2, a.getCustoViagem(), delta: 1);
    }
}
```

Figura 22: Exemplo teste unitário Demo 2

4.2 EvoSuite

O *EvoSuite* foi a primeira ferramenta de teste de *software* utilizada. Esta ferramenta gera automaticamente testes de unidade para software Java. O *EvoSuite* usa um algoritmo evolutivo para gerar testes *JUnit*. Pode ser executado a partir da linha de comandos e também possui *plugins* para integrá-lo ao *Maven*, *IntelliJ* (onde foi utilizado) e *Eclipse*. O *EvoSuite* foi utilizado em ambos os projetos, também permitindo encontrar possíveis bugs.



4.3 JaCoCo

Já o *JaCoCo* foi a segunda ferramenta de teste de *software* utilizada, onde se utilizou os testes unitários desenvolvidos anteriormente para obter uma cobertura do código. Isto é, o *JaCoCo* permite verificar as percentagens de código por classe que os testes unitários desenvolvidos cobrem. Esta ferramenta foi utilizada nos dois projetos como podemos verificar a seguir.

4.3.1 Demo1

Como se pode notar pela figura seguinte, o *JaCoCo* fornece-nos dados estatísticos da cobertura em cada classe do projeto, por método e ainda por linha.

Element	Class, %	Method, %	Line, %
Car	100% (2/2)	66% (22/33)	61% (61/100)
Cars	100% (1/1)	52% (9/17)	44% (37/83)
CarsTest	100% (1/1)	100% (7/7)	95% (42/44)
Client	100% (1/1)	44% (4/9)	43% (10/23)
Owner	100% (1/1)	8% (1/12)	13% (5/38)
Parser	0% (0/1)	0% (0/4)	0% (0/67)
Rental	100% (1/1)	50% (9/18)	21% (17/80)
Rentals	100% (1/1)	7% (1/13)	7% (4/52)
RentalTest	100% (1/1)	100% (6/6)	100% (19/19)
Traffic	0% (0/1)	0% (0/1)	0% (0/7)
UMCarroJa	100% (1/1)	3% (1/33)	3% (5/146)
User	100% (1/1)	75% (9/12)	67% (25/37)
Users	100% (1/1)	80% (4/5)	62% (10/16)
UsersTest	100% (1/1)	100% (3/3)	100% (13/13)
Weather	0% (0/1)	0% (0/3)	0% (0/8)
Utils	50% (1/2)	10% (3/28)	15% (6/40)
View	0% (0/13)	0% (0/75)	0% (0/441)
Main	0% (0/1)	0% (0/1)	0% (0/12)

Figura 23: Cobertura Demo 1



4.3.2 Demo2

Element	Class, %	Method, %	Line, %	Branch, %
Aluguer	100% (1/1)	57% (15/26)	32% (30/93)	0% (0/30)
AluguerNaoExisteException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
AluguerTest	100% (1/1)	100% (14/14)	100% (33/33)	100% (0/0)
CarroEletrico	0% (0/1)	0% (0/6)	0% (0/15)	0% (0/6)
CarroGasolina	0% (0/1)	0% (0/6)	0% (0/15)	0% (0/6)
CarroHibrido	0% (0/1)	0% (0/6)	0% (0/15)	0% (0/6)
Classificação				
Cliente	100% (1/1)	35% (5/14)	25% (10/40)	0% (0/16)
ClienteTest	100% (1/1)	100% (4/4)	100% (12/12)	100% (0/0)
ComparadorAutonomia	0% (0/1)	0% (0/1)	0% (0/3)	0% (0/2)
ComparadorKm	0% (0/1)	0% (0/1)	0% (0/3)	0% (0/2)
ComparadorNAluguer	0% (0/1)	0% (0/1)	0% (0/3)	0% (0/2)
ComparadorPreco	0% (0/1)	0% (0/1)	0% (0/3)	0% (0/2)
Coordinate	100% (1/1)	61% (8/13)	55% (20/36)	10% (1/10)
CoordinateManager	0% (0/1)	0% (0/10)	0% (0/46)	0% (0/38)
Input	0% (0/1)	0% (0/7)	0% (0/112)	0% (0/14)
Menu	0% (0/1)	0% (0/6)	0% (0/36)	0% (0/18)
NaoEfetuouNenhumAluguerE...	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
NaoExistemAlugueresException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
NaoExistemClientesException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
NaoExistemVeiculosDisponiv...	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
ParDados	0% (0/1)	0% (0/11)	0% (0/36)	0% (0/22)
ParseDados	0% (0/1)	0% (0/6)	0% (0/79)	0% (0/4)
PasswordIncorrectaException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
Proprietario	0% (0/1)	0% (0/6)	0% (0/12)	0% (0/6)
UmCarroJa	0% (0/2)	0% (0/56)	0% (0/370)	0% (0/198)
UmCarroJaApp	0% (0/2)	0% (0/49)	0% (0/742)	0% (0/199)
Utilizador	100% (1/1)	38% (7/18)	25% (14/54)	0% (0/18)
