

UmCarroJá: Análise e Teste de Software

Henrique Faria A82200 and Sandra Baptista PG35390

Departamento de Informática, Universidade do Minho

Resumo Neste trabalho propusemo-nos a analisar e testar o software feito no âmbito da disciplina de *Programação Orientada a Objetos*.

Este relatório encontra-se estruturado em 4 secções: *Qualidade do Código Fonte*, *Refactoring da Aplicação*, *Teste da Aplicação* e *Análise de Desempenho da Aplicação*.

Foram também utilizadas as seguintes ferramentas para realizar o trabalho:

Eclipse, SonarQube, JStanley, IntelliJDEA, RAPL, EvoSuite.

Palavras-chave: Code Smells · Technical Debt · Eclipse · SonarQube · JStanley · IntelliJDEA · RAPL · EvoSuite

1 Tarefa 1: Qualidade do Código Fonte

Nesta tarefa utilizamos o SonarQube com o intuito de obter uma avaliação de qualidade do código, com análise estática para detectar *bugs*, *code smells* e vulnerabilidades.

1.1 SonarQube

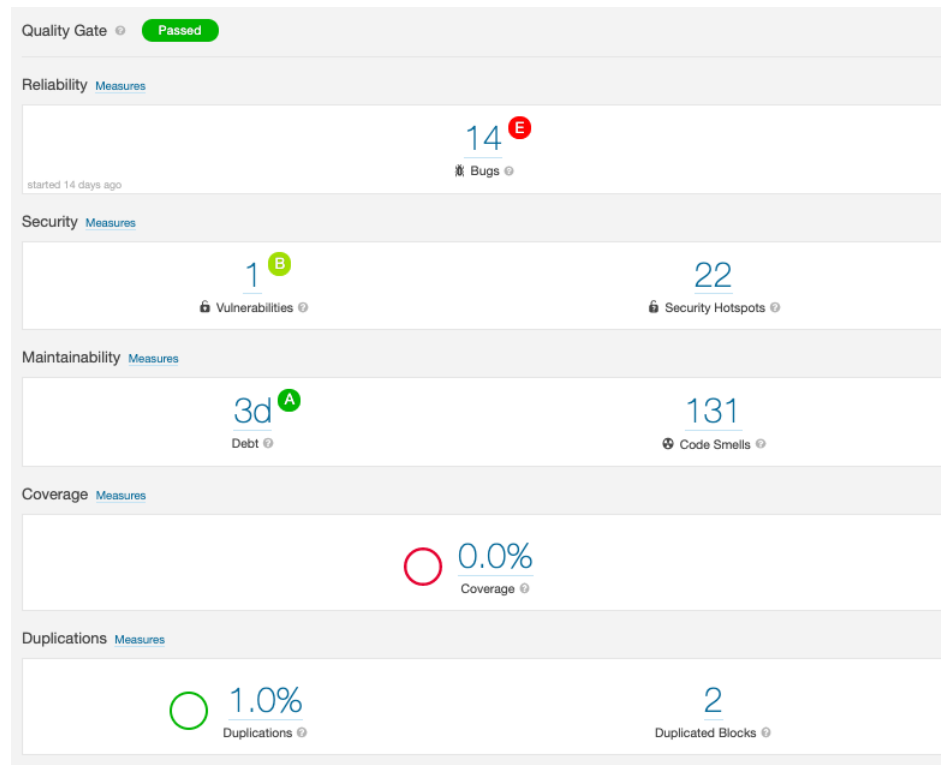


Figura 1. Menu geral de avaliação do SonarQube

Como se pode ver pela figura 1 este projeto possui alguns bugs, pelo menos 1 vulnerabilidade uma quantidade considerável de *code smells* e 2 blocos duplicados.

2 Tarefa 2: Refactoring da Aplicação

Na tarefa 2 utilizamos ferramentas para efectuar refactor e assim se eliminar os *bad smells* e *red smells* existentes no código fornecido.

O relatório detalhado dos tipos de erros e da sua gravidade bem como das respectivas soluções implementadas obteve-se com recurso ao Eclipse, já o relatório de erros surgiu da utilização do *Sonarqube* (incluindo as descrições dos erros e *code smells* encontrados):

2.1 Bugs

Versão com e sem *bugs*, com e sem *smells* (por tipos de *smells*).

- Não usar blocos *try/catch* ao escrever em ficheiros.
Ficheiro: UMCarroJa.java

Podemos visualizar nas seguintes imagens o código analisado e a solução respetiva.

```
public void save(String fName) throws IOException {
    FileOutputStream a = new FileOutputStream(fName);
    ObjectOutputStream r = new ObjectOutputStream(a);
    r.writeObject(this);
    r.flush();
    r.close();
}
```

Figura 2. Código com bug

```
public static UMCarroJa read(String fName) throws IOException, ClassNotFoundException {
    FileInputStream r = new FileInputStream(fName);
    ObjectInputStream a = null;
    a = new ObjectInputStream(r);
    UMCarroJa u = null;
    try{
        u = (UMCarroJa) a.readObject();
    } catch(Exception e) {
        LOGGER.info("Can't read the specified file!!\n");
    } finally {
        a.close();
    }
    return u;
}
```

Figura 3. Código corrigido

- Não usar blocos *try/catch* ao ler de ficheiros.
Ficheiro: UMCarroJa.java

Podemos visualizar nas seguintes imagens o código analisado e a solução respetiva.

```
public static UMCarroJa read(String fName) throws IOException, ClassNotFoundException {
    FileInputStream r = new FileInputStream(fName);
    ObjectInputStream a = new ObjectInputStream(r);
    UMCarroJa u = (UMCarroJa) a.readObject();
    a.close();
    return u;
}
```

Figura 4. Código com bug

```

public void save(String fName) throws IOException {
    FileOutputStream a = new FileOutputStream(fName);
    ObjectOutputStream r = null;
    r = new ObjectOutputStream(a);
    try{
        r.writeObject(this);
        r.flush();
    }catch(Exception e) {
        LOGGER.info("Can't write to file!!\n");
    } finally {
        r.close();
    }
}

```

Figura 5. Código corrigido

Note-se que é usado no fim dos blocos *try/catch* o bloco *finally* para fechar o descritor de escrita/leitura. Isto serve para, caso alguma coisa corra mal na escrita em/leitura de um ficheiro, o descritor ser fechado.

2.1.1 Critical Bugs:

- Guardar e reutilizar variáveis random.
Ficheiro: Traffic.java

Para resolver o problema basta verificar que o *random* estava a ser gerado sempre que a função *getTrafficDelay()* era invocada. Para resolver basta gerar o *random* uma única vez quando a classe for criada e usar o mesmo sempre que a função em causa for invocada.

```

class Traffic {
    Random b = new Random();
    public double getTrafficDelay(double delay) {
        int a = LocalDateTime.now().getHour();
        Random b = new Random();
        if(a == 18 || a == 8)
            return (b.nextDouble() % 0.6) + (delay % 0.2);
        if(a > 1 && a < 6)
            return (b.nextDouble() % 0.1) + (delay % 0.2);
        return (b.nextDouble() % 0.3) + (delay % 0.2);
    }
}

```

Figura 6. Recolocação do método de geração de um número *Random*

2.1.2 Major Bugs:

- Não obrigar a usar o método redefinido usando *override*.
Ficheiro: Car.java

Para resolver este problema da forma mais simples foi preciso renomear o método *equals* para o método *isEqual*, visto que usar um *@Override* sobre

este método obrigaria a implementação um método de super tipo e da função *hashCode()*.

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

Figura 7. Definição da função isEqual

2.1.3 Minor Bugs:

- Obrigar o override do equals e não o do método *hashCode()*.
Ficheiros: Car.java, Cars.java, Cliente.java, Owner.java, Parser.java, Rental.java, Rentals.java, User.java, Users.java.

Para corrigir este problema, basta definir um *hashCode()* que chame o método *super.hashCode()* como se pode ver na figura seguinte.

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

Figura 8. Solução do problema de @Override do método hashCode

2.2 Vulnerabilitys

- Utilizar *printStackTrace()* pode revelar informação sensível sobre o nosso código.
Ficheiro: Parser.java

Para evitar que tal vulnerabilidade ocorra, o *printStackTrace()* foi substituído por um *LOGGER.info()* que imprime uma mensagem de erro pré-determinada que não revela nada sobre a implementação do código que a originou.

```
} catch (IOException e) {
    e.printStackTrace();
    String msg = "IOException";
    LOGGER.info(msg);
}
```

Figura 9. Solução do problema de *Override* do método *hashCode*

2.3 CodeSmells

2.3.1 Critical CodeSmells:

- Possuir um método complexo com cerca de 290 linhas, é muito difícil manter e até mesmo perceber um código tão extenso.

Ficheiro: Controller.java

Para resolver o problema cada case do *switch* foi dividido em 1 função de complexidade inferior de média 15 linhas. podemos ver nas duas figuras em baixo, um dos cases e a função para o qual foi passado o código correspondente.

```
case Login:
    error = caseLogin();
    break;

public String caseLogin() {
    String error = "";
    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"; }
    return error;
}
```

Figura 10. Solução do problema de complexidade extrema do método run()

- Repetir várias vezes a atribuição da mesma string pode tornar o código confuso e ineficiente.

Para ultrapassar essa dificuldade essa string passou a ser criada e guarda numa constante quando um objeto da classe é inicializado e essa constante é depois atribuída quando necessário.

Ficheiros: Controller.java, Rental.java, Weather.java e Menu.java

- Usar switch sem caso *default*:. Bastou substituir o ultimo case "...": por *default*: para cumprir o mesmo objetivo do código anterior. No caso do ficheiro Menu.java o default foi adicionado após todos os cases existentes para garantir que o programa corria da forma correta.

Ficheiros: Controller.java, Car.java, Parser.java e Menu.java

- Ter constantes numa enumeração escritas em letras minúsculas. Basta escreve-las em maiúsculas para que passem a seguir a convenção. Para além disso todas as ocorrências destas palavras sofrerão a mesma modificação. Ficheiros: Car.java e Menu.java

- Atualizar uma variável *static* através de um método não estático.
Ficheiro: Rentals.java

Para contornar este problema, transformou-se a variável *static private int id* em *private int id*.

