UNIVERSIDADE DO MINHO

ANÁLISE E TESTES DE SOFTWARE

ANÁLISE E TESTES DE PROJECTOS UMCARROJÁ

Grupo:
João Nunes - A82300
Luís Braga - A81088
Bárbara Cardoso - A80453
Bruna Cruz - PG41905

Braga, Portugal
2 de Fevereiro de 2020

Conteúdo

Lista de Figuras

1 Introdução

O presente relatório documenta o projecto proposto na unidade curricular de **Análise e Testes de Software**. Este projecto consiste na realização de um conjunto de análises e testes a sistemas de software, desenvolvidos na linguagem de programação Java, no contexto da unidade curricular de *Programação Orientada aos Objectos* (**POO**), no ano lectivo de 2018/2019. A aplicação a desenvolver em **POO**, denominada **UMCarroJá**, traduzia-se num sistema de gestão de um serviço de aluguer de veículos particulares na internet.

Foram fornecidas duas soluções do trabalho prático **UMCarroJá** (demo1 e demo2) de forma a alcançar os cinco objectivos base deste projecto:

1. Analisar a qualidade do código fonte, identificando os *bad smells* e o seu *technical debt*.
2. Realizar o *refactoring* de modo a eliminar *bad smells* e de modo a reduzir ou até eliminar o *technical debt*.
3. Testar o software de modo a garantir que cobre todos os requisitos do enunciado da aplicação **UMCarrojá**.
4. Gerar aleatoriamente *inputs* para a aplicação.
5. Analisar a performance em termos de tempo de execução e consumo de energia da versão inicial e a obtida depois de eliminados os smells.

2 Tarefa 1: Qualidade do código fonte

De modo a analisar a qualidade do código fonte, foi utilizado o *SonarQube*, onde nesta plataforma é possível inspeccionar o código fonte dos projectos, sendo, então, possível avaliar o projecto tendo em conta os seus eventuais *bugs*, *bad smells* e outras métricas também disponibilizadas por esta plataforma.

O código relativo aos dois projectos, *demo1* e *demo2*, foi também avaliado no IDE *IntelliJ*, através do uso de um plugin *CodeMR*, que também possui o intuito de analisar a qualidade do código através do uso de várias métricas de software, tais como, a coesão do código, *coupling* e a complexidade do código. Gerando, no final, gráficos que relacionam estas métricas com o código avaliado.

2.1 Estrutura dos projectos

2.1.1 demo1

A estrutura da solução **demo1**, relativamente a directorias, packages e classes implementadas, pode ser visualizada no seguinte esquema:

```
src/
    main/
        controller/
            Controller.java
        Exceptions/
            CarExistsException.java
            InvalidCarException.java
            InvalidNewRegisterException.java
            InvalidNewRentalException.java
            InvalidNumberOfArgumentsException.java
            InvalidRatingException.java
            InvalidTimeIntervalException.java
            InvalidUserException.java
            NoCarAvailableException.java
            UnknowCarTypeException.java
            UnknowCompareTypeException.java
            UserExistsException.java
            WrongPasswordExection.java
        Model/
            Car.java
            Cars.java
```

```
            Client.java
            Owner.java
            Parser.java
            Rental.java
            Rentals.java
            Traffic.java
            UMCarroJa.java
            User.java
            Users.java
            Weather.java
        Utils/
            Point.java
            StringBetter.java
        View/
            ViewModel/
                AutonomyCar.java
                CheapestNearCar.java
                NewLogin.java
                RateOwnerCar.java
                RegisterCar.java
                RegisterUser.java
                RentCarSimple.java
                SpecificCar.java
                TimeInterval.java
            ITable.java (interface)
            Menu.java
            Table.java
        Main.java
```

Analisando a estrutura acima, podemos verificar que esta solução, com quarenta e três classes distintas, apresenta um maior grau de organização e compreensibilidade em termos de localização de componentes, relativamente à outra solução fornecida.

Numa primeira análise, o maior número de classes também poderá significar uma melhor aplicação do paradigma de programação orientada aos objectos, uma vez que, possivelmente, implica a separação de responsabilidades entre componentes. Este maior número de classes também se poderá traduzir num menor grau de código duplicado.

2.1.2 demo2

Relativamente à estrutura do projeto demo2, este apresenta a seguinte estrutura de ficheiros e pastas:

```
src/
    main/
        java/
            Aluger.java
            AlugerNaoExisteException.java
            CarroEletrico.java
            CarroGasolina.java
```

```
CarroHibrido.java
Classificação.java (interface)
Cliente.java
CompradorAutonomia.java
CompradorKm.java
CompradorNAluguer.java
CompradorPreco.java
Coordinate.java
CoordinateManager.java
Input.java
Menu.java
NaoEfetuouNenhumAluguerException.java
NaoExistemAlugueresException.java
NaoExistemClientesException.java
NaoExistemVeiculosDisponiveisException.java
ParDatas.java
ParseDados.java
PasswordIncorretaException.java
Proprietario.java
UmCarroJa.java
UmCarroJaAPP.java
Utilizador.java
UtilizadorJaExisteException.java
UtilizadorNaoExisteException.java
Veiculo.java
VeiculoIndisponivelException.java
VeiculoJaExisteException.java
VeiculoNaoESeuException.java
VeiculoNaoExisteException.java
Weather.java
```

Portanto, tal como é possível observar pela estrutura do código anterior, este possui trinta e quatro classes diferentes, e, como é possível observar, este encontra-se desorganizado, pelo que seria necessário organizar melhor as classes dentro de *packages*. Uma vez que, sem uma melhor organização, é muito mais dispendioso o processo de, por exemplo, acrescentar uma *feature* à aplicação.

Tendo finalizado a análise da estrutura de ambos os projectos, procedeu-se à análise da qualidade do código fonte através das ferramentas anteriormente referidas, ou seja, o *SonarQube* e o *CodeMR*.

2.2 Bad Smells

2.2.1 SonarQube

O *SonarQube* avalia métricas que podem variar desde o número total de *bugs* e vulnerabilidades, que eventualmente podem colocar em causa a integridade do sistema, até *code smells*, que embora não sendo erros, são falhas no design do software que torna o código menos legível e que por consequência, torna mais difícil a evolução deste. A última métrica principal que o

SonarQube avalia são os *Security Hotspots*, que indica locais no código que são sensíveis e que poderão vir a tornar-se numa vulnerabilidade no futuro. Compete depois ao programador analisar esse código e identificar as potenciais vulnerabilidades.

Como tal, foi corrido o *SonarQube* nos dois projectos, de modo a avaliar o número total destas métricas.

Discorrendo mais sobre estas métricas, podemos defini-las concretamente da seguinte forma:

- **Bug**: Significa que há algo de errado com o código e que deverá ser corrigido imediatamente.
- **Vulnerability**: Um problema relacionado com segurança, que mais tarde se poderá agravar se houver alguém a atacar o código.
- **Code Smells**: Um problema relacionado com a manutenibilidade do código. Será muito mais complicado modificar o código.
- **Security Hotspot**: Um problema relacionado com segurança, que destaca um segmento que utiliza uma API sensível e com pouca segurança.

O *SonarQube* apresenta alguns conceitos com os quais podemos classificar os problemas do código a ser analisado, nomeadamente:

- **Blocker**: Alta probabilidade de ter impacto no comportamento da aplicação.

  Por exemplo: fuga de memória, conexão JDBC por fechar.
- **Critical**: Baixa probabilidade de ter impacto no comportamento da aplicação ou representa uma falha de segurança.

  Por exemplo: SQL injection, um bloco catch vazio.
- **Major**: Falha na qualidade do produto a ser desenvolvido que pode ter grande impacto na produtividade de quem está a desenvolver.

  Por exemplo: parâmetros não utilizados, código duplicado, código que nunca irá ser percorrido.
- **Minor**: Falhas de qualidade que têm pouco impacto na produtividade do produto a ser desenvolvido.

  Por exemplo: linhas de código demasiado longas, declarações de *switch* devem ter pelo menos três *cases*.

2.2.2 SonarQube - demo1

Type	Blocker	Critical	Major	Minor	Total
Bug	2	2	1	9	14
Vulnerability	-	-	-	1	1
Code smell	6	41	24	59	130
Security Hotspot	-	-	-	-	22

Tabela 2.1: Número total de métricas avaliadas pelo SonarQube no projeto demo1.

2.2.3 SonarQube - demo2

Type	Blocker	Critical	Major	Minor	Total
Bug	4	-	2	27	33
Vulnerability	-	-	-	10	10
Code smell	10	47	87	176	321
Security Hotspot	-	-	-	-	16

Tabela 2.2: Número total de métricas avaliadas pelo SonarQube no projeto demo2.

Após correr a análise do SonarQube no demo2, foi possível extrair os número anteriores, onde, como é possível observar, o demo2 é consideravelmente pior que o demo1, possuíndo um número bastante elevado de bugs, sendo que 4 desses bugs potencialmente alteram o comportamento da aplicação, ou seja, são *blockers* bem como 2 major bugs e 27 minor bugs. A aplicação possui também um número elevado de code smells, sendo que 10 deles são blockers e 47 são críticos, ou seja, possui uma probabilidade mais pequena de afetar o comportamento da aplicação, contudo têm de ser corrigidos mesmo assim. Possui também 87 major code smells e 176 minor. Há também 16 pedaços de código que têm de ser analisados de modo a averiguar se constituem ou não uma potencial vulnerabilidade do sistema.

2.3 Red Smells

2.3.1 jStanley

O jStanley é um plugin para o Eclipse IDE, que permite não só analisar a eficiência energética de código Java, como também aplicar técnicas de refactoring sobre o código analisado.