UtilizadorJaExisteException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
UtilizadorNaoExisteException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
UtilizadorTest	100% (1/1)	100% (6/6)	100% (15/15)	100% (0/0)
Veiculo	100% (1/1)	33% (12/36)	21% (24/110)	0% (0/30)
VeiculoIndisponivelException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
VeiculoJaExisteException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
VeiculoNaoESeuException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
VeiculoNaoExisteException	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
VeiculoTest	100% (1/1)	100% (11/11)	100% (26/26)	100% (0/0)
Weather	0% (0/2)	0% (0/2)	0% (0/25)	0% (0/2)

Figura 24: Cobertura Demo 2



4.4 Gerador de Inputs

Como forma de testes, foi criado um gerador de ficheiros *log*, que respeita o formato inicial, disponibilizado pelos docentes, da seguinte maneira:

- **Proprietário**

NovoProp: *Nome e Apelido, NIF, Email, Concelho*

- **Cliente**

NovoCliente: *Nome e Apelido, NIF, Email, Concelho, Posição X, Posição Y*

- **Carro**

NovoCarro: *Tipo de Combustível, Marca, Matricula, NIF Proprietário, Velocidade Média, Preço por KM, Consumo por KM, Autonomia, Posição X, Posição Y*

- **Aluguer**

Aluguer: *NIF Cliente, Posição destino X, Posição destino Y, Tipo Combustível, Preferência*

- **Classificação**

Classificar: *Matricula ou NIF, Nota (0-100)*

De forma a garantir que os dados gerados sejam mais aproximados à realidade, forma definidas várias estratégias de validação e criação. Passamos a analisar os invariantes definidos.

- De forma a normalizar os **Nomes Próprios e Apelidos**, foram criadas duas listas, com os 40 nomes e apelidos mais comuns em Portugal.

- Com a finalidade de manter a coerência criamos uma lista com todos os **concelhos** da zona do Minho.

- Para as **marcas** automóveis, fomos consultar a frequência das principais marcas de forma a atribuir marcas, às viaturas da UMCarroJá, com a frequência encontrada.

- Da mesma forma, que o invariante anterior, fomos consultar a frequência de carros a **gasolina, híbrido e eléctrico**.

- Em relação aos possíveis **formatos de matricula** (00-XX-00, 00-00-XX, XX-00-00), definimos que as chances de surgir uma matricula seria de 80%, 15% e 5%, das mais recentes para as mais antigas.

- De maneira a garantir coerência na **velocidade, preço, consumo e autonomia**, foram analisados este campos em relação à realidade e, da mesma forma, definidos os seus limites possíveis e frequência de criação.

- Para as **posições** tivemos em consideração as coordenadas, limitantes, X e Y em Portugal, para isso X tem limite em [-9.32,-6.32] e Y tem limite em [37.0,42.0].

- Como norma de **classificação**, assumimos que, 80% atribui uma nota entre 50 e 80, apenas 1% atribui uma nota inferior a 40, 4% atribui uma nota negativa superior a 40 e 15% atribui nota superior a 80.



Com isto, utilizando o seguinte comando é possível executar o programa, em **Haskell**, que gera o ficheiro de logs.

`$>.\generator [tamanho] > [ficheiro de saída]`

Sendo o tamanho um valor de multiplicação para a quantidade de clientes, proprietários, carros, avaliações e alugueres, onde o valor mínimo é 1. Logo o número de entidades criadas é:

- **Proprietários** – $200 * n$
- **Clientes** – $600 * n$
- **Carros** – $2000 * n$
- **Alugueres** – $500 * n$
- **Classificações** – $2000 * n$

Por fim, a geração automática do ficheiro de logs é feita por etapas, seguindo as normas de criação.