2.3.2 Major CodeSmells:

- A existência de blocos *try/catch* vazios não afeta o desempenho da aplicação mas pode ser um erro de falta de código, neste caso, assumimos que foi deixado propositadamente vazio, logo para não alterar o código restante preencheu-se o bloco com a atribuição de uma String vazia á variável erro.
Ficheiros: Controller.java e Main.java

```
catch (UnknownCompareTypeException ignored) {error = "";}}
```

Figura 11. Preenchimento do bloco vazio com uma linha de código que não afeta a aplicação

- A existência de blocos *try/catch* dentro de um case não é uma boa prática, deve-se criar um método e usar o *try/catch* dentro desta, sendo depois o método chamado dentro do case.
Ficheiros: Parser.java

Para corrigir isto basta criar o método com o bloco *try/catch* lá dentro.

- A utilização de uma classe que reporta uma exceção e não estende a class *Exception* viola a convenção, para corrigir o erro mudou-se a expressão *extends Throwable* para *extends Exception*.
Ficheiros: InvalidNewRentalException.java e InvalidNumberOfArgumentsException.java

O código corrigido é apresentado na seguinte figura.

```
public class InvalidNewRentalException extends Exception {
    private static final long serialVersionUID = 4378462538950802892L;
}
```

Figura 12. Troca de Throwable por Exception

- A utilização de uma classe que reporta uma exceção e não entende a class *Exception* viola a convenção, para corrigir o erro mudou-se a expressão *extends Throwable* para *extends Exception*.

Ficheiros: InvalidNewRentalException.java e InvalidNumberOfArgumentsException.java

O código corrigido é apresentado na seguinte imagem.

```
public class InvalidNewRentalException extends Exception {
    private static final long serialVersionUID = 4378462538950802892L;
}
```

Figura 13. Troca de Throwable por Exception

- Ao reportar algo o utilizador deve ser capaz de aceder aos logs facilmente, estes logs têm de ter um formato uniforme, devem ser guardados e dados sensíveis devem ser guardados de forma segura. A utilização de `LOGGER.info()` pode comprometer um destes aspetos. Portanto tivemos de importar a biblioteca `-textitjava.util.logging.Logger` e criar um *Logger* para guardar o output disponibilizado ao utilizador.

Ficheiros: Main.java e Menu.java

O código com o Logger é apresentado na seguinte imagem.

```
import java.util.logging.Logger;

public class Main {
    private final static Logger LOGGER = Logger.getLogger(Main.class.getName());
    public static void main(String[] args) {
        UMCarroJa model = new UMCarroJa();

        try {
            model = UMCarroJa.read(".tmp");
            LOGGER.info("adasdsada1");
        }
        catch (IOException | ClassNotFoundException e) {
            LOGGER.info("adasdsada2");
            new Parser("db/logsP00_carregamentoInicial.bak", model);
        }
        try { Thread.sleep(10000);} catch (Exception e) {}
        new Controller(model).run();
        try {
            model.save(".tmp");
        }
        catch (IOException ignored) {}
    }
}
```

Figura 14. Uso de Logger para seguir requerimentos dos outputs

- A utilização de muito parâmetros pode significar que esta classe está a fazer muitas coisas.

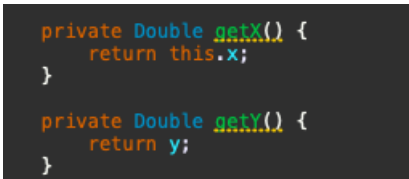
Ficheiros: Car.java e RegisterCar.java

Achamos por bem para já não alterar este codeSmell, pois a função recebe 9 argumentos quando só devia receber no máximo 7. O excesso de parâmetros não prejudica o desempenho de forma alguma.

- A presença de métodos que não são utilizados (dead code) deve ser removida.

Ficheiros: Point.java e StringBetter.java

Na figura em baixo podemos ver um exemplo dos métodos removidos. Na classe StringBetter.java removeu-se o método setStr().



```
private Double getX() {
    return this.x;
}

private Double getY() {
    return y;
}
```

Figura 15. Métodos removidos da classe Point.java

- A presença de uma variável com o mesmo nome da classe em que se insere pode confundir, as boas práticas indicam que devem ser nomes distintos. Portanto a variável *menu* foi renomeada para *mymenu*.

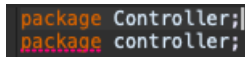
Ficheiro: Menu.java

2.3.3 Minor CodeSmells:

- Nomes de pacotes com letras maiúsculas.
Ficheiros: todos os packages excepto o main.java.

Para resolver este problema basta substituir as letras maiúsculas nos nomes dos packages por minúsculas, o eclipse trata de renomear o nome do package nas declarações feitas dentro dos ficheiros do próprio package.

Em seguida temos um exemplo da correção de uma dessas ocorrências.



```
package Controller;
package controller;
```

Figura 16. Solução do problema de renomear packages

- Utilização de métodos ineficientemente.
Ficheiros: Controller.java e Menu.java.

Neste caso num if é utilizado o método `size()` do `ArrayList` e seguidamente verifica-se se o array está vazio. A solução para esta ineficiência passa por invocar o método `isEmpty()` do `ArrayList`.

Em seguida temos um exemplo da correção de uma dessas ocorrências.

```
if (!R.size() == 0){
```

Figura 17. Solução do problema de ineficiência no uso de funções de Coleções na classe Controller

Como na classe *Menu.java* foi preciso trocar a execução do if com o else, para uma melhor percepção do que foi feito, a mudança será mostrada no figura seguinte.

```
if (this.prev.size() > 0) {
    this.menu = this.prev.pop();
    this.pickChildMenus();
} else {
    this.run = false;
}

if (this.prev.isEmpty()) {
    this.run = false;
} else {
    this.menu = this.prev.pop();
    this.pickChildMenus();
}
```

Figura 18. Solução do problema de ineficiência no uso de funções de Coleções na classe Menu

- Classe sem package.
Ficheiro: main.java.

Neste caso basta declarar o package a que a classe pertence, como se pode ver na figura seguinte.

```
package main.java;
```

Figura 19. Declaração do package a que a classe main.java pertence

- Utilização de métodos ineficientemente.
Ficheiro: Car.java.

Para colmatar a ineficiência do código basta mudar a negação de uma operação de `>`, extremamente custosa, para uma operação de `>=` que tem o

mesmo efeito.

A resolução encontra-se na figura seguinte.

```
return (this.position.distanceBetweenPoints(dest) * 1.2 <= this.range);
return !(this.position.distanceBetweenPoints(dest) * 1.2 > this.range);
```

Figura 20. Solução do problema de ineficiência do return

- Declaração de um clone() sem implementar Cloneable.
Ficheiros: Car.java, Cars.java, Cliente.java, Owner.java e Point.java.

Para resolver o problema basta mudar o nome dest para myclone no método, como se pode ver na seguinte figura.

```
public Car myclone() {
    return new Car(this);
}
```

Figura 21. Implementação do método clone() da classe Car

- Método devolve ArrayList em vez de List, dando informação sobre a implementação do método.
Ficheiros: Cars.java e Owner.java.

Para resolver o problema basta por o método a devolver uma interface genérica.

Note-se que as funções que invocam estes métodos devem ser corrigidas declarando o tipo recebido com (*ArrayList<Rental>*).

```
public ArrayList<Car> listOfCarType(Car.CarType b) {
    public List<Car> listOfCarType(Car.CarType b) {
```

Figura 22. Exemplo da mudança do tipo de interface retornada por um método

- Utilização desnecessária de parêntesis num filter.
Ficheiros: Cars.java e UmCarroJa.java.

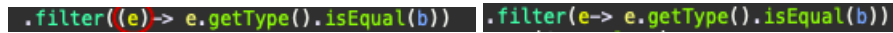


Figura 23. filter sem parêntesis

- Declaração de variáveis pela ordem errada, dificultando a leitura do código, estão declaradas da seguinte ordem: static final, final, private. Na figura podemos ver a ordem reescrita da forma correta.
Ficheiro: Rentals.java.

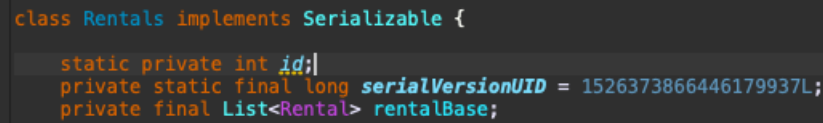


Figura 24. Ordenação correta das declarações de variáveis

- Declaração de um método em com a primeira letra maiúscula, o nome Original deste era *RESET()*.
Ficheiro: StringBetter.java.

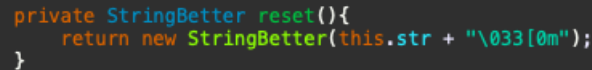


Figura 25. Declaração correta do método, usando letra minúscula no início

- Declaração de um método fazendo uso de um underscore no seu nome, seguindo a expressão regular da nomeação de métodos estes não devem ter underscore. De seguida apresenta-se um exemplo de um dos métodos corrigidos.
Ficheiro: StringBetter.java.

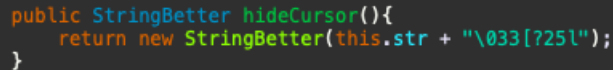


Figura 26. Declaração correta do método retirando o underscore e substituindo letra seguinte por maiúscula

- Declaração de variáveis começadas por letra maiúscula.
Ficheiro: NewLogin.java.

Apesar de não ser mostrado, no método NewLogin(), o nome e a password também foram modificados para fazer match com a renomeação demonstrada na figura.

```
private final String user;
private final String password;
```

Figura 27. Declaração correta das variáveis user e password

2.4 Security Hotspot

- A utilização da class *Random* não é segura por permitir que um atacante consiga prever o próximo random gerado e conseguir fazer-se passar por quem não deve.

para corrigir este problema bastou substituir a utilização de *Random* por *SecureRandom*. Ficheiros: Traffic.java e Weather.java

Adicionalmente foram tratados três problemas referidos pelo IDE Eclipse.

O primeiro problema diz respeito ao uso de uma variável error na classe Controller.java. Esta era usada para guardar uma mensagem explicativa do erro ocorrido durante a execução do programa, mas este nunca era mostrado ao utilizador caso ocorresse. Para isso bastou imprimir caso ocorra, com recurso ao System.out.println(), o erro obtido após um ciclo.