2.3.2 demo1

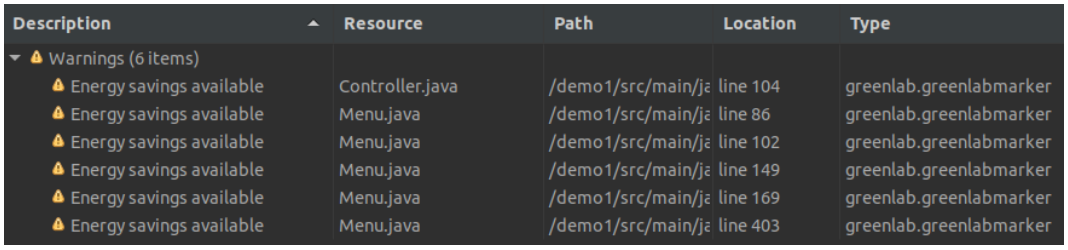


Figura 2.1: Deteção de Red Smells do demo1 feita pelo jStanley

2.3.3 demo2

Description	Resource	Path	Location	Type
Warnings (16 items)				
Energy savings available	CoordinateManager.java	/demo2/src/main/ja	line 181	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 184	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 204	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 248	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 371	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 500	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 559	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 570	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 645	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 679	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 709	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 763	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 783	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 824	greenlab.greenlabmarker
Energy savings available	UmCarroJa.java	/demo2/src/main/ja	line 858	greenlab.greenlabmarker
Energy savings available	UmCarroJaApp.java	/demo2/src/main/ja	line 790	greenlab.greenlabmarker

Figura 2.2: Deteção de Red Smells do demo2 feita pelo jStanley

Após a execução do plugin jStanley, foi possível observar Red Smells tanto na demo1 como na demo2, representados pelas imagens apresentadas anteriormente.

Observou-se que existem 16 Red Smells na demo2, enquanto que na demo1 foram encontrados apenas 6 Red Smells. Com estes valores foi possível observar que a demo2 tem menos optimizações energéticas que a demo1, esperando-se assim um maior maior consumo energético na demo2. Na demo1, os Red Smells situam-se nas classes "Controller"e "Menu", e na demo2 estes foram observados nas classes "CoordinateManager", "UmCarroJaApp"e "UmCarroJa", sendo esta última classe a mais afetada contendo 14 Red Smells.

2.4 Technical Debt

2.4.1 demo1

Considerando a análise realizada pelo *SonarQube*, podemos observar que o tempo estimado para eliminar todas as características indesejadas que o código possui actualmente ronda os três dias. Esta análise contempla *bugs*, *smells*, *vulnerabilities* entre outras métricas incluídas na análise.

O *overview* desta análise encontra-se na seguinte figura.

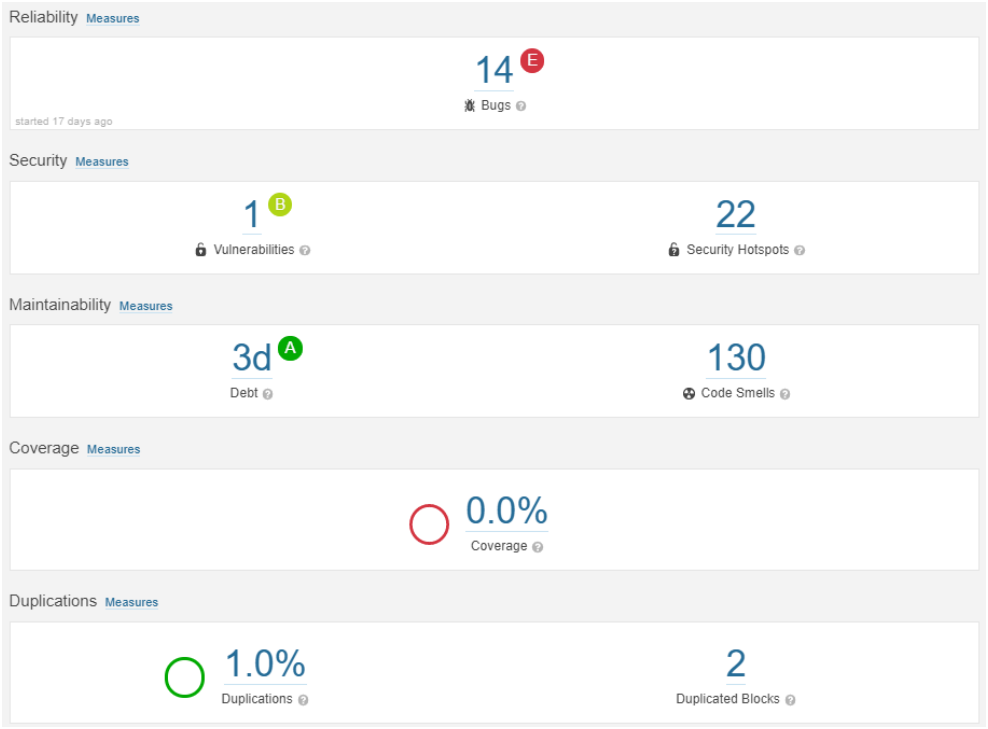


Figura 2.3: Overview do projecto demo1.

Para além desta análise geral, é possível observar com detalhe o *technical debt* de cada classe. Sendo que, a classe *Controller* seria a que demoraria mais tempo a ser corrigida.

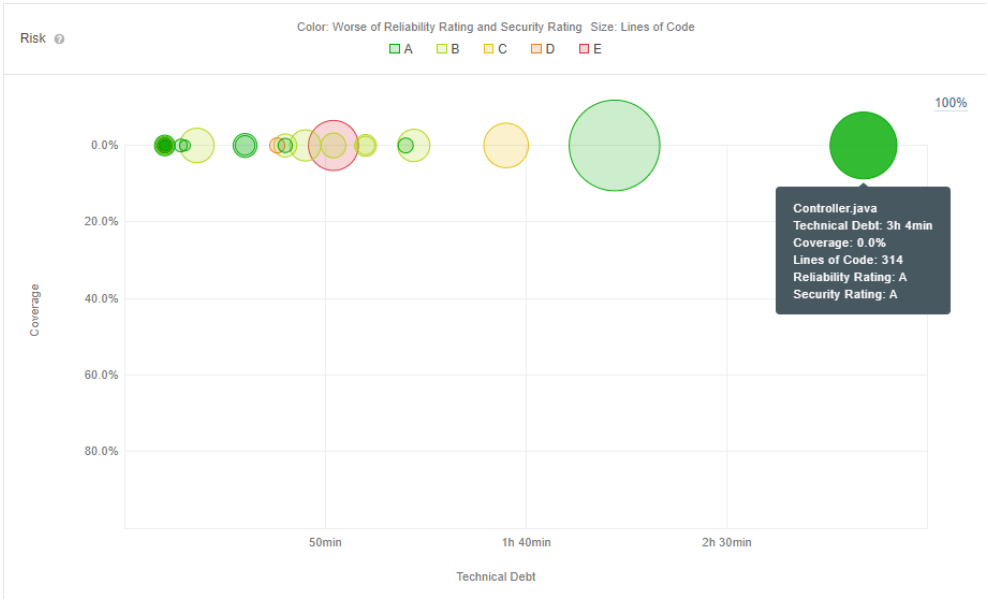


Figura 2.4: Technical debt do demo1.

2.4.2 demo2

O *SonarQube* estima portanto, que seria necessário, aproximadamente, seis dias de modo a poder corrigir e eliminar o *technical debt* associado aos code smells, ou seja, demoraria,

aproximadamente, seis dias a fazer *refactoring* do código de modo a eliminar todas as métricas indesejáveis. O mesmo poderá ser verificado no seguinte *overview* disponibilizado pela plataforma.

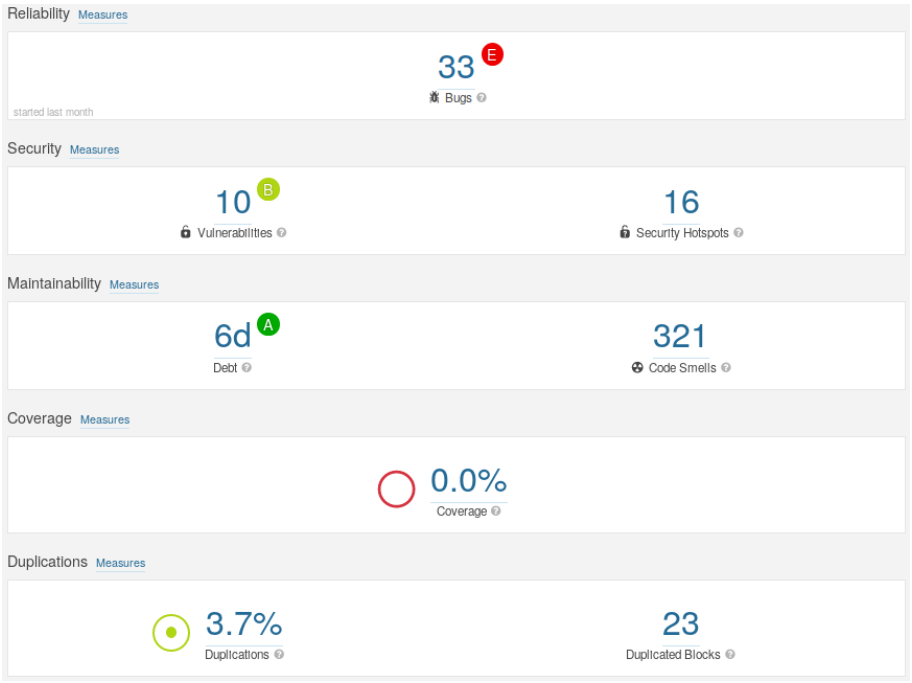


Figura 2.5: Overview do projeto demo2.

Também é possível verificar, na plataforma, quais as classes que irão demorar mais tempo a corrigir, sendo que, neste caso, a classe *UmCarroJaAPP* irá demorar bastante mais tempo a eliminar os *code smells*, *bugs* e vulnerabilidades relativamente a todas as outras. É estimado que se demore trinta e uma horas a eliminar o *technical debt* dessa classe.