1. Geração de NIFs para Proprietários
2. Geração de NIFs para Clientes
3. Criação de Proprietários
4. Criação de Clientes
5. Geração de Matriculas
6. Criação de Carros
7. Criação de Alugueres
8. Criação de Classificações

Deste modo, o executável de geração aleatória respeita todas as condições para criação de um ficheiro logs aproximado à realidade.



5 Análise de Desempenho da Aplicação

Por forma a fazer uma análise detalhada do desempenho da aplicação **UmCarroJá** relativamente ao tempo de execução e consumo de energia, é necessário ter atenção as especificações da máquina onde estes testes são efetuados. Desta forma, podemos fazer uma análise mais assertiva, tendo em conta as condições onde este balanço é realizado. Neste contexto, as especificações mais relevantes do computador são as seguintes:

- **SO:** Ubuntu 18.04.3 LTS;
- **CPU:** *Intel Core i7-8750H*, 2.20GHz x 12;
- **Memória RAM:** 16 Gb, 2400 MHz;
- **Energia:** 71 Wh;

Neste contexto, utilizamos a interface RAPL (*Running Average Power Limit*) para proceder á verificação dos consumos de energia, relativos aos dois projetos. Para tal, decidimos analisar o consumo de energia relativamente a três ficheiros de *logs*: o fornecido pelos professores e outros dois criados por o grupo (um de tamanho razoavelmente pequeno e outro maior).

Vamos também analisar mais detalhadamente o **red smell streams** que como vamos ver mais a frente afecta bastante os consumos e a performance do código. Isto deve-se maioritariamente ao facto de a classe que implementa programação funcional ainda ser relativamente "nova", ou seja, ainda não se encontra tão optimizada como seria desejado.

5.1 Demo1

Em relação a este projeto, optamos por fazer o consumo de energia e verificar o tempo de execução relativamente a sete parâmetros que achamos relevantes:

- Carregamento do ficheiro de *logs*;
- O método que verifica os 10 melhores clientes em termos de alugueres;
- O método que regista um utilizador;
- O método que adiciona um veículo ao sistema;
- O método que permite realizar o *log in* de um utilizador no sistema;
- O método que apresenta os melhores tempos dos melhores clientes;
- O método que apresenta as viagens dos melhores clientes;



5.1.1 logsPOO_carregamentoInicial

Carregamento ficheiro logs	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.46933	11.04143	11.61956	276.7
Com Refactor	0.56535	13.49201	14.11139	267.7
Com Refactor (Streams)	0.40115	9.87419	10.38219	257.9
Com Auto Refactor	0.56285	13.10953	13.71637	269.4

Tabela 5: Carregamento ficheiro logs com logsPOO_carregamentoInicial

Melhores Clientes	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.030695	0.77697	0.82355	20.5
Com Refactor	0.03231	0.85719	0.88773	16.8
Com Refactor (Streams)	0.02883	0.70421	0.74047	18.3
Com Auto Refactor	0.03023	0.76673	0.80081	16.2

Tabela 6: Melhores Clientes com logsPOO_carregamentoInicial

Registrar	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.00013	0.00638	0.00453	0.3
Com Refactor	0.00029	0.00796	0.01095	0.5
Com Refactor (Streams)	0.0001	0.00574	0.00351	0.4
Com Auto Refactor	0.00051	0.00908	0.02646	0.1

Tabela 7: Registrar com logsPOO_carregamentoInicial

Adicionar Carro	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.0001	0.00595	0.00427	0.3
Com Refactor	0.00034	0.00828	0.01217	0.2
Com Refactor (Streams)	0.00017	0.00731	0.00466	0.3
Com Auto Refactor	0	0.01128	0	0.5

Tabela 8: Adicionar Carro com logsPOO_carregamentoInicial

Login	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.00042	0.00394	0.00743	0.1
Com Refactor	0.00011	0.00609	0.00467	0.1