No segundo problema existe uma variável "private int id" no ficheiro Rentals.java que nunca é utilizada para nada, apenas é incrementada quando é adicionado um novo objeto rental mas não tem um propósito no código. Para resolver esta dependência bastou remover esta variável.

Por fim, na classe Menu.java em todos os métodos em que se criava um Scanner este nunca era fechado. Para resolver isto basta fechar o mesmo em todos esses métodos com a adição do código: "scanner.close();" no fim do mesmo.

2.5 Technical Debts

Ao fazer a análise dos technical debts foram analisadas versões guardadas no github que tenham sido modificadas de tal forma a que pudesse ter-se dado uma redução do tempo de execução estimado do programa. Os resultados obtidos pelo JStanley são mostrados na figura abaixo.

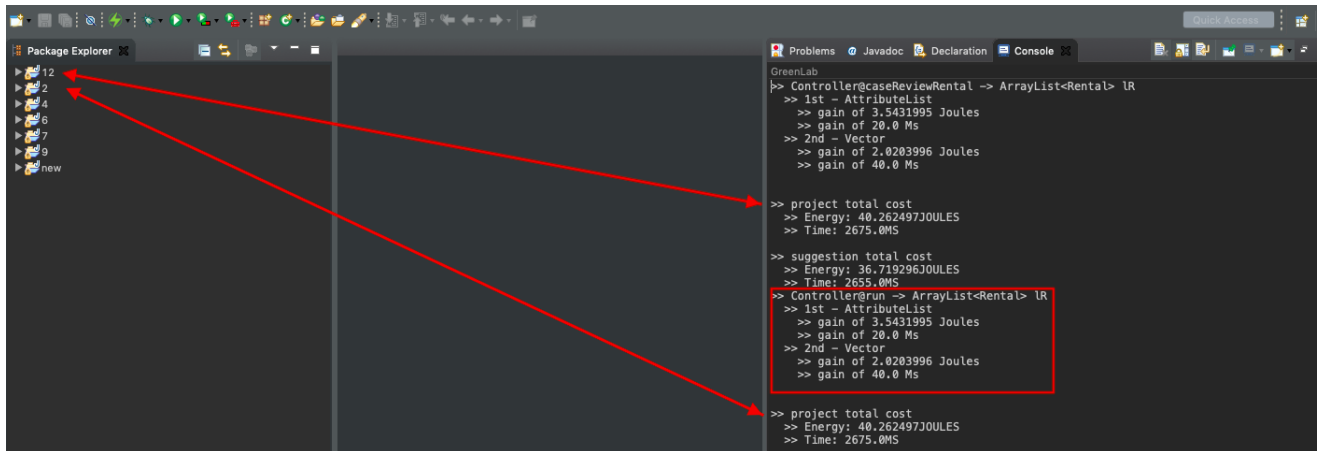


Figura 28. Resultados JStanley pré-correção de redSmells

Como se pode ver pela figura não se obteve diferença entre as duas versões assinaladas, sendo a 2 a versão com todos os code Smells e bugs originais e a 12 a versão atualmente corrigida (as setas ajudam a visualizar melhor quais os tempos correspondentes). No entanto, convém atentar que o JStanley detetou formas de melhorar o tempo de resposta da aplicação e/ou de melhorar o custo deste, sendo estas assinaladas por um retângulo.

A correção escolhida foi a aplicação de um AttributeList em vez do ArrayList. A leitura após a implementação destas sugestões apresenta-se de seguida assinalada por um retângulo vermelho.

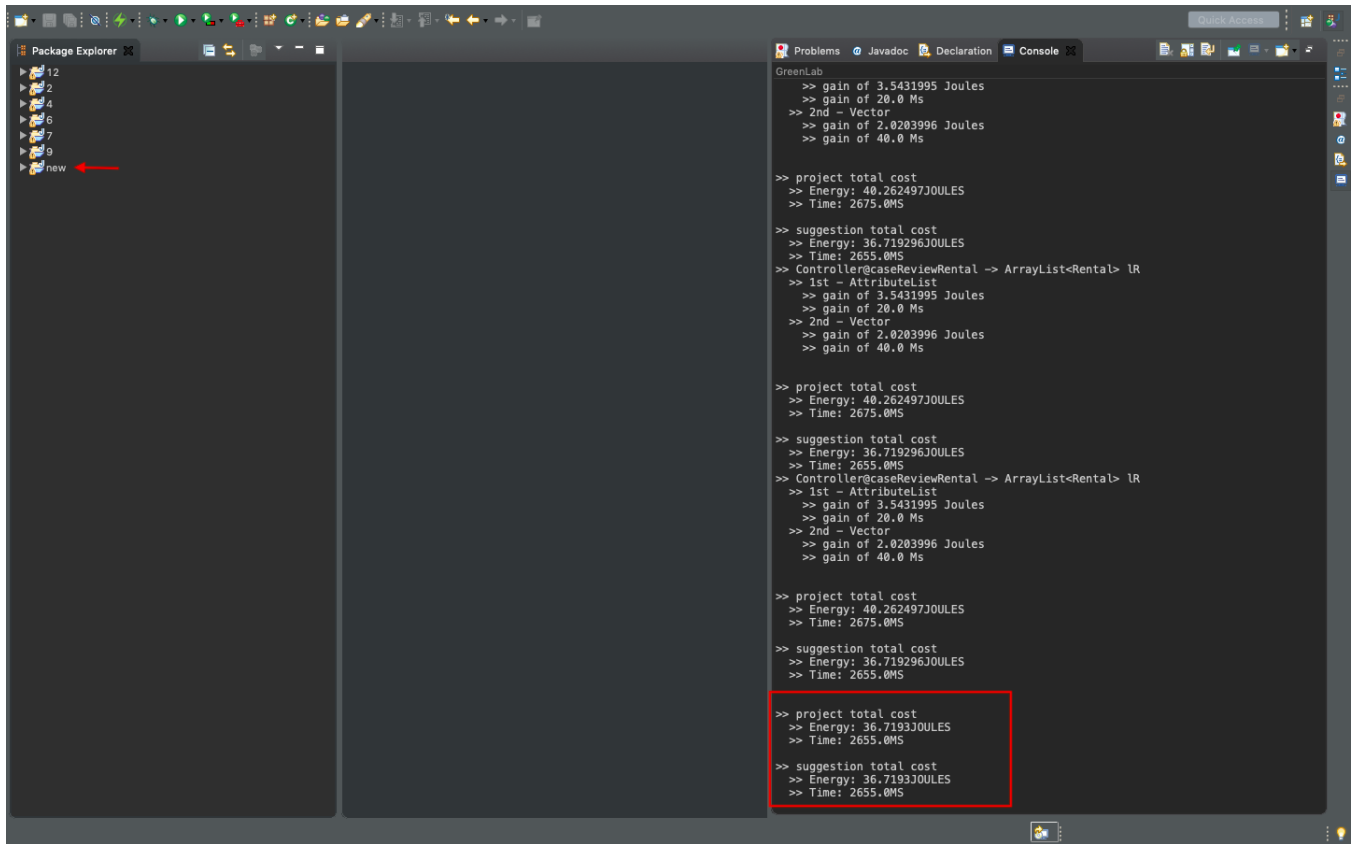


Figura 29. Resultados JStanley pós-correção de redSmells

Par após todos estes processos de correção de codeSmells e bugs, obtivemos o seguinte resultado no sonarQube.

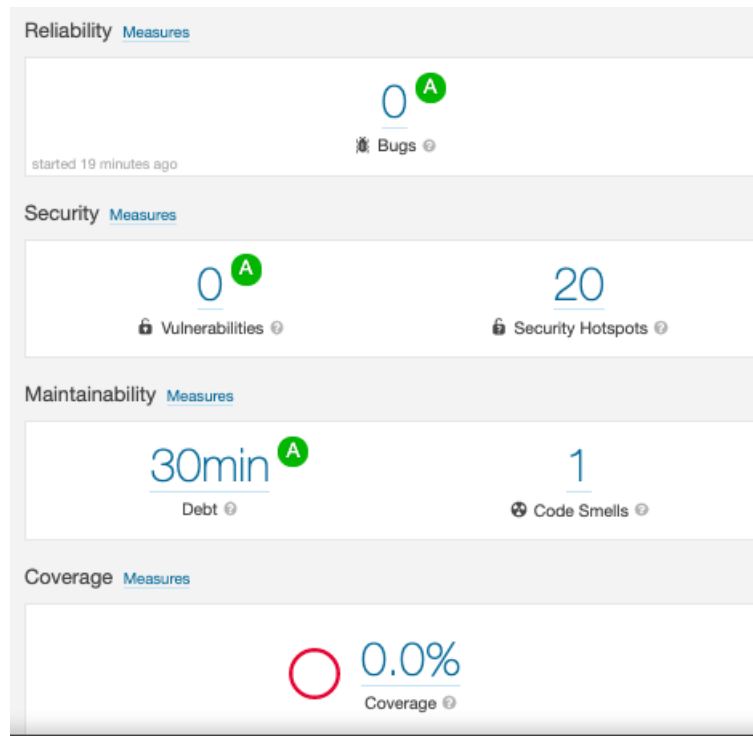


Figura 30. Todo o código foi corrigido com sucesso

3 Teste da Aplicação

3.1 Testes Unitários

Os testes unitários foram realizados sobre a classe *Client.java*. Na figura abaixo pode-se ver a classe criada para os testes.