Como tal, na seguinte figura é possível observar os *technical debts* associados a cada classe do projeto.

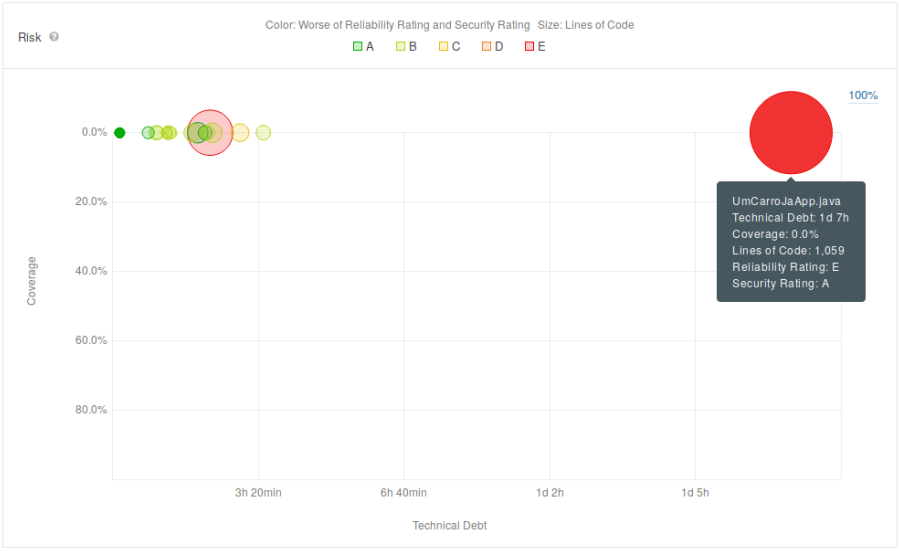


Figura 2.6: Technical debt do demo2.

2.5 CodeMR

Tal como foi abordado anteriormente, o *CodeMR* é uma ferramenta de análise da qualidade do código fonte, sendo que este define várias métricas, tais como, a complexidade do código, a coesão do código e entre outros. Estas métricas são, de seguida, avaliadas juntamente com o código fonte, criando de seguida um todo conjunto de ferramentas gráficas que ajudam à visualização destas métricas quando aplicadas ao código fonte produzido.

Estas ferramentas gráficas são extremamente úteis, principalmente no que toca a identificar numa maneira intuitiva e rápida que classes poderão ser melhoradas, e os principais pontos de falha da aplicação.

2.5.1 Métricas do CodeMr

O *CodeMR* também possui a funcionalidade de visualizar a relação entre as classes na forma de um grafo, onde cada classe é representada por uma forma geométrica, que indica o quão problemática a classe é para o funcionamento da aplicação, sendo que, são utilizadas as métricas da complexidade e do *coupling* de modo a avaliar cada classe. O seguinte esquema visa descrever as representações dos nodos de modo a, depois, apresentar o grafo deste projeto.

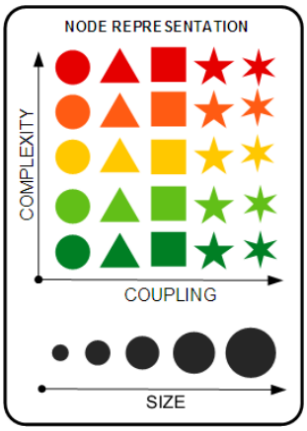


Figura 2.7: Representação dos nodos.

Na sua generalidade as cores apresentadas têm o seguinte significado:

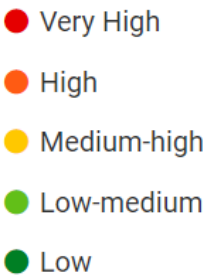


Figura 2.8: Código de cores.

Apresentam-se, agora, mais algumas das métricas utilizadas pelo *CodeMr* e que se encontram neste relatório.

NOM	**Number of Methods** Related Quality Attributes: Size The number of methods in a class.

Figura 2.9: Métrica NOM.

NOF	**Number of Fields** Related Quality Attributes: Size The number of fields (attributes) in a class

Figura 2.10: Métrica NOF.

WMC

Weighted Method Count

Related Quality Attributes: Complexity, Size

The weighted sum of all class' methods an represents the McCabe complexity of a class. It is equal to number of methods, if the complexity is taken as 1 for each method. The number of methods and complexity can be used to predict development, maintaining and testing effort estimation. In inheritance if base class has high number of method, it affects its' child classes and all methods are represented in sub-clasess. If number of methods is high, that class possibly domain spesific. Thefore they are less reusable. Also these classes tend to more change and defect prone.

Figura 2.11: Métrica WMC.

RFC

Response For a Class

Related Quality Attributes: Complexity

The number of the methods that can be potentially invoked in response to a public message received by an object of a particular class. It includes the full call graph of any method called from the originating method.If the number of methods that can be invoked at a class is high, then the class is considered more complex and can be highly coupled to other classes. Therefore more test and maintain effort is required.

Figura 2.12: Métrica RFC.

SRFC

Simple Response For a Class

Related Quality Attributes: Complexity

The number of the methods that can be potentially invoked in response to a public message received by an object of a particular class. It includes methods directly invoced from the class. If the number of methods that can be invoked at a class is high, then the class is considered more complex and can be highly coupled to other classes. Therefore more test and maintain effort is required.

Figura 2.13: Métrica SRFC.

DIT

Depth of Inheritance Tree

Related Quality Attributes: Complexity

The position of the class in the inheritance tree. Has 0 (zero) value for root and non-inherited classes.For the multiple inheritance, the metric shows the maximum length. Deeper class in the inheritance tree, probably inherit. Therefore, it is harder to predict its behavior. Also this class relatively complex to develop, test and maintain.

Figura 2.14: Métrica DIT.

CBO

Coupling Between Object Classes

Related Quality Attributes: Coupling

The number of classes that a class is coupled to. It is calculated by counting other classes whose attributes or methods are used by a class, plus those that use the attributes or methods of the given class. Inheritance relations are excluded. As a measure of coupling CBO metric is related with reusability and testability of the class. More coupling means that the code becomes more difficult to maintain because changes in other classes can also cause changes in that class. Therefore these classes are less reusable and need more testing effort.

Figura 2.15: Métrica CBO.

ATFD

Access to Foreign Data

Related Quality Attributes: Coupling

ATFD (Access to Foreign Data) is the number of classes whose attributes are directly or indirectly reachable from the investiggated class. Classes with a high ATFD value rely strongly on data of other classes and that can be the sign of the God Class.

Figura 2.16: Métrica ATFD.

SI

Specialization Index

Related Quality Attributes: Complexity

Defined as NORM * DIT / NOM. The Specialization Index metric measures the extent to which subclasses override their ancestors classes. This index is the ratio between the number of overrid- den methods and total number of methods in a Class, weighted by the depth of inheritance for this class. Lorenz and Kidd precise : Methods that invoke the superclass' method or override template are not included.

Figura 2.17: Métrica SI.

LCAM

Lack of Cohesion Among Methods(1-CAM)

Related Quality Attributes: Cohesion

CAM metric is the measure of cohesion based on parameter types of methods. LCAM = 1-CAM

Figura 2.18: Métrica LCAM.

NOC

Number of Children

Related Quality Attributes: Coupling

The number of direct subclasses of a class. The size of NOC approximately indicates how an application reuses itself. It is assumed that the more children a class has, the more responsibility there is on the maintainer of the class not to break the children's behaviour. As a result, it is harder to modify the class and requires more testing.

Figura 2.19: Métrica NOC.

LTCC

Lack of Tight Class Cohesion

Related Quality Attributes: Cohesion

The Lack of Tight Class Cohesion metric measures the lack cohesion between the public methods of a class. That is the relative number of directly connected public methods in the class. Classes having a high lack of cohesion indicate errors in the design.

Figura 2.20: Métrica LTCC.

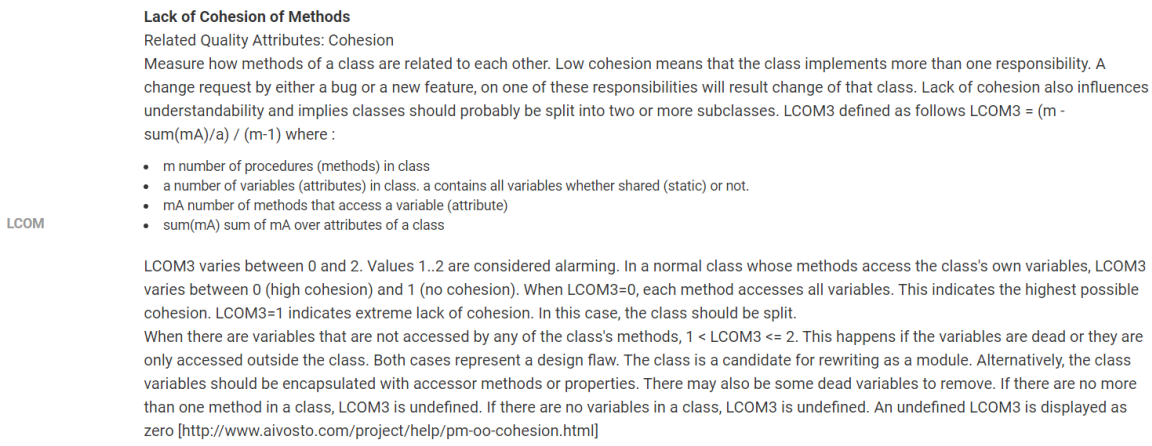


Figura 2.21: Métrica LCOM.

2.5.2 demo1

Utilizando as ferramentas de análise do *CodeMr* podemos extrair o gráfico que se encontra a seguir. É de notar que nenhum dos diagnósticos alcançou a meta de *Very high*. Verifica-se que 16% da complexidade se encontra em duas classes, o que poderá não ser muito mau, uma vez que o projecto contém 43 classes. O acoplamento alto também poderá indicar que as "responsabilidades"de cada classe não estão bem definidas.