Com Refactor (Streams)	0.00013	0.00635	0.00373	0.3
Com Auto Refactor	0.00037	0.00859	0.00728	0.3

Tabela 9: Login com logsPOO_carregamentoInicial



Tempos de Melhores Clientes	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.0411	0.98048	1.03278	22.4
Com Refactor	0.03778	1.00224	1.04869	17.8
Com Refactor (Streams)	0.01808	0.5277	0.55978	15
Com Auto Refactor	0.03447	0.93373	0.97715	16.5

Tabela 10: *Tempos de Melhores Clientes com logsPOO_carregamentoInicial*

Viagens de melhores clientes	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.04131	1.03997	1.09451	24.8
Com Refactor	0.03291	0.89758	0.94182	15.8
Com Refactor (Streams)	0.02346	0.57859	0.60417	15.4
Com Auto Refactor	0.0299	0.77328	0.80577	13.6

Tabela 11: *Viagens de Melhores Clientes com logsPOO_carregamentoInicial*

5.1.2 Large

Carregamento ficheiro logs	DRam	CPU	Package	Tempo Execução
Sem Refactor	3.97291	86.77091	92.3949	2820.5
Com Refactor	4.15744	86.22073	91.85996	2763.7
Com Refactor (Streams)	3.98113	87.66478	93.20988	2796.3
Com Auto Refactor	4.54245	88.78002	94.62709	2749.7

Tabela 12: *Carregamento ficheiro logs com Large*

Melhores Clientes	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.12238	3.15486	3.33388	90.2
Com Refactor	0.12887	3.32548	3.51652	81.5
Com Refactor (Streams)	0.12286	3.01981	3.21517	92.1
Com Auto Refactor	0.13607	3.46433	3.63136	80.3

Tabela 13: *Melhores Clientes com Large*

Registrar	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.00032	0.00641	0.00972	0.4
Com Refactor	0.00031	0.01099	0.01052	0.2
Com Refactor (Streams)	0.00035	0.00881	0.01213	0.2
Com Auto Refactor	0.00011	0.00864	0.00391	0.1

Tabela 14: *Registrar com Large*



Adicionar Carro	DRam	CPU	Package	Tempo Execução
Sem Refactor	0	0.00527	0	0.2
Com Refactor	0.00105	0.01147	0.02222	0.3
Com Refactor (Streams)	0.00034	0.00591	0.00494	0.2
Com Auto Refactor	0	0.0085	0.00612	0.3

Tabela 15: Adicionar Carro com Large

Login	DRam	CPU	Package	Tempo Execução
Sem Refactor	0	0.00535	0.00482	0.1
Com Refactor	0.00023	0.0081	0.0121	0.2
Com Refactor (Streams)	0.00031	0.00539	0.00453	0.5
Com Auto Refactor	0.00018	0.00787	0.00701	0.1

Tabela 16: Login com Large

Tempos de Melhores Clientes	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.13162	3.3616	3.54296	90.7
Com Refactor	0.14509	3.66923	3.85129	89.9
Com Refactor (Streams)	0.11702	2.99698	3.17341	82.9
Com Auto Refactor	0.1537	3.85509	4.0467	83.9

Tabela 17: Tempos de Melhores Clientes com Large

Viagens de Melhores Clientes	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.10833	2.76988	2.91947	76.5
Com Refactor	0.11616	2.73273	2.89239	75.5
Com Refactor (Streams)	0.12173	2.89762	3.07394	81.3
Com Auto Refactor	0.13398	3.01202	3.18085	73.7

Tabela 18: Viagens de Melhores Clientes com Large

5.1.3 Mini

Carregamento ficheiro logs	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.79481	19.40978	20.42648	490.3
Com Refactor	1.02849	23.72703	24.85604	493.6
Com Refactor (Streams)	0.77753	18.34432	19.33546	474.4
Com Auto Refactor	0.98628	22.31411	23.44163	500.9

Tabela 19: Carregamento ficheiro logs com Mini



Melhores Clientes	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.03669	0.95645	1.01507	24.2
Com Refactor	0.04113	1.04878	1.0884	19.4
Com Refactor (Streams)	0.03216	0.85118	0.88956	22.5
Com Auto Refactor	0.03557	0.91702	0.94807	19.2

Tabela 20: *Melhores Clientes com Mini*

Registrar	DRam	CPU	Package	Tempo Execução
Sem Refactor	0	0.00732	0	0.1
Com Refactor	0.00028	0.00756	0.01279	0.1
Com Refactor (Streams)	0	0.00656	0	0.4
Com Auto Refactor	0	0.00811	0	0.1

Tabela 21: *Registrar com Mini*