Nesta classe, criaram-se as seguintes variáveis globais:

- 2 pontos;
- 1 cliente;
- 1 owner;

Adicionalmente criaram-se variáveis do objeto Car e Rental para verificar a correção de alguns métodos da classe visada.

```
class ClientTest {
    Point p = new Point( x: 1.0, y: 1.0);
    Client u1 = new Client(p, email: "u1@gmail.com", passwd: "passU1", name: "u1 nome", address: "morada do u1", nif: 999111111);
    Owner o = new Owner( email: "emailDono@gmail.com", name: "Dono1", address: "moradaDono1", nif: 912123123, passwd: "dono123");
    Point dest = new Point( x: 2.0, y: 2.0);

    @Test
    void getPos() {
        Point p1 = u1.getPos();
        boolean boolean0 = p1.equals(p);
        assertTrue(boolean0);
    }

    @Test
    void addPendingRental() throws Exception{
        Car c = new Car( numberPlate: "AA-11-11",o, Car.CarType.fromString("Electrico"), avgSpeed: 80.4, basePrice: 3, gasMileage: 3, range: 30,p, brand: "InventadaxD");
        Rental r = new Rental(c,u1,dest);
        u1.addPendingRental(r);
        boolean boolean1 = r.equals(u1.getPendingRates().get(0));
        assertTrue(boolean1);
    }

    @Test
    void rate() throws Exception{
        Car c = new Car( numberPlate: "AA-11-11",o, Car.CarType.fromString("Electrico"), avgSpeed: 80.4, basePrice: 3, gasMileage: 3, range: 30,p, brand: "InventadaxD");
        Rental r = new Rental(c,u1,dest);
        u1.rate(r, ratingCar: 1, ratingOwner: 1);
        boolean boolean2 = u1.getPendingRates().isEmpty();
        assertTrue(boolean2);
    }

    @Test
    void setPos() {
        Point x = new Point( x: 10.0, y: 10.0);
        u1.setPos(x);
        Point x1 = u1.getPos();
        boolean boolean3 = x.equals(x1);
        assertTrue(boolean3);
    }
}
```

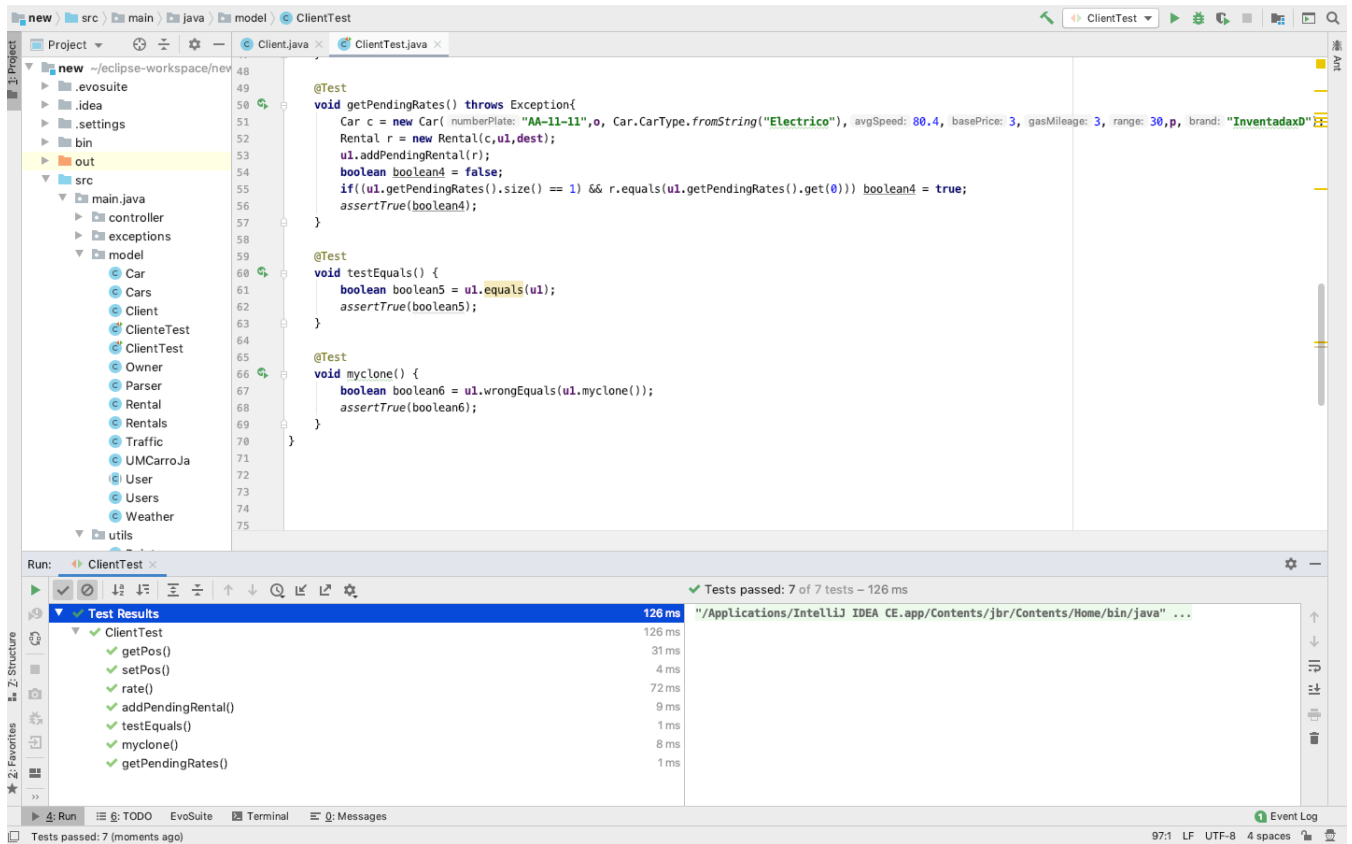


Figura 31. Classe ClientTest.java e resultado da sua aplicação

Note-se que para testar a correção do método `myclone()` foi criado um método extra na classe `Cliente` para verificar se os clientes partilhavam o mesmo apontador, esse método foi chamado `wrongEquals()`.

Na figura seguinte podemos ver o código coberto por este ficheiro de testes.

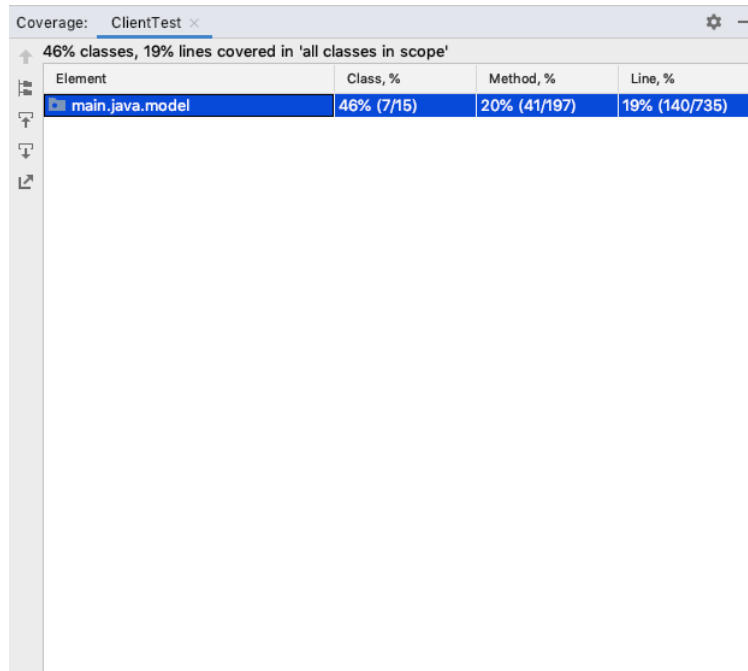


Figura 32. Cobertura da classe ClientTest.java sobre o package main.java.model

Nota1: Note-se que o facto do teste cobrir 46% das classes do package main.java.model e 20% dos métodos deve-se à necessidade, por parte da classe visada, de incluir objetos e recorrer a métodos de outras classes do mesmo package para ser completamente testada.

Nota2: Note-se que o teste de cobertura só abrange o package main.java.model. No entanto, a classe testada utiliza uma classe de nome *Point.java* de outro package.

3.2 Evosuite

3.2.1 Geração de testes Para compilar o Evosuite precisei de instalar um plugin chamado Choose Runtime no IntelliJ. Este foi usado para escolher a versão do JDK 1.8.0_231 como a versão a usar para correr o Evosuite. Após correr o Evosuite para gerar os teste o resultado mostra-se nas seguintes figuras.