Figura 2.22: Análise do CodeMr no projecto demo1.

O *CodeMr* também produziu o seguinte estudo que indica que existe apenas uma classe com elevada complexidade e acoplamento. Após uma averiguação, usando o código, foi possível

aferir que a classe possui 321 linhas de código, sendo que cerca de 300 destas constituem apenas um método.

Classes with high coupling, high complexity (#1)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	WMC	RFC	NOM
1	Controller					321	28	62	194	2

Figura 2.23: Classes mais problemáticas do demo1.

O diagnóstico adiante vem a reafirmar o que tem ser dito acerca da classe *Controller*.

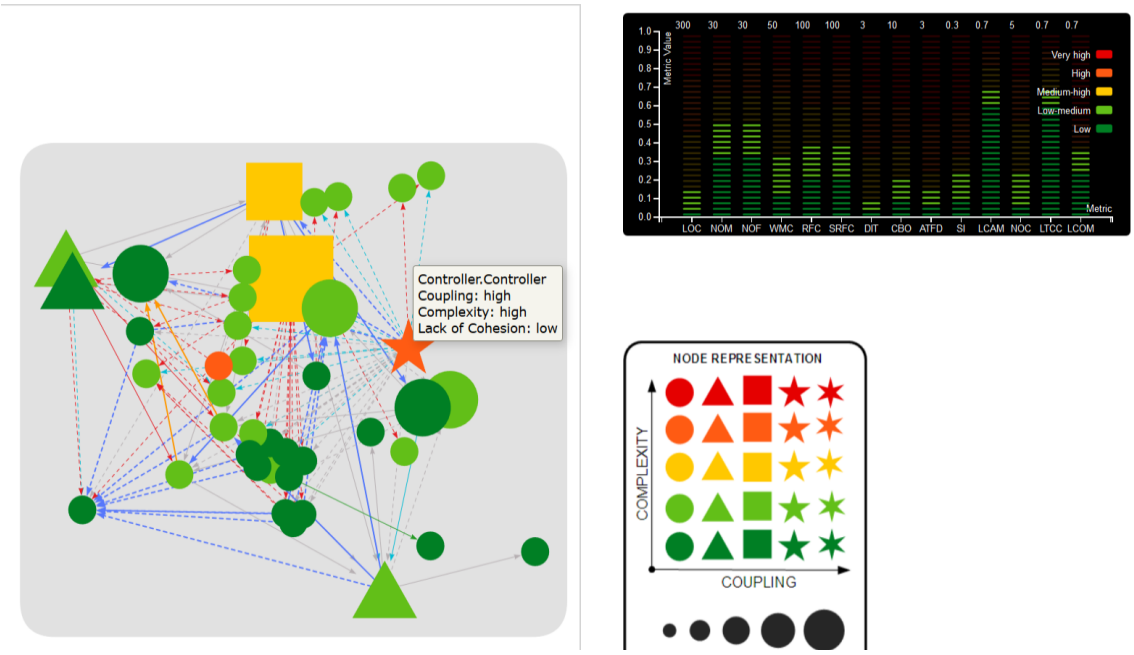


Figura 2.24: Grafo do projeto demo1.

2.5.3 demo2

No projeto demo2, e após correr a análise do código do projeto demo2, foi possível obter as seguintes distribuições de métricas, onde tal como será possível observar na seguinte figura, o projeto possui um elevado grau de complexidade, sendo que também exibe uma alarmante falta de coesão. O *coupling* também está elevado, pelo que é um mau sinal, uma vez que geralmente um *coupling* baixo está associado a um projeto bem estruturado e bem definido. Relativamente ao tamanho das classes, também é uma métrica importante uma vez que ao avaliar o número de linhas de código, o projeto relativamente a esta métrica está dentro do esperado tendo em conta o grau de complexidade do enunciado.

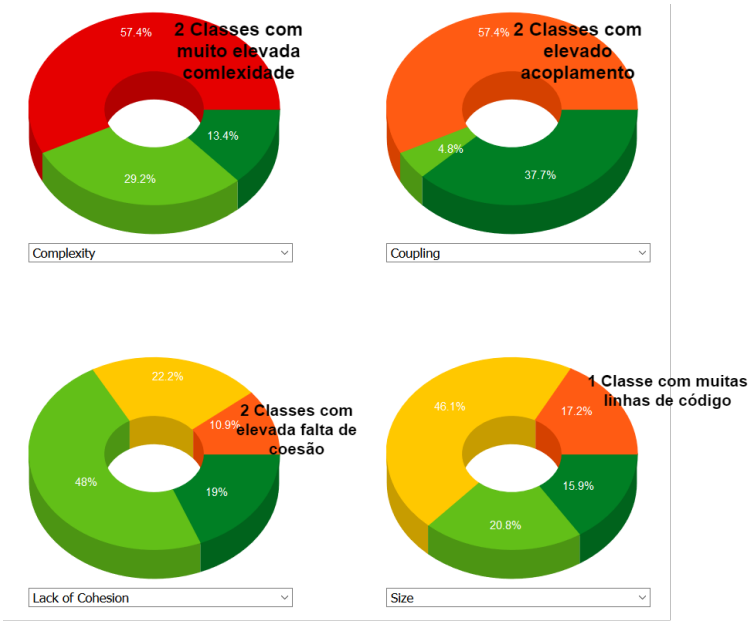


Figura 2.25: Análise do CodeMR no projeto demo2.

A análise do *CodeMR* também produziu os seguintes resultados, onde identificou duas classes que possuem duplamente alta complexidade e alto *coupling* o que indica portanto, uma falta de estrutura e de planeamento durante o decorrer do desenvolvimento do código.

Classes with high coupling, high complexity (#2)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	WMC	RFC	NOM
1	UmCarroJaApp					884	25	**195**	**243**	1
2	UmCarroJa					378	21	**144**	151	50

Figura 2.26: Classes mais problemáticas no demo2.

Onde, tal como é possível observar, na classe *UmCarroJA* para além de possuir um coupling elevado, juntamente com a complexidade, possui também uma falta de coesão considerável.

Portanto, tendo em conta a explicação da representação dos nodos, foi gerado o seguinte grafo representativo da avaliação das métricas abordadas anteriormente, juntamente com o projeto demo2.

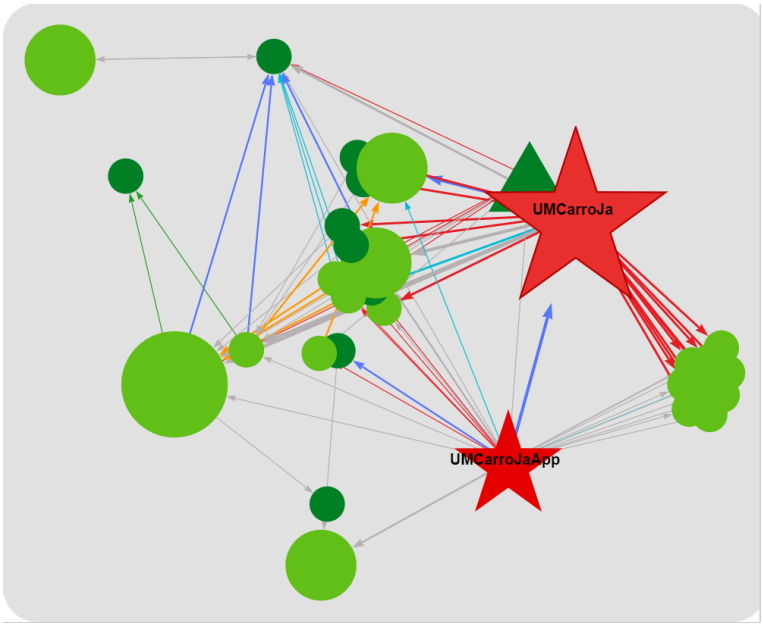
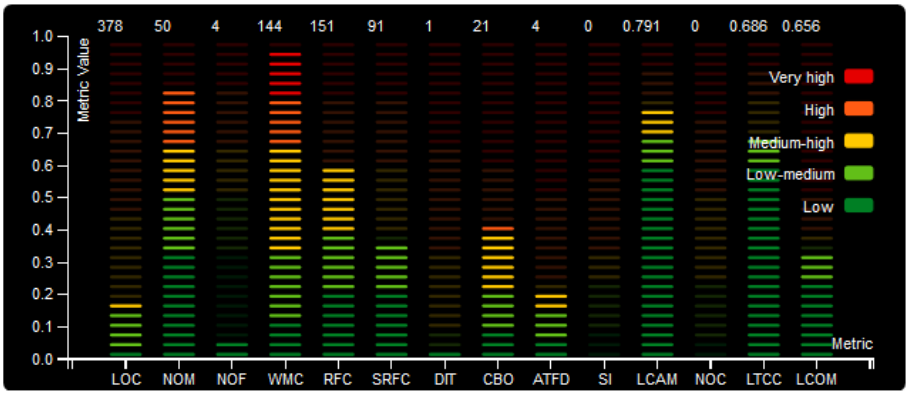


Figura 2.27: Grafo do projeto demo2.

O grafo anterior explicita as relações entre as diferentes classes, sendo que, é possível através da interface disponibilizada pelo *CodeMR* inspecionar a qualidade de cada uma das diferentes classes. Neste caso, foi inspecionada a classe que corresponde à estrela maior, ou seja, a classe que possui maior complexidade juntamente com maior *coupling*, sendo essa classe a *UmCarroJa*, de onde foi possível obter também o seguinte gráfico de barras, onde são aplicadas e de seguida apresentadas as diferentes métricas de análise da qualidade do software relativas a esta classe, tal como o número de linhas de código (LOC), o número de métodos (NOM), o número de campos (NOF), entre outros.