Adicionar Carro	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.00043	0.00679	0.01002	0.5
Com Refactor	0.00037	0.00791	0.0133	0.5
Com Refactor (Streams)	0.00045	0.00603	0.00905	0
Com Auto Refactor	0.00033	0.00946	0.00548	0.2

Tabela 22: *Adicionar Carro logs com Mini*

Login	DRam	CPU	Package	Tempo Execução
Sem Refactor	0	0.00436	0.00519	0.2
Com Refactor	0	0.00801	0	0.2
Com Refactor (Streams)	0.00059	0.0053	0.01316	0.2
Com Auto Refactor	0.00059	0.00797	0.01599	0.1

Tabela 23: *Login com Mini*

Tempos de Melhores Clientes	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.03891	1.068	1.12211	26
Com Refactor	0.05383	1.34055	1.40905	22.3
Com Refactor (Streams)	0.03187	0.76627	0.8207	19.7
Com Auto Refactor	0.04579	1.17842	1.23506	23.4

Tabela 24: *Tempos de Melhores Clientes com Mini*

Viagens de Melhores Clientes	DRam	CPU	Package	Tempo Execução
Sem Refactor	0.03535	0.90188	0.94297	23.1
Com Refactor	0.03646	0.9295	0.97062	16.1
Com Refactor (Streams)	0.0244	0.67509	0.71142	18.8
Com Auto Refactor	0.03234	0.79443	0.84407	15.9

Tabela 25: *Viagens de Melhores Clientes com Mini*



5.2 Demo 2

Relativamente a este projeto, tal como anteriormente, o grupo decidiu averiguar o consumo de energia e verificar o tempo de execução relativamente a seis parâmetros:

- O carregamento do ficheiro do logs para o projeto;
- O método que verifica os 10 melhores clientes em termos de alugueres;
- O método que regista um utilizador;
- O método que adiciona um veículo ao sistema;
- O método que permite o início de sessão de um utilizador no sistema;
- O método que constituí os 10 melhores clientes em termos de quilómetros efetuados;

É de salientar, tal como se procedeu anteriormente, que foram realizados 10 execuções por cada tipo de refator, por forma a tornar os valores resultantes mais acertivos. Os valores gerados nas tabelas representam a média dessas 10 execuções.

5.2.1 logsPOO_carregamentoInicial

Carregamento ficheiro logs	DRam	CPU	Package	Tempo de Execução
Sem Refactor	2.06946	38.82888	41.16296	970.5
Com Refactor	1.72605	33.37742	35.30006	784.5
Com Auto Refator	2.18342	39.48154	41.65881	968.2
Com Refactor (Streams)	1.32033	26.70349	28.09232	611.9

Tabela 26: Carregamento ficheiro logs com logsPOO_carregamentoInicial

Registrar Utilizador	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.00030	0.00641	0.00468	0.2
Com Refactor	0.00025	0.00559	0.00322	0.4
Com Auto Refator	0.0	0.00699	0.0	0.3
Com Refactor (Streams)	0.00011	0.00656	0.00599	0.6

Tabela 27: Registrar com logsPOO_carregamentoInicial

Adiciona Veículo	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.00032	0.00383	0.00482	0.2
Com Refactor	0.00011	0.00598	0.01125	0.2
Com Auto Refator	0.00012	0.00656	0.00807	0.3
Com Refactor (Streams)	0.00011	0.00656	0.00599	0.6

Tabela 28: Adicionar carro com logsPOO_carregamentoInicial



Iniciar Sessão	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.0	0.00378	0.0	0.2
Com Refactor	0.00037	0.00356	0.01022	0.3
Com Auto Refator	0.00012	0.00387	0.0	0.1
Com Refactor (Streams)	0.00012	0.00234	0.00357	0.2

Tabela 29: *Login com logsPOO_carregamentoInicial*

Melhores Clientes (Kilómetros)	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.0022	0.0444	0.04509	1.4
Com Refactor	0.0019	0.05209	0.04885	1.4
Com Auto Refator	0.00287	0.05274	0.05569	1.2
Com Refactor (Streams)	0.00127	0.02188	0.02691	0.9

Tabela 30: *Melhores Clientes (km) com logsPOO_carregamentoInicial*

5.2.2 Large

Carregamento ficheiro logs	DRam	CPU	Package	Tempo de Execução
Sem Refactor	7.88351	127.66588	137.16061	5244.5
Com Refactor	6.98155	109.78915	117.77985	4269.2