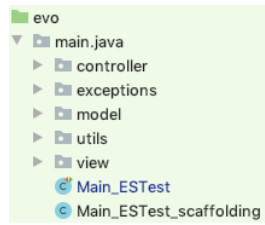


Figura 33. Testes class Main

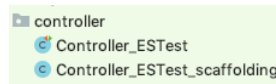


Figura 34. Testes package Controller

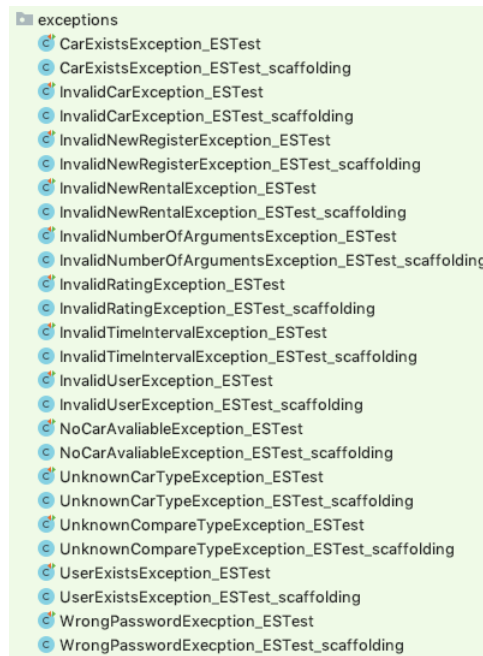


Figura 35. Testes do package exceptions

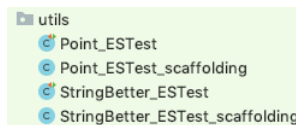


Figura 36. Testes do package utils

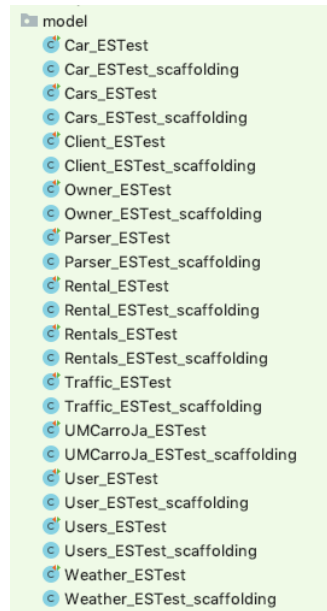


Figura 37. Testes do package model

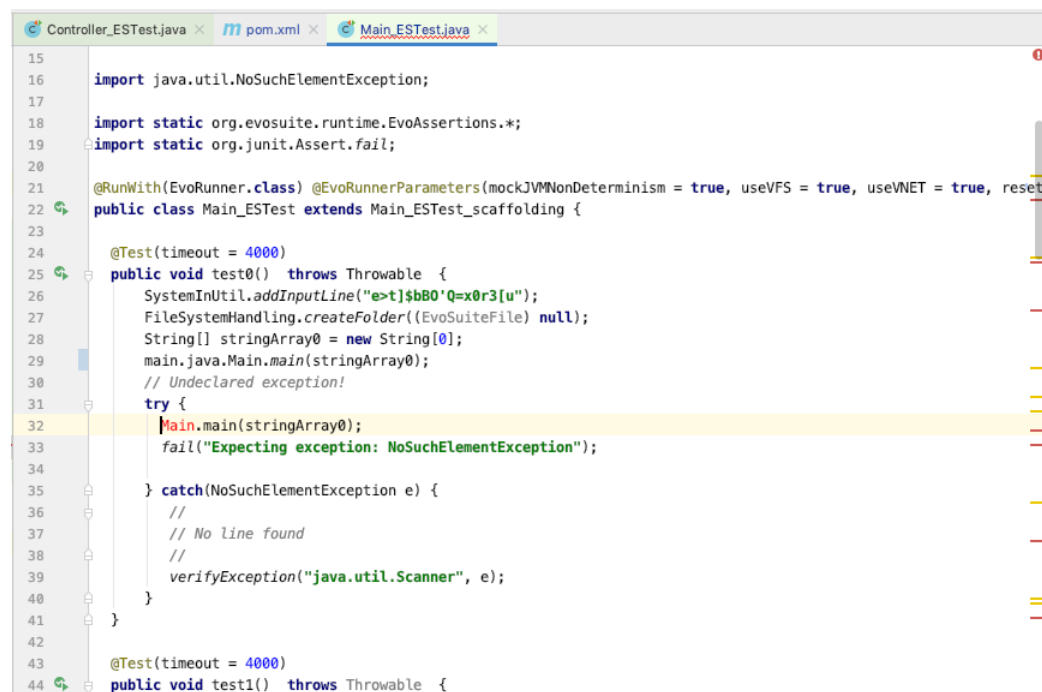


Figura 38. Testes do package view

Após o Evosuite gerar os ficheiros de teste, outro problema surgiu. Os ficheiros de teste não conheciam o JUnit apesar de este estar instalado. Para resolver este novo problema seguiram-se os seguintes passos:

- A primeira mudança para resolver este problema foi modificar a pasta onde os testes se encontravam para que fosse reconhecida como uma pasta de Sources root.
- Em segundo lugar a pasta do Evosuite foi marcada como Test Sources Root.
- Finalmente o ficheiro pom.xml foi alterado, adicionando como dependencia o Evosuite bem como adicionar nas propriedades o Evosuite.

Após a realização destes passos surgiu aquilo que parecia ser mais um problema. Nos ficheiros de teste gerados pelo Evosuite, este reconhece a class Main do projeto. No entanto, quando o Main é declarado este aparece a vermelho (como se estivesse errado) como se pode ver na figura abaixo.



```

15
16 import java.util.NoSuchElementException;
17
18 import static org.evosuite.runtime.EvoAssertions.*;
19 import static org.junit.Assert.fail;
20
21 @RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true, useVFS = true, useVNET = true, reset
22 public class Main_ESTest extends Main_ESTest_scaffolding {
23
24     @Test(timeout = 4000)
25     public void test0() throws Throwable {
26         SystemInUtil.addInputLine("e>t]5bB0'Q=x0r3{u");
27         FileSystemHandling.createFolder((EvoSuiteFile) null);
28         String[] stringArray0 = new String[0];
29         main.java.Main.main(stringArray0);
30         // Undeclared exception!
31         try {
32             Main.main(stringArray0);
33             fail("Expecting exception: NoSuchElementException");
34
35         } catch (NoSuchElementException e) {
36             //
37             // No line found
38             //
39             verifyException("java.util.Scanner", e);
40         }
41     }
42
43     @Test(timeout = 4000)
44     public void test1() throws Throwable {

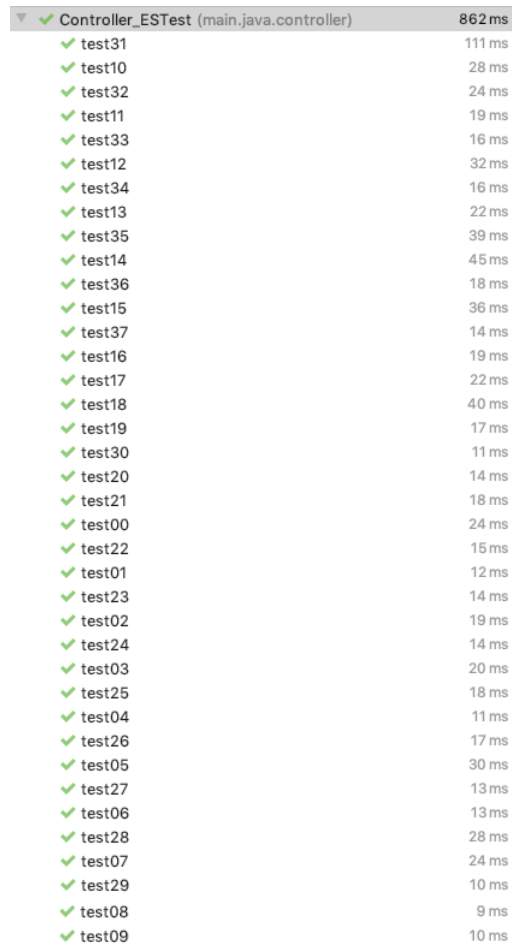
```

Figura 39. Excerto do ficheiro Main_ESTest

no entanto o erro não se verifica uma vez que podemos correr o ficheiro sem problemas, o que nos leva a pensar que deve ser um erro de identificação do IDE.

3.2.2 Verificação de classes utilizando os testes Após se ter confirmado a correta geração dos ficheiros de teste por parte do Evosuite procedeu-se a verificação das classes do nosso projeto com os mesmos.

A título de exemplo em seguida apresentam-se os resultados de alguns testes realizados sobre algumas das classes mais importantes do projeto em questão.

A screenshot of a test runner interface showing the results of a test suite named 'Controller_ESTest (main.java.controller)'. The suite has a total execution time of 862 ms and all tests passed, indicated by green checkmarks. The tests are listed in descending order of execution time, starting with 'test31' at 111 ms and ending with 'test09' at 10 ms. The interface has a light gray header for the suite name and total time, and a white background for the individual test results.

▼ ✓ Controller_ESTest (main.java.controller)	862 ms
✓ test31	111 ms
✓ test10	28 ms
✓ test32	24 ms
✓ test11	19 ms
✓ test33	16 ms
✓ test12	32 ms
✓ test34	16 ms
✓ test13	22 ms
✓ test35	39 ms
✓ test14	45 ms
✓ test36	18 ms
✓ test15	36 ms
✓ test37	14 ms
✓ test16	19 ms
✓ test17	22 ms
✓ test18	40 ms
✓ test19	17 ms
✓ test30	11 ms
✓ test20	14 ms
✓ test21	18 ms
✓ test00	24 ms
✓ test22	15 ms
✓ test01	12 ms
✓ test23	14 ms
✓ test02	19 ms
✓ test24	14 ms
✓ test03	20 ms
✓ test25	18 ms
✓ test04	11 ms
✓ test26	17 ms
✓ test05	30 ms
✓ test27	13 ms
✓ test06	13 ms
✓ test28	28 ms
✓ test07	24 ms
✓ test29	10 ms
✓ test08	9 ms
✓ test09	10 ms

Figura 40. Resultados da execução do ficheiro Controller_ESTest.java

▼ ✓ Main_ESTest (main.java)	530 ms
✓ test0	208 ms
✓ test1	67 ms
✓ test4	67 ms
✓ test2	124 ms
✓ test3	64 ms

Figura 41. Resultados da execução do ficheiro Main_ESTest.java

▼ ✓ Client_ESTest (main.java.model)	332 ms
✓ test20	26 ms
✓ test10	40 ms
✓ test21	7 ms
✓ test00	8 ms
✓ test11	12 ms
✓ test22	11 ms
✓ test01	11 ms
✓ test12	8 ms
✓ test02	13 ms
✓ test13	5 ms
✓ test03	12 ms
✓ test14	31 ms
✓ test04	8 ms
✓ test15	13 ms
✓ test05	14 ms
✓ test16	14 ms
✓ test06	9 ms
✓ test17	14 ms
✓ test07	12 ms
✓ test18	24 ms
✓ test08	15 ms
✓ test19	12 ms
✓ test09	13 ms

Figura 42. Resultados da execução do ficheiro Client_ESTest.java

3.3 Tests Coverage using JaCoCo

Para realizar os testes com "coverage" recorreremos ao JaCoCo, como se pode ver na imagem abaixo.