UmCarroJa
Coupling: high
Complexity: very-high
Lack of Cohesion: medium-high

Figura 2.28: Métricas associadas à classe UmCarroJa.

Em cima de cada métrica, é também possível ver o resultado da aplicação dessa métrica na classe em questão, sendo que a classe *UmCarroJa* possui um número elevado de métodos,

um número também elevado de *Weighted Method Count (WMC)*, que corresponde à soma das complexidades dos métodos da classe em questão, sendo que, como o número é elevado, mais difícil será a manutenção e o desenvolvimento da classe.

3 Tarefa 2: Refactoring

3.1 Eliminação de bugs

Utilizando o *SonarQube* foi possível identificar os *bugs* existentes no projeto e dessa maneira foi também possível eliminar os diversos *bugs* de tipos diferentes.

3.1.1 Blocker

Os *blocker bugs* existentes no projeto *demo1* eram referentes a não fechar o *object output* com o *try resource*, como tal foi necessário mudar as funções identificadas pelo *SonarQube* com este bug, ou seja, a função *read* e a função *save*.

A função apresentada de seguida sofreu o *Refactoring* de modo a corrigir o *blocker bug*, tendo sido utilizada a funcionalidade de *Rule* aquando da apresentação do *bug* no *SonarQube* de modo a guiar o *refactoring*.

```
public void save(String fName) throws IOException {
    try (FileOutputStream a = new FileOutputStream(fName);
         ObjectOutputStream r = new ObjectOutputStream(a)) {
             r.writeObject(this);
             r.flush();
             r.close();
    } catch (Exception e) {
             System.out.println(e.getMessage());
         }
   }
```

O mesmo princípio foi aplicado com a função *read* de modo a eliminar desta maneira os dois *blocker bugs* existentes no projeto.

3.1.2 Critical

Relativamente aos *critical bugs* no projeto, estes surgiram devido a não ser guardada uma instância do *Random* na classe em si, sendo chamada constantemente a classe *Random* dentro do corpo da função.

O *SonarQube* portanto identificou duas classes onde este bug crítico existia, ana classe *Weather* e na classe *Traffic*, tendo-se feito o *refactoring*, por exemplo, na classe *Weather*, da seguinte maneira:

```
private Random rand = SecureRandom.getInstanceStrong();
Weather() throws NoSuchAlgorithmException { }
...
```

```
public double getSeasonDelay() {
    Double a = this.rand.nextDouble();
    switch (getSeason()){
        case "Summer":
            return a % 0.1;

        case "Spring":
            return a % 0.3;

        case "Fall":
            return a % 0.35;

        default:
            return a % 0.6;
        }
}
```

De realçar também, que foi utilizado o *SecureRandom* ao invés de utilizar só o *Random* uma vez que este foi o indicado na *rule* disponibilizada pelo *SonarQube*.

O mesmo princípio anterior foi também aplicado à respetiva função na classe *Traffic*.

3.1.3 Major

O *SonarQube* identificou apenas um *major bug* no projeto *demo1*, sendo que este encontra-se na classe *Car* sendo que o *bug* é relativo ao nome do método ser *equals* uma vez que dá *overlap* com o método *equals* já definido pelo java. A solução sugerida pela *rule* do *SonarQube* foi ou mudar o nome do método ou colocar *"@Override"* antes do método. O grupo optou por utilizar a ferramenta de *refactoring* de *rename* disponibilizada pelo *IntelliJ* para mudar o nome do método não só na própria classe mas também em outras classes que utilizam esse método.

```
public boolean equalsCarType(CarType a) {
    return a == this || a == any;
}
```

3.1.4 Minor

O *SonarQube* identificou nove *minor bugs*, sendo os *bugs* identificados possuem o mesmo cerne, estes surgem devido ao facto de ao fazer *"@Override"* do método *equals* também se deve fazer *"@Override"* do método *hashCode*, portanto nas nove classes identificadas pelo *SonarQube* foi adicionada a seguinte função.

```
@Override
public int hashCode() {
    return super.hashCode();
}
```

3.1.5 Verificação dos resultados no *SonarQube*

Após terem sido removidos os *bugs* através do *refactoring* no código, de seguida foi utilizado novamente o *SonarQube* de modo a averiguar os resultados finais do *refactoring*.

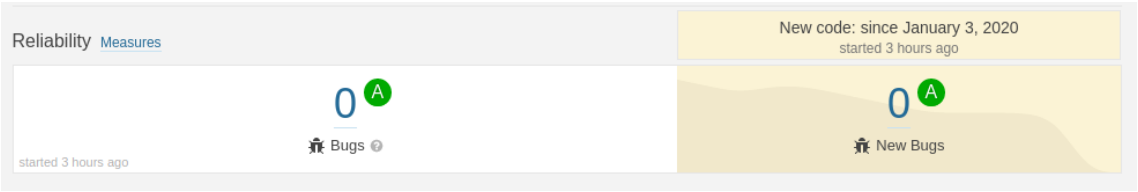


Figura 3.1: Verificação do número de *bugs* após o *refactoring*.

3.2 Eliminação das vulnerabilidades

Tal como na secção anterior, também foi utilizado o *SonarQube* de modo a auxiliar no processo de *refactoring* de modo a eliminar as vulnerabilidades no código.

3.2.1 Minor

No projeto, o *SonarQube* identificou apenas uma vulnerabilidade, sendo ela uma *minor vulnerability*. A vulnerabilidade que identificou devia-se ao facto de utilizar o *printStackTrace* ao invés de utilizar um *logger*, uma vez que os *logs* garantem muita mais flexibilidade.

Como tal, na classe identificada pelo *sonar*, na classe *Parser* foi necessário alterar o *catch* do *IOException* para o seguinte:

```
} catch (IOException e) {
    LOGGER.warning(e.toString());
}
```

Para além disse, foi necessário também definir a variável *LOGGER* da seguinte maneira:

```
private static final Logger LOGGER = Logger.getLogger(Parser.
   class.getName());
```

3.2.2 Verificação dos resultados no *SonarQube*

Após ter sido removida a única vulnerabilidade do projeto, foi possível verificar e comprovar os resultados do *refactoring* através da interface gráfica disponibilizada pelo *SonarQube*.

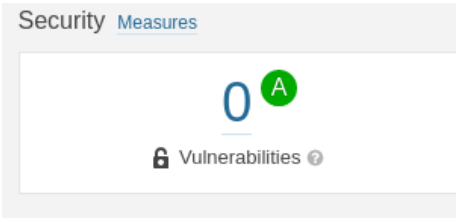


Figura 3.2: Verificação do número de vulnerabilidades após o *refactoring*.

3.3 Eliminação dos *code smells*

Tal como nos outros casos, utilizou-se novamente o *sonarqube* para identificar os *smells* bem como a sua ferramenta *built in* de *rules* de modo a corrigir os *smells*.

3.4 Críticos

Muitos *smells* críticos que o *sonarqube* identificou eram relacionados com o uso excessivo de uma *String*, sendo esta repetida várias vezes, quando por sua vez poderia ter sido definida uma constante.

Como tal, e pegando no exemplo da classe *Menu* é repetida nessa mesma classe a *String "Tipo do Carro: [electric, gas, hybrid or any]"* cerca de três vezes, tendo sido definida uma contante no início da classe para evitar essa repetição

```
private static final String REPEATEDSTRINGTWO = "Tipo do Carro
    : [electric, gas, hybrid or any]";
```

Este mesmo *refactoring* foi bastante comum uma vez que era cometido este smell em muitas outras classes.

Outro tipo de *critical smell* é relacionado com as expressões dos nomes dos parâmetros não respeitarem a expressão regular, como tal, foi também necessário fazer o *rename* de certos parâmetros de acordo a regra de *renaming* apresentada.

Um exemplo dos parâmetros não obedecerem de acordo com a expressão regular ocorre na classe *Menu* onde foi necessário colocar todos os parâmetros do *enum* em *caps lock*.

```
public enum MenuInd {
    INITIAL,
    LOGIN,
    REGISTER,
    REGISTERCLIENT,
    REGISTEROWNER,
    ...
}
```

Também foi necessário corrigir os *switches* uma vez que estes muitas vezes não possuíam um *default case*.

Por exemplo, na classe *Car* na função *fromString* não existia no *switch* um *default case*, como tal foi necessário adiciona-lo.

```
 switch (s) {
    case "Electrico":
        return CarType.ELECTRIC;
    case "Gasolina":
        return CarType.GAS;
    case "Hibrido":
        return CarType.HYBRID;
    case "Todos":
        return CarType.ANY;
    default:
        throw new UnknownCarTypeException();
}
```

De certo modo estes três tipos de *critical smells* englobam a generalidade dos casos apresentados pelo *sonarqube*, e como tal foi necessário aplicar os mesmos métodos que os apresentados anteriormente de modo a eliminar os outros *critical smells*.

3.4.1 Major

Relativamente aos *major smells*, estes em número são menores que os *critical smells*, e um tipo de *major smell* que foi muito comum aparecer no projeto era relacionado com o facto de em muitas situações de *catch* de uma *exception* não ser efetuado nenhum tratamento dessa mesma *exception*.

Tomando o exemplo da classe *Controller*, na função *run*, no case correspondente ao *Closest* a exceção não é tratada no *catch* do *UnknownCompareTypeException* como tal foi necessário utilizar o *LOGGER* de modo a reportar esta *exception*.

```
catch (UnknownCompareTypeException ignored) {LOGGER.warning(
   ignored.toString());}
```

Foi também necessário substituir algumas utilizações do *system.out* com um *logger* uma vez que este é mais aconselhável.