Com Auto Refator	8.40006	137.02126	137.02126	5419.1
Com Refactor (Streams)	5.08705	86.00192	91.78955	3077.5

Tabela 31: *Carregamento ficheiro logs com Large*

Melhores Clientes (Alugueres)	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.00455	0.16949	0.1676	3.4
Com Refactor	0.00501	0.17731	0.1907	3.5
Com Auto Refator	0.00524	0.17242	0.18334	3.4
Com Refactor (Streams)	0.00435	0.15324	0.15924	2.9

Tabela 32: *Melhores Clientes (Alugueres) com Large*

Registrar Utilizador	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.000124	0.00676	0.00518	0.4
Com Refactor	0.00010	0.00546	0.00320	0.3
Com Auto Refator	0.0004	0.00802	0.0194	0.3
Com Refactor (Streams)	0.00035	0.00814	0.00507	0.5

Tabela 33: *Registrar com Large*

5.2.3 Mini



Adiciona Veículo	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.00013	0.00584	0.0	0.2
Com Refactor	0.0004	0.00659	0.01858	0.2
Com Auto Refator	0.00013	0.00761	0.00517	0.2
Com Refactor (Streams)	0.00023	0.00649	0.00490	0.3

Tabela 34: Adicionar carro com Large

Iniciar Sessão	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.0	0.00449	0.00548	0.1
Com Refactor	0.00010	0.00573	0.00447	0.3
Com Auto Refator	0.0	0.00654	0.0	0.1
Com Refactor (Streams)	0.00013	0.00463	0.00661	0.1

Tabela 35: Login com Large

Melhores Clientes (Kilómetros)	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.00259	0.10088	0.08756	2
Com Refactor	0.00311	0.09772	0.10074	2
Com Auto Refator	0.00305	0.10319	0.0985	2.2
Com Refactor (Streams)	0.00284	0.09333	0.09742	2.2

Tabela 36: Tempos de Melhores Clientes (km) com Large

Carregamento ficheiro logs	DRam	CPU	Package	Tempo de Execução
Sem Refactor	1.36161	32.85292	34.37971	865.3
Com Refactor	1.29393	29.96796	31.39994	744.2
Com Auto Refator	1.48829	35.34703	37.05529	894.3
Com Refactor (Streams)	0.95975	23.66612	24.72881	562

Tabela 37: Carregamento ficheiro logs com Mini

Melhores Clientes (Alugueres)	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.00305	0.10111	0.1144	2
Com Refactor	0.00339	0.10247	0.11044	2,1
Com Auto Refator	0.00307	0.09922	0.11434	2.2
Com Refactor (Streams)	0.00215	0.07865	0.07794	1.6

Tabela 38: Melhores Clientes (Alugueres) com Mini

Registrar Utilizador	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.00024	0.00699	0.01599	0.2
Com Refactor	0.00041	0.00579	0.01486	0.1
Com Auto Refator	0.00041	0.00645	0.01528	0.2
Com Refactor (Streams)	0.0	0.00627	0.0	0.1

Tabela 39: Registar com Mini



Adiciona Veículo	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.0	0.00743	0.0	0.7
Com Refactor	0.00017	0.00695	0.00524	0.3
Com Auto Refator	0.00026	0.00699	0.01093	0.1
Com Refactor (Streams)	0.00009	0.00707	0.0050	0.4

Tabela 40: Adicionar carro com Mini

Iniciar Sessão	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.00023	0.00524	0.00906	0.0
Com Refactor	0.0	0.00501	0.0	0.2
Com Auto Refator	0.0	0.00556	0.0	0.5
Com Refactor (Streams)	0.0	0.00549	0.0	0.3

Tabela 41: Login com Mini

Melhores Clientes (Kilómetros)	DRam	CPU	Package	Tempo de Execução
Sem Refactor	0.00112	0.05032	0.04745	1
Com Refactor	0.00144	0.05172	0.05369	1,1
Com Auto Refator	0.0014	0.04601	0.05128	1
Com Refactor (Streams)	0.97706	0.04681	0.04776	1.1

Tabela 42: Melhores Clientes (km) com Mini

5.3 Análise de Resultados (Com Refactoring vs Sem Refactoring)

Tendo os resultados obtidos a partir da utilização do **RAPL** com e sem Refactoring, podemos agora comparar os resultados obtidos para verificar se houve ou não melhorias, em termos do tempo de execução, consumo de energia e averiguar se estas melhorias são ou não significativas.