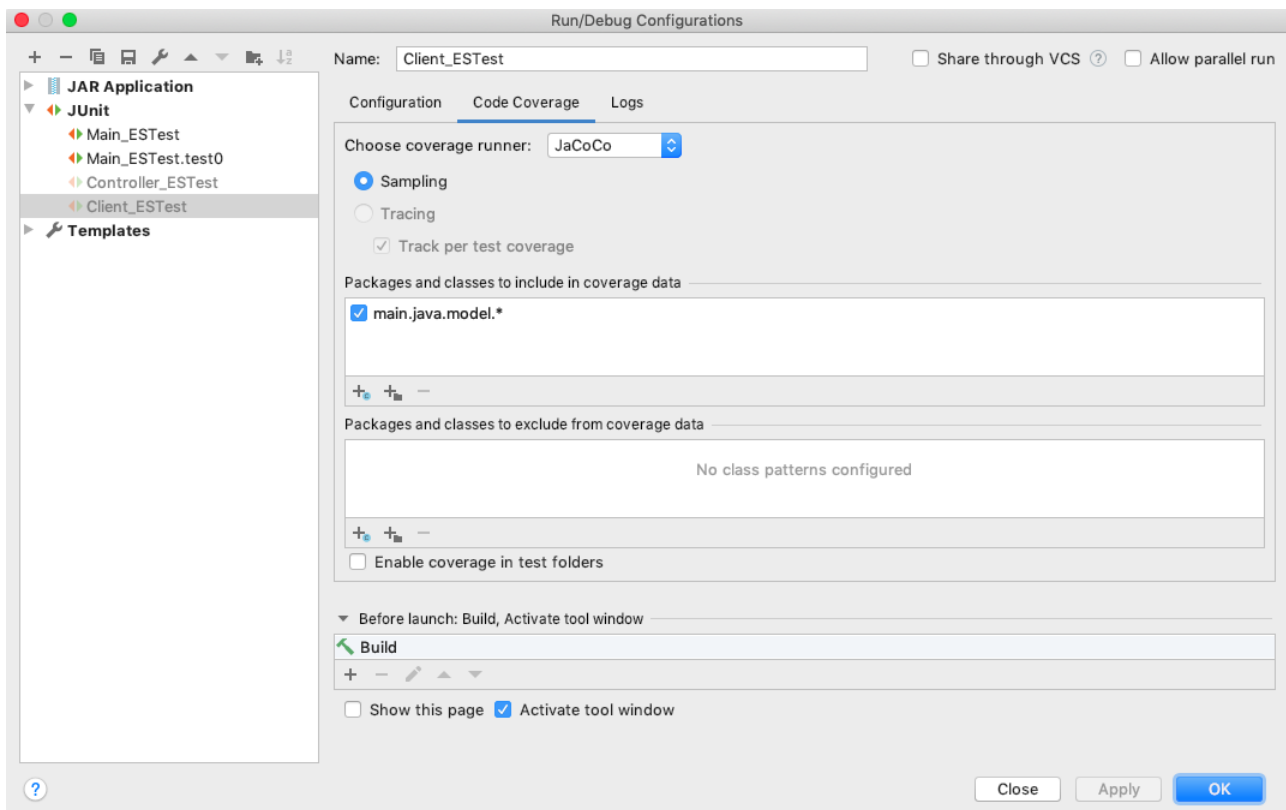


Figura 43. Configuração do Code Coverage para usar o plugin JaCoCo

Agora para correr os testes ao nosso programa basta-nos clicar no botão do canto superior direito assinalado na imagem seguinte para correr os testes com coverage.

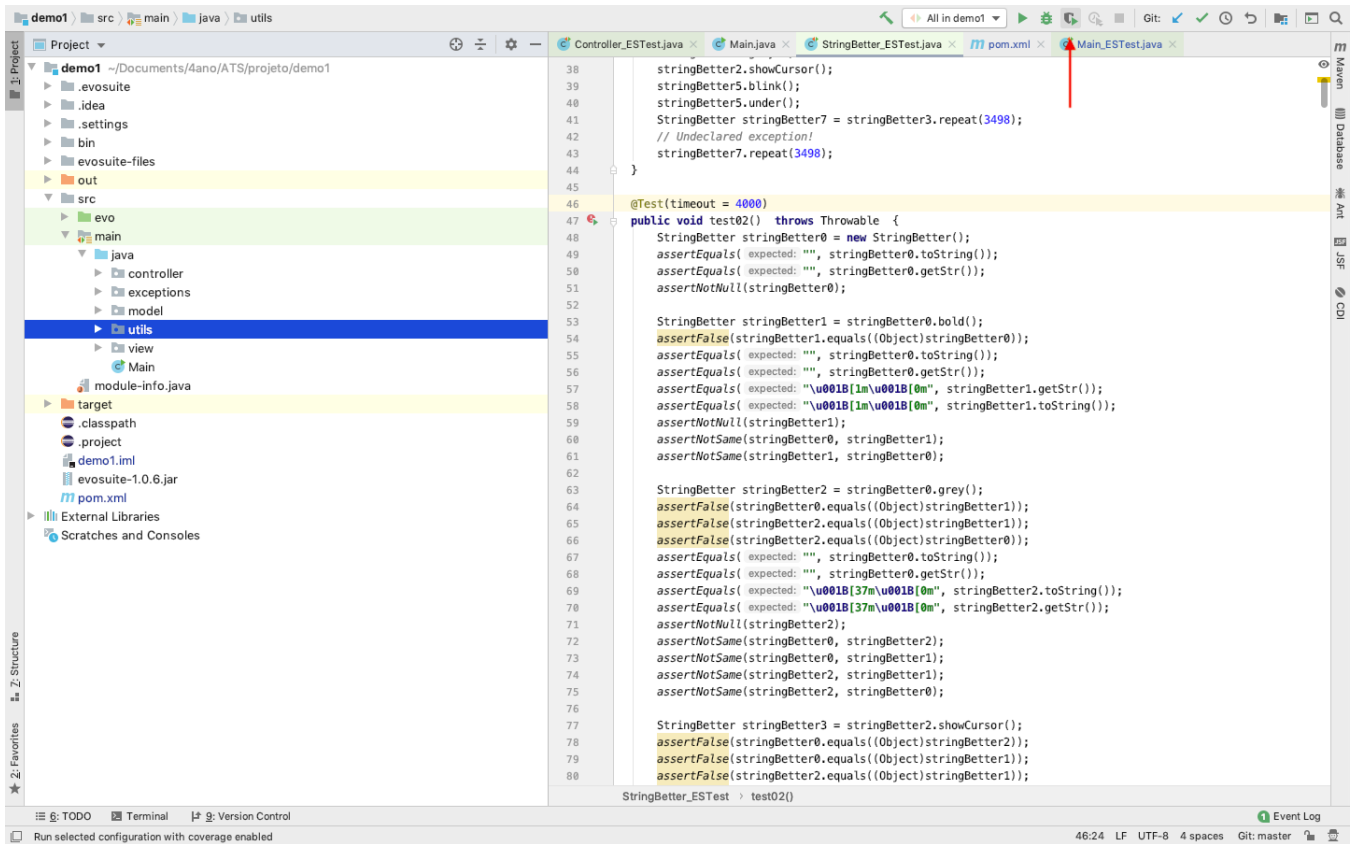


Figura 44. Execução dos testes usando coverage com o plugin JaCoCo

Por fim podemos ver os resultados do coverage das classes de cada package, em percentagem, na imagem abaixo.

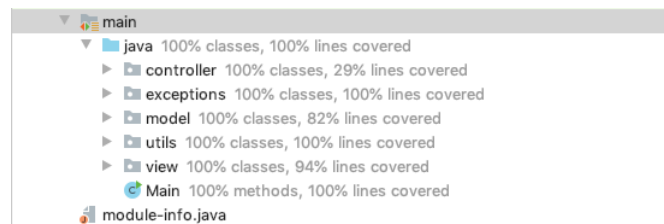


Figura 45. Resultados do coverage utilizando o plugin JaCoCo

Nota3: Note-se que um dos testes falhou na primeira execução, como se pode ver na seguinte imagem.



Figura 46. Falha detetada no teste 01 referente a Java Heap Space

Este problema acontece porque a Máquina virtual do Java tem um tamanho máximo de heap finito, isto leva a um erro como o que se pode verificar na imagem acima.

3.4 Geração de Ficheiros Log

A geração de ficheiros de log foi feita com recurso á linguagem de programação *Haskell* e á biblioteca *Test.QuickCheck*.

Para estes registos foram criadas as estruturas que se podem ver nas imagens seguintes.

```

data NovoProprietario = NovoProprietario Nome NIF Email Morada
    deriving Show

data Cliente          = NovoCliente Nome NIF Email Morada X Y --X e Y são coordenadas
    deriving Show

data NovoCarro        = Novocarro Tipo Marca Matricula NIF VelocidadeMedia PrecoKm Consumo Autonomia X Y
    deriving Show

data Aluger           = Aluger NIF X Y Tipo Preferencia
    deriving Show

data Classificar      = ClassificacaoM Matricula Nota
    | ClassificacaoN NIF Nota
    deriving Show

data Resultado        = Res [Cliente] [NovoProprietario] [NovoCarro] [Aluger] [Classificar]
    deriving Show

```

Figura 47. Estruturas Criptográficas dos registos

```

type Nome           = String
type NIF            = Int
type Email          = String
type Morada         = String
type X              = Float
type Y              = Float
type Tipo           = String
type Marca          = String
type Matricula      = String
type VelocidadeMedia = Int
type PrecoKm        = Float
type Consumo        = Float
type Autonomia      = Int
type Preferencia    = String
type Nota           = Int

```

Figura 48. Tipos dos dados dos registos

Foi criada uma função, com as respetivas funções auxiliares, para criar e preencher uma lista para cada estrutura mostrada anteriormente sendo a ultima estrutura aquela que agrega todas as outras para serem imprimidas como um resultado no ficheiro *logsPOO_carregamentoInicial.bak*.

3.4.1 Bug extra

A criação deste ficheiro de forma aleatória permitiu corrigir um erro que passou despercebido ao SonarQube. No ficheiro `Weather.java`, temos o seguinte:

```
private static final String[] seasons = {
    winter, winter,
    spring, spring, spring,
    summer, summer, summer,
    fall, fall, fall,
    winter
};

private String getSeason() {
    return seasons[LocalDateTime.now().getMonthValue()];
}
```

Se repararmos na lista `seasons` vemos que tem 12 posições, ou seja varia entre as posições 0 e 11. No entanto ao utilizarmos o método `LocalDateTime.now().getMonthValue()` estamos a obter valores entre 1 e 12.

Este bug foi corrigido prontamente modificando o código para o que se apresenta em seguida.

```
private static final String[] seasons = {
    winter, winter,
    spring, spring, spring,
    summer, summer, summer,
    fall, fall, fall,
    winter
};

private String getSeason() {
    return seasons[LocalDateTime.now().getMonthValue() - 1];
}
```

4 Tarefa 4: Análise de Desempenho da Aplicação

Com o fim de elaborar uma análise de desempenho da aplicação **UmCarroJá**, utilizou-se a solução *Running Average Power Limit - RAPL*.