Na classe *Parser* foi necessário extrair um pedaço de código na função *parseLine* para um método à parte, uma vez que torna-se mais difícil a compreensão do código ao ocorrer um *nesting* de um *try/catch block* dentro de um *case*.

```
case "Aluguer":
    if (content.length != 5)
        break;
    AlugerRental(model,content);
    break;
```

O método *AlugerRental* possui exatamente o mesmo código que estava no bloco *try/catch*.

```
private void AlugerRental(UMCarroJa model, String[] content){
    try {
        model.rental(new StringBuilder()
                .append(content[0])
                .append("@gmail.com")
                .toString(),
                (...)
```

Um outro tipo de *major smell* que foi eliminado correspondeu a eliminar métodos que foram declarados mas que não eram utilizados em mais nenhum outro lado no código. O mesmo acontecia, por exemplo, na classe *Point*, onde foram eliminados os métodos *getX* e *getY* que não eram utilizados.

Os tipos de smells referidos anteriormente repetiam-se bastantes vezes no código, portanto foi necessário implementar a metodologia apresentada anteriormente para os eliminar.

3.4.2 Minor

A maior parte dos *smells* identificados pelo *sonarqube* tratavam-se de *minor smells*.

Um tipo muito comum deste tipo de *smell* tinha a ver com os nomes dos *packages* uma vez que estes não obedeciam à expressão regular na medida em que o nome continha uma letra maiúscula. Portanto, foi elaborado um *rename* de modo a alterar o nome dos *packages* para estes possuírem apenas letras minúsculas.

Um outro *smell* devia-se com o facto de não ser utilizada a função *isEmpty()* para verificar se uma dada coleção está vazia ou não. Um exemplo de uma ocorrência deste *smell* ocorre na classe *Controller* e foi necessário mudar a condição do *if* no *case ReviewRental* para o seguinte.

```
if (lR.isEmpty()){
    this.menu.back();
    break;
}
```

A classe *Car*, por exemplo, também possui um *smell* relativamente à maneira de como na função *hasRange* é utilizada a negação ao invés de ser utilizado o operador <= no *return*.

```
 boolean hasRange(Point dest) {
    if(this.range / this.getFullTankRange() < 0.1) return
       false;
    return (this.position.distanceBetweenPoints(dest) * 1.2 <=
        this.range);
}
```

Um outro *smell* que também foi facilmente corrigido era relativamente à classe *Rentals* a variável privada *id* não estava de acordo com a *Java Specification Language*, portanto foi necessário reordenar os argumentos de modo a corresponder com a especificação.

```
private static int id;
```

Mais uma vez, os *smells* anteriores englobam a generalidade dos *minor smells* encontrados pelo *sonarqube*. Existiu contudo outros *smells*, como é óbvio, como por exemplo passar a classe *Main* para um *package* próprio ou remover blocos duplicados, cujo *refactoring* foi também assegurado.

3.4.3 Verificação dos resultados no *SonarQube*

Após ter sido aplicado todos os *refactorings* necessários para corrigir ao máximo os 130 *smells* identificados pelo mesmo, foi possível diminuir de 130 para 22 *smells*, o que implicou uma diminuição de cerca de 83% dos *smells* existentes no código. Deixam-se apenas *smells* cuja correcção implicaria uma alteração profundas no código provido.

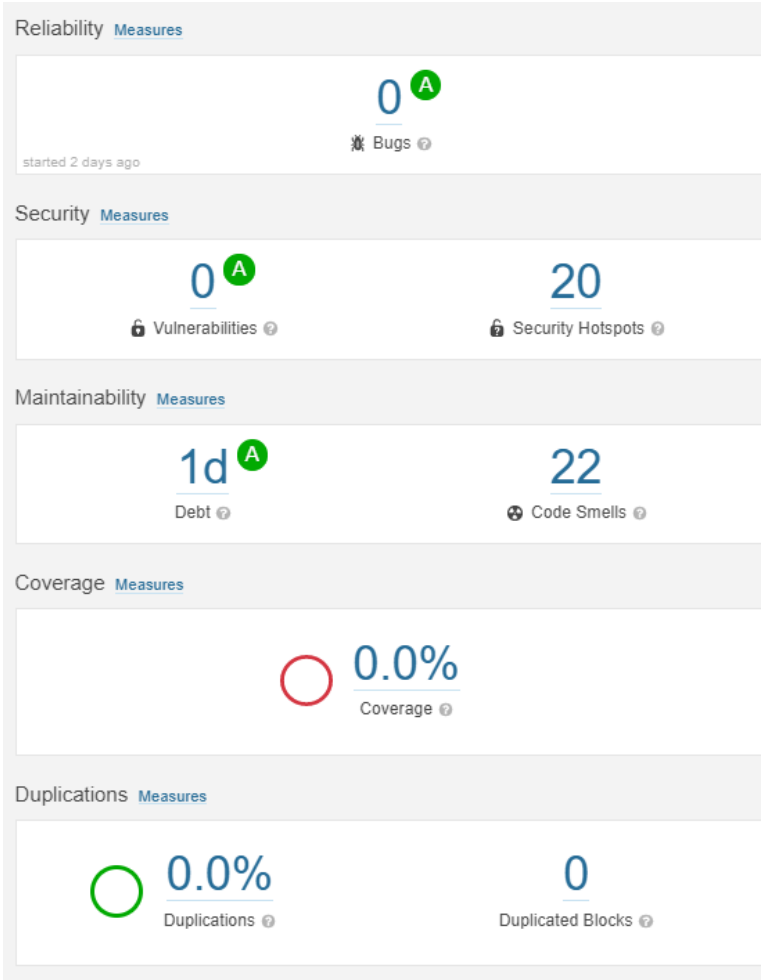


Figura 3.3: Análise do *SonarQube* após o refactoring.

3.5 Eliminação dos *Energy Smells*

Foram corrigidos todos os seis *smells* energéticos apontados pelo *jStanley*.

Contudo, no primeiro *smell* deste tipo, em que foi aconselhada a substituição de `ArrayList` por `AttributeList` ou `CopyOnWriteArrayList`, não foi feita a melhor opção, como se pode ver em ***GreenSoftware-Analysis*** (slide 41, operação "get"), devido a incompatibilidade e optou-se por `CopyOnWriteArrayList`.

Assim deu-se por concluído o *refactor energético*.

4 Tarefa 3: Testes

Uma vez que através da análise anterior dos *smells* e de outros tipos de métricas, tal como a complexidade, foi possível inferir que o projeto *demo1* encontra-se num melhor estado que o *demo2*, portanto o grupo decidiu elaborar os testes no projeto *demo1*.

4.1 Teste unitários em *JUnit*

Relativamente aos testes unitários, estes foram focados apenas em métodos menos cobertos pelos testes automáticos gerados pelo *Evosuite*. Contudo, e como vai ser possível verificar posteriormente, como estes já eram bastantes extensos, foram apenas gerados dois testes unitários em *JUnit*.

O primeiro teste unitário efetuado foi para testar a função *searchCar* da classe *Cars*, este método irá retornar um objeto da instância *Car* dado a matrícula do próprio carro. Como tal foi necessário declarar os objetos necessários para declarar o objeto *Car* com o intuito de depois utilizar no teste da função anterior. De seguida é adicionado o *Car* à *HashMap* de carros, e de seguida é utilizada a função *assertEquals* para comprovar se de facto consegue retirar o carro correspondente à incialização do objeto anterior.

```
@Test
public void test22() throws CarExistsException,
   InvalidCarException {
      // arrange
      Cars newCar = new Cars();
      Owner owner0 = new Owner("$t+Lq(8", "$t+Lq(8", (String)
         null, 0, "$t+Lq(8");
      Car.CarType car_CarType0 = Car.CarType.GAS;
      Double double0 = new Double((-1668.5140747657));
      Point point0 = new Point(double0, double0);
      Car car0 = new Car("AB-88-22", owner0, car_CarType0, 0,
         0, 0, 7, point0, (String) null);
      // act
      newCar.addCar(car0);
      // assert
      assertEquals(car0,newCar.searchCar("AB-88-22"));
}
```

O próximo teste unitário efetuado foi no método *getTotalBilledCar* da classe *Rentals* onde foi necessário, tal como no caso anterior foi necessário instanciar os objetos necessários para declarar os objetos relativos ao *Car*, é também instanciada um objeto da classe *Rentals*. De seguida, é testado com o *assertEquals* se a chamada do método *getTotalBilledCar* corresponde

a zero que é o valor real e o esperado que se obtenha.

```
@Test
public void test23()  throws Throwable  {
    // arrange
    Rentals newRental = new Rentals();
    Double double0 = 100.2345;
    Double double1 = 123.2356;
    Point point0 = new Point(double0, double1);
    Owner owner0 = new Owner("grupo2@gmail.com", "grupo2", "
       Braga", 80982213, "passwd");
    Car.CarType car_CarType0 = Car.CarType.ELECTRIC;
    Car car0 = new Car("AB-22-22", owner0, car_CarType0, 90,
       20000, 10, 500, point0, "Tesla");
    // act
    double doubleRentals = newRental.getTotalBilledCar(car0);
    // assert
    assertEquals(0.0, doubleRentals, 0.01);
}
```

Em ambos os casos, os testes unitários foram concluídos com sucesso tal como seria de esperar.

4.2 Testes automáticos no *Evosuite* antes do *refactoring*

Através do *Evosuite* é possível a geração de testes unitários automaticamente, como tal, foram gerados um conjunto de testes, no total 292 testes, de modo a testar extensivamente todas as funcionalidades e classes principais do projeto antes de efetuar o *refactoring* do código.

Como tal, de seguida apresenta-se uma tabela onde se apresenta o número de testes efetuados e o número de testes que efetivamente passaram em cada uma das classes testadas do projeto.

Classe testada	Número de testes efetuados	Número de testes que passou
Car	63	63
Cars	21	21
Client	17	17
Owner	18	18
Rental	49	49
Rentals	22	22
Traffic	3	3
UmCarroJa	65	65
User	17	17
Users	12	12
Weather	4	4
Controller	1	1
Total	292	292

Como é possível verificar, todos os testes unitários gerados pelo *Evosuite* foram concluídos com sucesso.