Desta forma, consideramos o tempo de execução e três parâmetros de avaliação relativamente ao consumo de energia, respetivamente: **DRAM** (*Dynamic Random-Access Memory*), **CPU** (*Central Process Unit*) e **GPU** (*Graphics Processing Unit*).

Tal como se pode observar pelos resultados em formato tabular, para os diferentes projetos, ficheiros de *logs* e para métodos distintos, os resultados com a aplicação do refator são significativamente benéficos em relação ao sem refator. Tendo em atenção algumas particularidades, o auto refator (aplicado pelo *plugin* do *IntelliJ*) apresenta resultados substancialmente piores em termos energéticos, mas melhores em performance (tempos de execução). Também nos deparamos que com a eliminação do *red smell streams*, na maioria dos casos obtém-se melhorias bastante consideráveis.

Contudo, podemos concluir que fazer Refactoring no código fonte dos projetos tem uma ligeira vantagem, em termos do tempo de execução do programa. Podemos também afirmar, que houve um ganho mais significativo no consumo de energia e memória quando é feito Refactoring.



Conclusão e Trabalho Futuro

Este relatório têm como objetivos, a análise e compreensão de 4 tarefas distintas.

Primeiramente, após a análise da qualidade do código das duas aplicações, verificamos os parâmetros de confiabilidade, manutenção, testabilidade, portabilidade e reutilização. Com isto, utilizando o **SonarQube**, analisamos as métricas de **NCLOC, n.º de Funções/Classes, n.º de Statements**, entre outro. Verificamos que o projeto 1 apresenta 33 bugs, 10 vulnerabilidades, 327 code smells e uma percentagem de duplicados na ordem dos 3.7%, e o projeto 2 apresenta 14 bugs, 1 vulnerabilidade, 131 code smells e exibe uma densidade de 1% de duplicados em relação ao código. Com estes valores seguimos para uma análise comprehensiva e extendida dos dois projetos.

De seguida, foi-nos proposto o refactor dos dois projetos, para isto, verificamos a existência de vários *smells*, como, duplicação e desuso de código, métodos/funções/classes extensos e parametrização excessiva. Após a identificação e refactor dos *code smells*, previamente catalogados, tivemos a sensibilidade de estudar outro tipo de *smells*, designados *red smells*, levando-nos a uma nova análise da qualidade do código, após o refactor.

Seguidamente, já numa etapa de testes, foi criado um gerado para os ficheiros de logs, desenvolvido em **haskell**, no qual, foi implementado tendo em atenção vários invariantes de modo a melhor simular a realidade. Tendo em atenção os novos ficheiros de logs gerados, desenvolvemos vários casos de teste, começando pelos testes unitários em **JUnit**, seguindo para a geração automática com a ferramenta **EvoSuite**. Por fim, utilizando o **JaCoCo**, obtivemos a cobertura do código através dos teste unitários criados anteriormente.

Por fim, de modo a verificar o desempenho aplicativos dos dois projetos, utilizamos a interface **RAPL**, para a verificação dos consumos energéticos. De seguida, procedemos à comparação, sobre os dados obtidos pelo **RAPL**, entre os projetos sem refactor, com refactor, com refactor (Streams) e com auto refactor, tendo em atenção as funções e métodos onde o refactor foi feito.

Com isto, apesar do trabalho desenvolvido, achamos que futuramente o código fornecido para análise pode ser ainda mais explorado, em termos de ser realizado mais Refactoring, no sentido de melhorar ainda mais a compreensão do código e também do seu tempo de execução e consumos de energia e memória.

Concluindo, achamos que, ao longo do estudo deste projeto UmCarroJa, conseguimos obter conhecimentos que nos permitem testar por completo um programa, ou uma aplicação, sendo eles a análise e teste da Aplicação UmCarroJá, a criação de testes para a correta verificação do software UmCarroJa e a melhoria da qualidade do software para futuros desenvolvimentos.