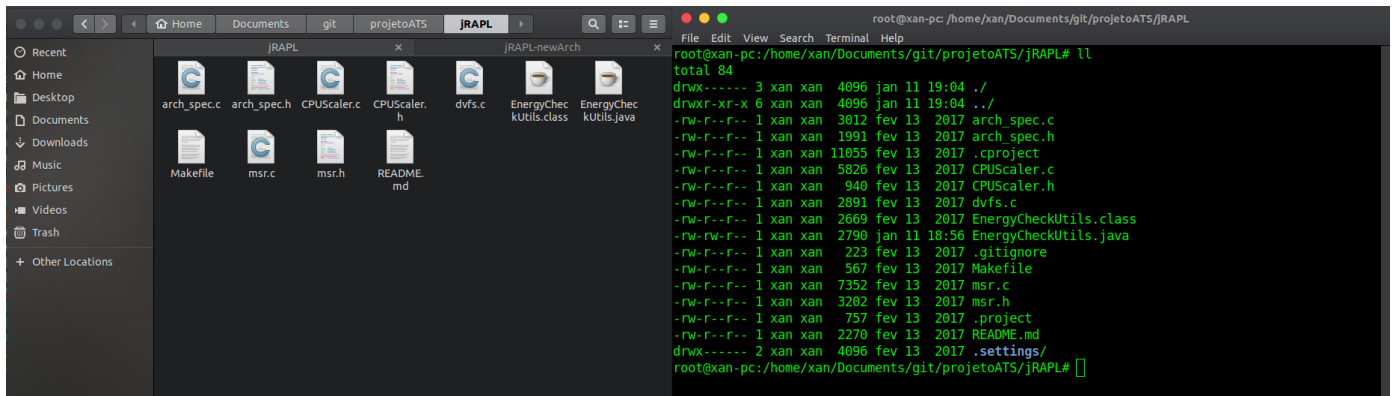


Figura 49. Pasta inicial do jRAPL

Antes de se poder correr o *RAPL* é necessário correr o comando *make* com o intuito de compilar os ficheiros.

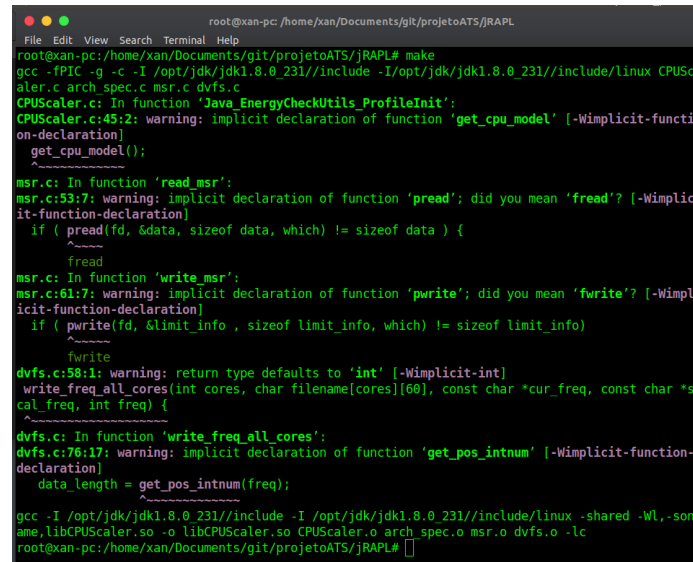


Figura 50. jRAPL make

Para além de executar o comando *make* é necessário, no projeto, apagar o ficheiro *.tmp* da respetiva pasta para que no código responsável por carregar a base de dados se use o ficheiro previamente gerado *generatedOutput.bak*.

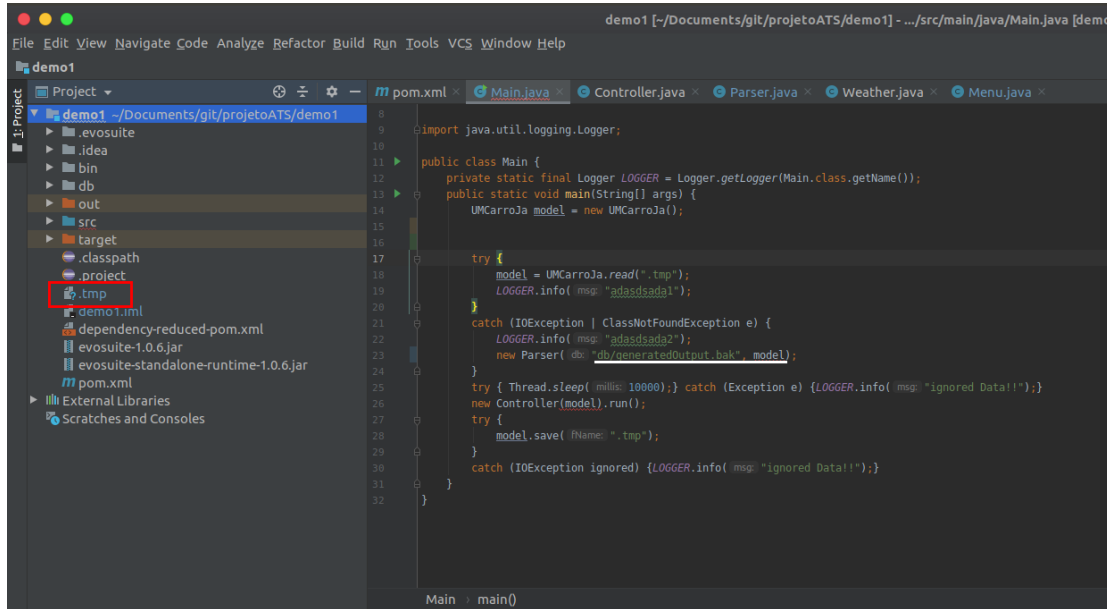


Figura 51. Remover *.tmp* do programa UmCarroJá

Para obter os dados de energia é necessário correr o ficheiro java do *RAPL* chamado *EnergyCheckUtils* que é responsável por indicar o valor de energia do dram / uncore gpu energy (depende da arquitetura da cpu), do CPU e do pacote de energia.

Quando corremos pela primeira vez o java o *EnergyCheckUtils* deu erro de passer dos dados, por isso foi necessário efectuar uma alteração no código do mesmo, ver imagem em baixo.

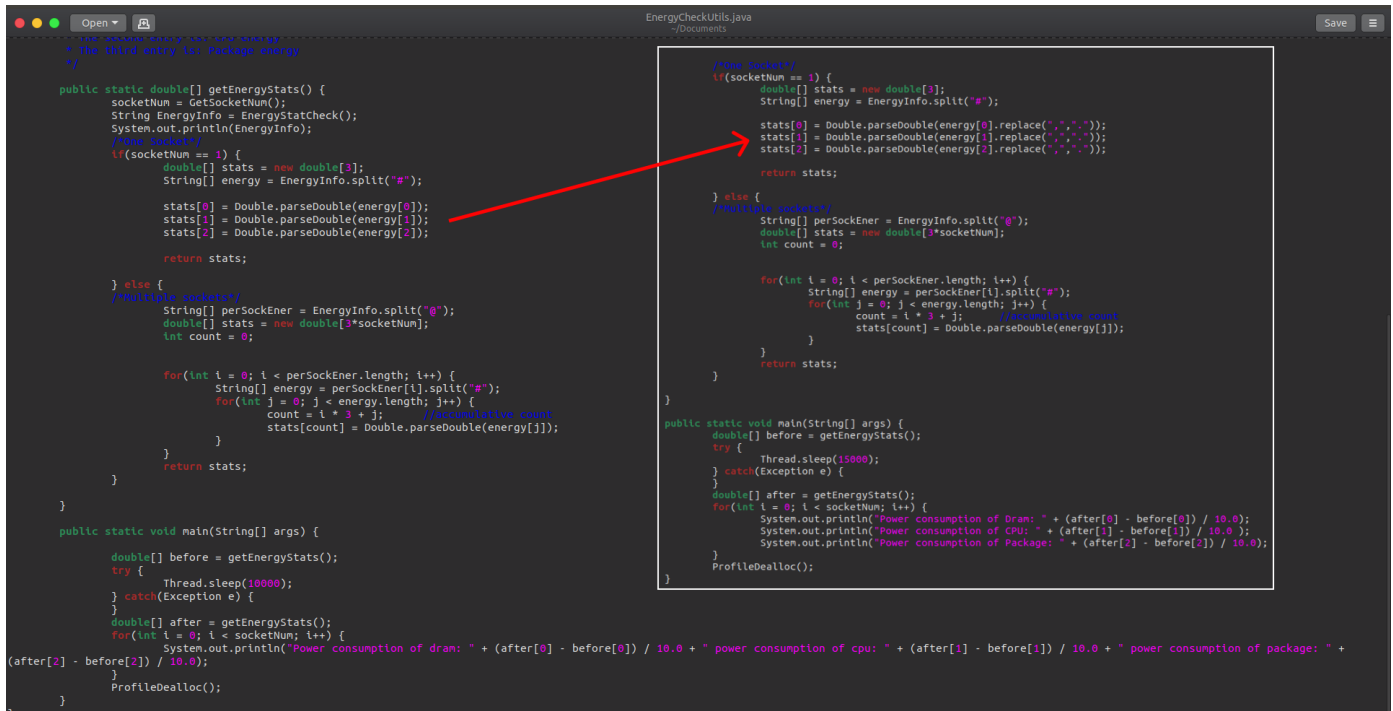


Figura 52. Alterações no ficheiro EnergyCheckUtils

Depois de corrigir o EnergyCheckUtils procedemos à execução dos comandos necessários para obter os dados de energia.

Comandos:

1. *sudo modprobe msr*
2. *javac EnergyCheckUtils.java*
3. *java EnergyCheckUtils*

```

root@xan-pc: /home/xan/Documents/git/projetoATS/jRAPL# sudo modprobe msr
root@xan-pc: /home/xan/Documents/git/projetoATS/jRAPL# javac EnergyCheckUtils.java
root@xan-pc: /home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils

```

Figura 53. Comandos para correr o RAPL

Para testar o UmCarroJá é primeiro necessário correr o EnergyCheckUtils e depois correr o UmCarroJá e assim obter os dados de energia do mesmo. Assim

efectuamos quatro testes para a primeira versão da solução seguidos de quatro para a versão final.

4.1 Primeira Versão do Código

Resultados da versão inicial da aplicação, ainda com *bugs* e *code smells*.