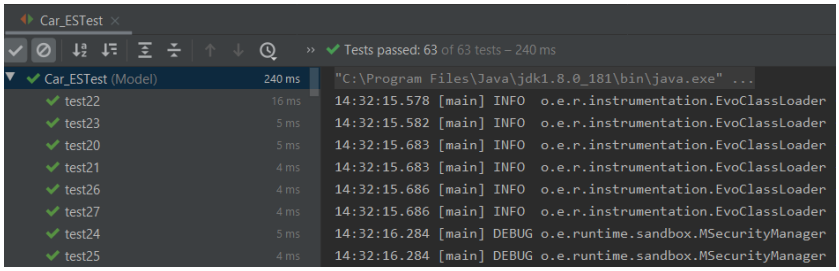


Figura 4.1: Testes gerados concluídos com sucesso na classe carro.

Para além disso, é também possível verificar no projeto as seguintes classes de teste geradas automaticamente pelo *EvoSuite*, bem como um exemplo de um teste gerado automaticamente.

É de realçar que o teste seguinte possui o intuito de testar a função *getPasswd* e a função *getRates* da classe *User*.

```
@Test(timeout = 4000)
 public void test03()  throws Throwable  {
     Owner owner0 = new Owner("", "", "", (-4738), "rIpTm5QJ"
        );
     String string0 = owner0.getPasswd();
     assertEquals("rIpTm5QJ", string0);
     assertEquals(100, owner0.getRates());
 }
```

Figura 4.2: Classes de teste geradas pelo *Evosuite*.

4.3 Testes automáticos no *Evosuite* depois do *refactoring*

Depois de fazer o *refactoring* do código com o intuito de eliminar os *smells* foi repetida a geração dos testes automáticos com o intuito de testar se mesmo após o *refactoring* extensivo do código, se mesmo assim as funcionalidades principais da aplicação continuam operacionais.

Foram apenas gerados alguns testes automáticos nas classes principais da aplicação, pelo que foram por consequência geradas as seguintes classes de teste.

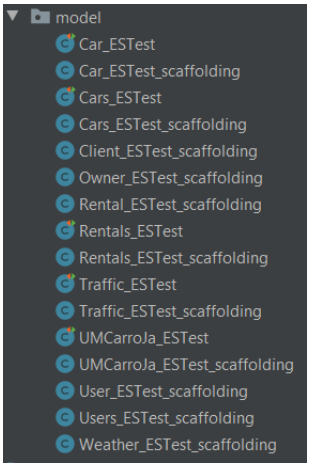


Figura 4.3: Classes de teste geradas pelo *Evosuite* no projeto com *refactoring*.

Portanto e como é possível observar, foi apenas gerados os testes automáticos relativos às classes *Car*, *Cars*, *Rentals*, *Traffic* e *UMCarroJa* uma vez que estas foram as classes identificadas pelo *SonarQube* que possuíam mais *smells* e como tal, são as classes que necessitam mais de ser testadas de modo a averiguar que o *refactoring* do código manteve o mesmo código funcional.

Como tal, e por exemplo ao fazer *run* dos testes unitários na classe de teste *UMCarroJa* que é a que possui um maior número de dependências inter-classe, é possível verificar que os 67 testes unitários gerados automaticamente são concluídos com sucesso.

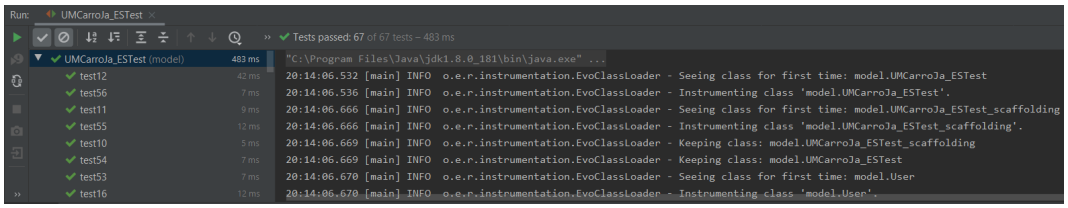


Figura 4.4: Testes concluídos com sucesso na classe *UMCarroJa*.

Para as restantes classes, os testes também foram executados pelo que foi também possível observar que também foram concluídos com sucesso, pelo que é possível inferir que mesmo após o *refactoring* do código para eliminar os *bugs*, vulnerabilidades e *code smells* o mesmo código continua funcional.

4.4 Sistema *JaCoCo*

Nesta secção, averiguou-se qual a cobertura dos testes gerados pelo *evosuite* numa fase anterior. Para tal, utilizou-se o *plugin* do *JaCoCo* disponibilizado pelo *InteliJ IDE*.

Como forma de avaliar a cobertura do código, foi necessário gerar através do *Evosuite* testes para todas as classes principais do projeto. De seguida é apresentada uma figura onde depois de correr cada uma das classes de teste geradas automaticamente apresenta-se à cobertura total do *package* das classes principais do projeto ou seja o *package model*.

Convém também referir que a cobertura foi analisada para o projeto depois de ter sido feito o *refactoring* do código.

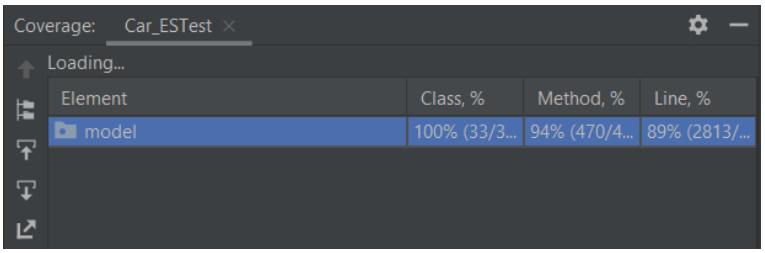


Figura 4.5: Cobertura calculada pelo *JaCoCo* no *package model*.

Para além disso, é também possível observar a cobertura individual de cada classe como se poderá verificar na seguinte figura.

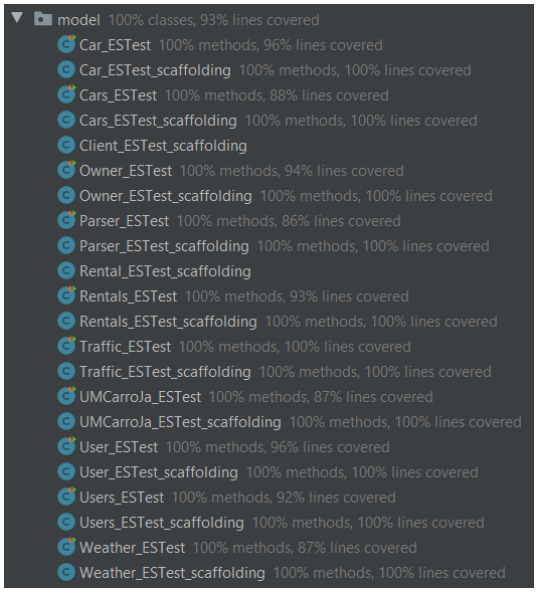


Figura 4.6: Cobertura individual calculada pelo *JaCoCo* para as classes principais do projeto.

Como é possível observar pelas duas figuras supracitadas, a cobertura do código automático gerado pelo *Evosuite* é quase total no que toca à cobertura dos métodos e das linhas de código e possui cobertura total no que toca às classes.

4.5 Geração Automática de Ficheiros de Logs

A geração automática de ficheiros (realizada em Haskell) para posterior carregamento de dados para a aplicação possui as seguintes qualidades:

- Não reescreve ficheiros gerados anteriormente;
- Gera coordenadas localizadas no interior de Portugal Continental;
- Gera classificações com matrículas e Nifs;
- Cerca de 2400 nomes para geração;
- Cerca de 21300 localidades para geração;
- Contém 73 marcas de automóveis com probilidades reais;
- Tipo de combustível de um automóvel gerado segundo probabilidades reais.

A geração é feita dando em primeiro lugar as quantidades desejadas. Na figura abaixo é possível ver um exemplo da chamada da função *"main"*, que é responsável pela execução geral do programa.

```
*Main> main
Numero de Proprietarios: 10
Numero de Clientes: 10
Numero de Carros: 100
Numero de Alugueres: 10
Numero de Classificacoes: 10
```

Figura 4.7: Chamada da função main do gerador de logs.

O gerador de logs criado também permite que nunca sejam perdidos ficheiros de log devido a reescritas de ficheiros. Criando sempre um ficheiro novo como se pode ver na figura adiante.

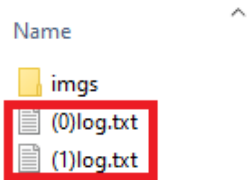


Figura 4.8: Exemplo da não reescrita de ficheiros.

Como resultado dessa geração pode-se ver, a título de exemplo, a seguinte imagem.

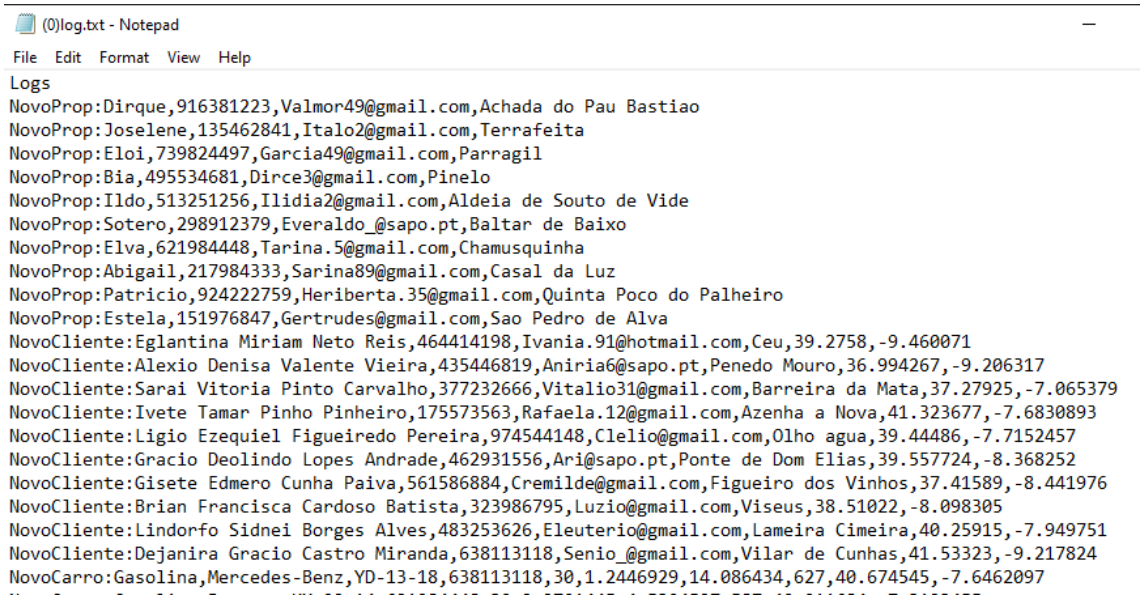


Figura 4.9: Exemplo do conteúdo dum ficheiro de logs.

5 Tarefa 4: Análise de Desempenho

Para realizar esta tarefa o grupo de trabalho decidiu empregar o **demo1** por ser a aplicação com menos *bugs* e *smells* e, consequentemente, conseguir-se extrair resultados e conclusões com maior celeridade.

De modo a realizar a análise de desempenho foram gerados ficheiros de carregamento para a aplicação, usando o gerador definido na tarefa 3. Contudo, o gerador implementado mostrou gerar resultados mais diversificados do que a aplicação conseguiria suportar. O **demo1** demonstra uma grande dependência do ficheiro de logs fornecido pelo docente, pois o código implementado não tem apenas em conta os campos que fazem parte de um ficheiro de dados (o que seria de esperar e está correto, exemplo: "NovoProp: nome, nif, email,morada") como também está dependente de como são formados os dados concretos. Essas dependências consistem em usar email como identificador único de um *user* e supor que estes serão sempre «nif Do User>@gmail.com". Por exemplo, tendo em conta que as informações de um carro apenas contêm o nif do proprietário, isto resulta aquando da criação, num acesso aos utilizadores usando `nif + "@gmail.com"`, o que resulta em elevada dependência da forma como os dados são criados. É de frisar que este tipo de acesso acontece em várias situações.

Desta forma, o gerador de ficheiros, forçosamente, foi alterado para uma versão considerada inferior por ser menos rica a nível de dados.

Para o efeito criaram-se três ficheiros de carregamento, designados por "pequeno", "médio"e "grande", com número de dados crescente.

Teve-se como ponto de partida o ficheiro concedido pelos docentes, que funciona como modelo a nível de número de campos para o ficheiro com tamanho inferior. A partir daí, geraram-se todos os outros, multiplicando os campos sempre por um factor de 5.

	Pequeno	Médio	Grande
Número de Proprietários	300	1500	7500
Número de Clientes	300	1500	7500
Número de Carros	2000	10000	50000
Número de Alugueres	500	2500	12500
Número de Classificações	200	1000	5000

5.1 Análise anterior ao Refactoring

Com o auxílio da ferramenta RAPL efectuamos a análise de energia consumida pela *RAM*, pelo *CPU* e por *package* de algumas das acções com execução que nós consideramos mais custosa.

Efectuamos, também, a análise do tempo despendido por cada das acções, com a ajuda do *Intellij*.

	Ficheiro Pequeno	Ficheiro Médio	Ficheiro Grande
Login	Tempo: 0 ms Energia RAM: 7.3300E-4 Energia CPU: 9.1600E-4 Energia Package: 0.0040	Tempo: 1 ms Energia RAM: 0.0000 Energia CPU: 9.7600E-4 Energia Package: 0.0000	Tempo: 0 ms Energia RAM: 5.5000E-4 Energia CPU: 7.9400E-4 Energia Package: 0.0038
RegisterClient	Tempo: 0 ms Energia RAM: 0.0000 Energia CPU: 9.7600E-4 Energia Package: 0.0037	Tempo: 0 ms Energia RAM: 5.4900E-4 Energia CPU: 7.3300E-4 Energia Package: 0.0029	Tempo: 0 ms Energia RAM: 4.2700E-4 Energia CPU: 4.2700E-4 Energia Package: 0.0039
RegisterOwner	Tempo: 0 ms Energia RAM: 0.0000 Energia CPU: 4.2800E-4 Energia Package: 0.0000	Tempo: 0 ms Energia RAM: 0.0000 Energia CPU: 5.4900E-4 Energia Package: 0.0000	Tempo: 0 ms Energia RAM:7.3200E-4 Energia CPU:6.1100E-4 Energia Package:0.0042
Closest	Tempo: 6 ms Energia RAM: 0.0017 Energia CPU: 0.0032 Energia Package: 0.0119	Tempo: 7 ms Energia RAM: 0.0139 Energia CPU: 0.0199 Energia Package: 0.0717	Tempo: 40 ms Energia RAM: 0.0487 Energia CPU: 0.3548 Energia Package: 0.4702
Cheapest	Tempo: 8 ms Energia RAM: 0.0046 Energia CPU: 0.0063 Energia Package: 0.0202	Tempo: 9 ms Energia RAM: 0.0134 Energia CPU: 0.0240 Energia Package: 0.0799	Tempo: 42 ms Energia RAM: 0.0516 Energia CPU: 0.4055 Energia Package: 0.5337
CheapestNear	Tempo: 9 ms Energia RAM: 0.0175 Energia CPU: 0.0363 Energia Package: 0.1083	Tempo: 10 ms Energia RAM: 0.0495 Energia CPU: 0.1312 Energia Package: 0.3534	Tempo: 52 ms Energia RAM: 0.0741 Energia CPU :1.1357 Energia Package: 1.3460
Autonomy	Tempo: 21 ms Energia RAM: 0.0109 Energia CPU:0.0255 Energia Package:0.0683	Tempo: 26 ms Energia RAM: 0.0272 Energia CPU: 0.0522 Energia Package:0.1481	Tempo: 65 ms Energia RAM: 0.0612 Energia CPU: 0.5425 Energia Package: 0.6937
Specific	Tempo: 0 ms Energia RAM: 0.0000 Energia CPU: 3.6600E-4 Energia Package: 0.0000	Tempo: 0 ms Energia RAM: 0.0000 Energia CPU: 5.4600E-4 Energia Package: 0.0000	Tempo: 0 ms Energia RAM: 0.0000 Energia CPU: 4.2700E-4 Energia Package: 0.0000
AddCar	Tempo: 0 ms Energia RAM: 6.7100E-4 Energia CPU: 6.7100E-4 Energia Package: 0.0041	Tempo: 1 ms Energia RAM: 0.0000 Energia CPU: 3.0500E-4 Energia Package: 0.0033	Tempo: 0 ms Energia RAM: 4.8800E-4 Energia CPU: 4.8900E-4 Energia Package: 0.0033

Com base na tabela apresentada, é possível verificar que o **CheapestNear** é o que consome mais energia, tanto a nível de energia de *CPU* como a nível de energia da *RAM* e a nível de *package*. A nível de tempo, o **Autonomy** é o que despende do maior tempo, onde a diferença, em relação a todos os outros, é substancial.

Comparando os tamanhos dos diferentes ficheiros, é possível verificar que o ficheiro grande consome, substancialmente, um maior número de energia do que o ficheiro médio e que o

ficheiro pequeno.

5.2 Análise posterior ao Refactoring

Devido a um erro, que nos deparamos com o uso do RAPL, não conseguimos elaborar a análise de energia após o refactoring. O objectivo era elaborar uma tabela idêntica à apresentada anteriormente e tirar conclusões sobre o consumo de energia comparando o antes do refactoring e o após ao refactoring.O objectivo era elaborar uma tabela idêntica à apresentada anteriormente e tirar conclusões sobre o consumo de energia comparando o antes do refactoring e o após ao refactoring.

O objectivo era elaborar uma tabela idêntica à apresentada anteriormente e tirar conclusões sobre o consumo de energia comparando o antes do refactoring e o após ao refactoring.

6 Conclusão e trabalho futuro

Ao longo da resolução deste trabalho, o grupo de trabalho deparou-se com vários "stumbling blocks". Numa primeira etapa, as dificuldades residiam na decisão de quais ferramentas utilizar a fim de realizar as várias análises ao código fornecido. Ultrapassado este problema surgiu então a implementação do gerador de logs em que, mais uma vez, foram encontradas adversidades no uso do *State Monad*. O grupo resolveu este contratempo simplificando a solução e decidiu por não usar o *State Monad*. Não obstante a decisão, o código implementa o que é pedido. Ainda mais adiante no projecto, o grupo deparou-se ainda com mais impasses no uso da ferramenta **RAPL**, devido às suas condições específicas de funcionamento, mais concretamente não funcionar em máquinas virtuais.

Existem ainda aspectos que o grupo reconhece que deveria melhorar. Tais como: realizar testes unitários mais extensivos, corrigir todos os *smells* e implementar o gerador de logs, usando a *State Monad*.

Assim, este trabalho permitiu, para além de consolidar a matéria dada ao longo de todas aulas, constatar a relevância da fase de planeamento anterior à implementação da solução. Se não se cumprir esta fase, tendo como máximas a modificabilidade, manutenabilidade, compreensão, baixo acoplamento e baixa dependência, em implementações que sejam susceptíveis a modificações futuras, a vida de quem esteja responsável por realizar estas alterações irá ser dificultada em grande medida. Neste caso, a modificação requisitada era efectuar o *refactoring*, mas antes de iniciá-lo foi necessário percebê-lo e entender a lógica geral, o que não foi imediato devido à qualidade da implementação provida.