- generatedOutput5k: base de dados com 5 mil utilizadores, 5 mil proprietários, 5 mil veículos e 10 mil avaliações 5 mil sobre os utilizadores e 5 mil sobre os proprietários;

```
xan@xan-pc:~/Documents/git/projetoATS/jRAPL$ sudo java EnergyCheckUtils
EnergyInfo: 2439,361755#2892,066162#5329,916504
EnergyInfo: 2463,544922#2958,764038#5419,311340
Power consumption of Dram: 2.418316700000014
Power consumption of CPU: 6.669787599999973
Power consumption of Package: 8.939483600000004
xan@xan-pc:~/Documents/git/projetoATS/jRAPL$ sudo java EnergyCheckUtils
EnergyInfo: 2496,653076#2971,168579#5466,154602
EnergyInfo: 2518,030457#3010,242004#5526,659424
Power consumption of Dram: 2.1377380999999787
Power consumption of CPU: 3.9073425000000044
Power consumption of Package: 6.0504822000000078
```

Figura 54. Testes elaborados com RAPL à solução UmCarroJá para 5k elementos

- generatedOutput10k: base de dados com 10 mil utilizadores, 10 mil proprietários, 10 mil veículos e 20 mil avaliações 10 mil sobre os utilizadores e 10 mil sobre os proprietários;

```
xan@xan-pc:~/Documents/git/projetoATS/jRAPL$ sudo java EnergyCheckUtils
EnergyInfo: 2572,320251#3034,432800#5607,406738
EnergyInfo: 2594,652893#3072,702515#5667,698486
Power consumption of Dram: 2.233264199999985
Power consumption of CPU: 3.8269714999999906
Power consumption of Package: 6.0291748000000055
xan@xan-pc:~/Documents/git/projetoATS/jRAPL$ sudo java EnergyCheckUtils
EnergyInfo: 2699,708313#3211,569580#5914,436584
EnergyInfo: 2721,530029#3247,057495#5972,026184
Power consumption of Dram: 2.182171599999992
Power consumption of CPU: 3.5487915000000156
Power consumption of Package: 5.7589600000000025
```

Figura 55. Testes elaborados com RAPL à solução UmCarroJá para 10k elementos

- generatedOutput25k: base de dados com 25 mil utilizadores, 25 mil proprietários, 25 mil veículos e 50 mil avaliações 25 mil sobre o utilizador e 25 mil sobre o proprietário;

```
xan@xan-pc:~/Documents/git/projetoATS/jRAPL$ sudo java EnergyCheckUtils
EnergyInfo: 2572,320251#3034,432800#5607,406738
EnergyInfo: 2594,652893#3072,702515#5667,698486
Power consumption of Dram: 2.233264199999985
Power consumption of CPU: 3.8269714999999906
Power consumption of Package: 6.0291748000000055
xan@xan-pc:~/Documents/git/projetoATS/jRAPL$ sudo java EnergyCheckUtils
EnergyInfo: 2699,708313#3211,569580#5914,436584
EnergyInfo: 2721,530029#3247,057495#5972,026184
Power consumption of Dram: 2.182171599999992
Power consumption of CPU: 3.5487915000000156
Power consumption of Package: 5.7589600000000025
```

Figura 56. Testes elaborados com RAPL à solução UmCarroJá para 25k elementos

- generatedOutput50k: base de dados com 50 mil utilizadores, 50 mil proprietários, 50 mil veículos e 100 mil avaliações 50 mil sobre o utilizador e 50 mil sobre o proprietário;

```
xan@xan-pc:~/Documents/git/projetoATS/jRAPL$ sudo java EnergyCheckUtils
EnergyInfo: 2764,104553#3271,156738#6040,661621
EnergyInfo: 2788,286255#3375,537476#6168,268066
Power consumption of Dram: 2.418170199999986
Power consumption of CPU: 10.438073799999984
Power consumption of Package: 12.760644499999945
xan@xan-pc:~/Documents/git/projetoATS/jRAPL$ sudo java EnergyCheckUtils
EnergyInfo: 2811,185059#3380,674744#6197,259949
EnergyInfo: 2834,284607#3474,469421#6313,186707
Power consumption of Dram: 2.309954800000014
Power consumption of CPU: 9.379467699999987
Power consumption of Package: 11.592675799999961
```

Figura 57. Testes elaborados com RAPL à solução UmCarroJá 50k elementos

4.2 Versão Final

Resultados da versão final da aplicação, com *bugs* e *code smells* corrigidos.

- generatedOutput5k: base de dados com 5 mil utilizadores, 5 mil proprietários, 5 mil veículos e 10 mil avaliações 5 mil sobre os utilizadores e 5 mil sobre os proprietários;

```

root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
EnergyInfo: 4255,911804#3094,301025#7123,120911
EnergyInfo: 4278,785278#3135,013489#7187,146912
Power consumption of Dram: 2.2873473999999999
Power consumption of CPU: 4.0712463999999973
Power consumption of Package: 6.4026001000000018
root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
EnergyInfo: 4337,710327#3161,968323#7273,624329
EnergyInfo: 4359,046204#3193,721558#7326,833557
Power consumption of Dram: 2.13358770000000436
Power consumption of CPU: 3.1753235000000013
Power consumption of Package: 5.32092279999999715

```

Figura 58. Testes elaborados com RAPL à solução UmCarroJá para 5k elementos

- generatedOutput10k: base de dados com 10 mil utilizadores, 10 mil proprietários, 10 mil veículos e 20 mil avaliações 10 mil sobre os utilizadores e 10 mil sobre os proprietários;

```

root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
EnergyInfo: 4501,067566#3303,098267#7585,912415
EnergyInfo: 4521,340332#3328,653381#7632,494812
Power consumption of Dram: 2.0272766000000005
Power consumption of CPU: 2.5555114000000023
Power consumption of Package: 4.6582397000000013
root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
EnergyInfo: 4546,639221#3341,958008#7671,405029
EnergyInfo: 4566,629272#3367,270874#7716,339966
Power consumption of Dram: 1.99900509999999767
Power consumption of CPU: 2.53128659999999756
Power consumption of Package: 4.4934936999999999

```

Figura 59. Testes elaborados com RAPL à solução UmCarroJá para 10k elementos

- generatedOutput25k: base de dados com 25 mil utilizadores, 25 mil proprietários, 25 mil veículos e 50 mil avaliações 25 mil sobre o utilizador e 25 mil sobre o proprietário;

```

root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
EnergyInfo: 4612,877869#3400,473145#7799,972473
EnergyInfo: 4633,834778#3432,461914#7853,389282
Power consumption of Dram: 2.095690900000045
Power consumption of CPU: 3.1988769000000046
Power consumption of Package: 5.3416809000000285
root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
EnergyInfo: 4671,244385#3453,142700#7913,595947
EnergyInfo: 4691,562927#3479,147583#7960,260193
Power consumption of Dram: 2.031854199999998
Power consumption of CPU: 2.6004883000000065
Power consumption of Package: 4.6664246000000028

```

Figura 60. Testes elaborados com RAPL à solução UmCarroJá para 25k elementos

- generatedOutput50k: base de dados com 50 mil utilizadores, 50 mil proprietários, 50 mil veículos e 100 mil avaliações 50 mil sobre o utilizador e 50 mil sobre o proprietário;

```

root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
EnergyInfo: 4735,014343#3504,845215#8032,432861
EnergyInfo: 4756,664673#3538,578125#8088,689148
Power consumption of Dram: 2.1650330000000393
Power consumption of CPU: 3.3732910000000174
Power consumption of Package: 5.6256287000000016
root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
EnergyInfo: 4771,161255#3550,357605#8115,049927
EnergyInfo: 4791,629028#3576,755066#8162,164795
Power consumption of Dram: 2.04677730000000307
Power consumption of CPU: 2.6397461000000002
Power consumption of Package: 4.7114867999999966

```

Figura 61. Testes elaborados com RAPL à solução UmCarroJá 50k elementos

5 Conclusão

Neste trabalho começamos por examinar os problemas do código que nos foi entregue, isto foi feito com recurso ao SonarQube. Após este procedimento passamos a fazer o refactoring da aplicação fazendo uso de ferramentas como o auto-refactor do IntelliJ e utilizando as dicas oferecidas pelas regras do SonarQube e o IDE do Eclipse. O refactoring começou por ser realizado no bugs, seguindo-se as vulnerabilidades e os code smells. No fim das correções ficamos com apenas 1 code smell.

Posteriormente passamos aos testes da aplicação refactorizada. Primeiro executamos um teste unitário sobre a classe *Cliente.java*, este teste serviu para demonstrar os nossos conhecimentos sobre a aplicação de testes a um código. De seguida usamos o *EvoSuite* para gerar automaticamente testes sobre todas as classes do nosso projeto. E para correr os testes verificando a cobertura destes face ao código escolhemos a opção correr testes com cobertura usando JaCoCo como o Coverage runner.

Por fim criamos um ficheiro chamado *quick.hs* para gerar o novo ficheiro de input do programa e posteriormente fizemos a análise de desempenho da aplicação, antes e depois do refactoring.

Nas imagens finais podemos ver que da aplicação original para a aplicação refactorizada houve uma melhoria em termos de energia consumida, especialmente quanto maiores os ficheiros, nos quais se obtém menos de metade do consumo face a aplicação original.

Em suma, existem vantagens em fazer o refactoring de um código, pois este pode permitir poupar energia e ser mais rápido e eficiente que o original. Para além disso, este possibilita uma mais fácil manutenção de código futuro e correção de partes de código que podem levar a um mau desempenho ou a uma falha da aplicação no futuro.

Referências

1. <http://www.evosuite.org/>
2. <https://www.eclipse.org/ide/>
3. <https://www.sonarqube.org/>
4. <https://greensoftwarelab.github.io/jStanley/>
5. https://www.jetbrains.com/idea/?gclid=CjwKCAiA6vXwBRBKEiwAYE7iS8pVgLLs1Jm9lD19jLyM-snkTFQII9BD9_MAsFrgBSS_Q-izV0URsBoC-IkQAvD_BwE
6. <https://01.org/community>
7. <https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl>
8. G. Ann Campbell, Patroklos P. Papapetrou, Olivier Gaudin. *SonarQube in Action*