

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 Analizar e testar o software feito no ambito 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

Palavras-chave: Code Smells · Technical debt

1 Qualidade do Código Fonte

1.1 SonarQube

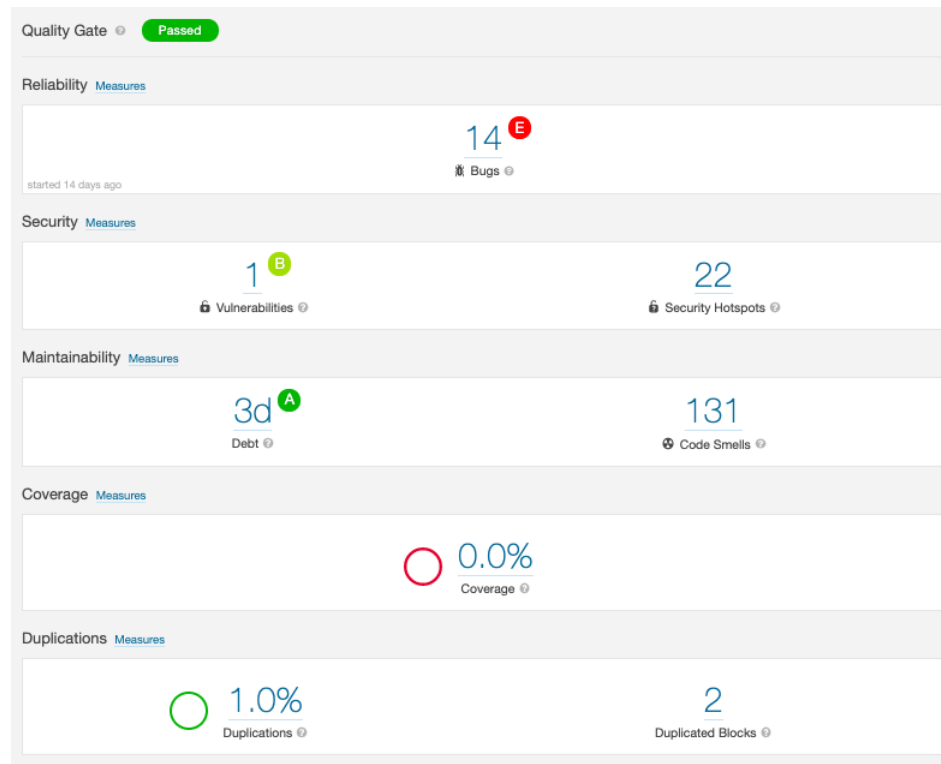


Figura 1. Menu geral de avaliação do SonarQube

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 Refactoring da Aplicação

De seguida apresenta-se um relatório detalhado dos tipos de erros e da sua gravidade bem como das respetivas soluções implementadas com recurso ao Eclipse e ao relatório de erros detalhado 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*) -> discriminar o impacto dos *smells*

Blocker Bugs:

- 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.

Foi corrigido um bug adicional que ocorreria na aplicação ao ser usada em Dezembro. No ficheiro Weather.java estava a ser calculada a posição do array seasons a ser usada da seguinte maneira: `LocalDateTime.now().getMonthValue()`, se atentar nos resultados desta resolução verá que em Dezembro é retornado 12, como o array começa em 0 a posição máxima deste é 11 e portanto incorre-se num erro, basta para isso subtrair 1 unidade ao resultado do método descrito na obtenção do número do mês.

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*

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*

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

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 guardada 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*.

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 `System.out.println()` 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

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 inefficientemente.
Ficheiros: Controller.java e Menu.java.

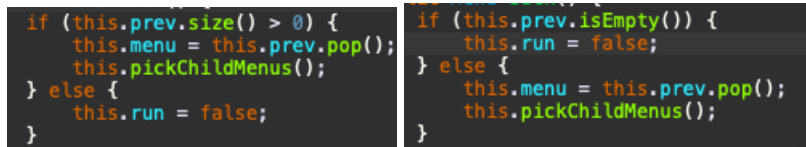
Neste caso num if é utilizado o método `size()` do `ArrayList` e seguidamente verifica-se se o array está vazio. A solução para esta ineficiência passa por invocar o método `isEmpty()` do `ArrayList`.

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

```
if (lR.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;
}

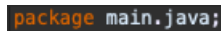
if (this.prev.isEmpty()) {
    this.run = false;
} else {
    this.menu = this.prev.pop();
    this.pickChildMenus();
}

```

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.



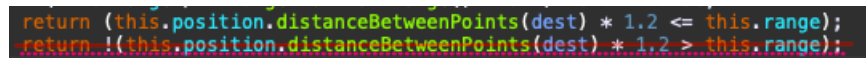
```
package main.java;
```

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 \geq 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.

```
.filter((e)-> e.getType().isEqual(b)) .filter(e-> e.getType().isEqual(b))
```

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.

```
class Rentals implements Serializable {
    static private int id;|
    private static final long serialVersionUID = 1526373866446179937L;
    private final List<Rental> rentalBase;
```

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.

```
private StringBetter reset(){
    return new StringBetter(this.str + "\033[0m");
}
```

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.

```
public StringBetter hideCursor(){
    return new StringBetter(this.str + "\033[725l");
}
```

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 modificado 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.

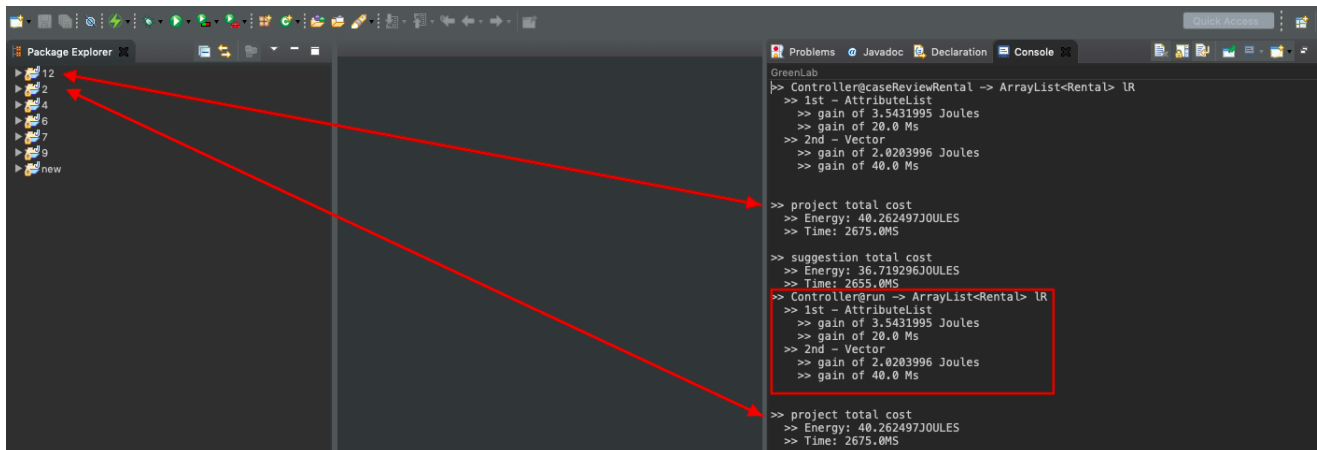


Figura 28. Resultados JStanley pré-correção de redSmells

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.

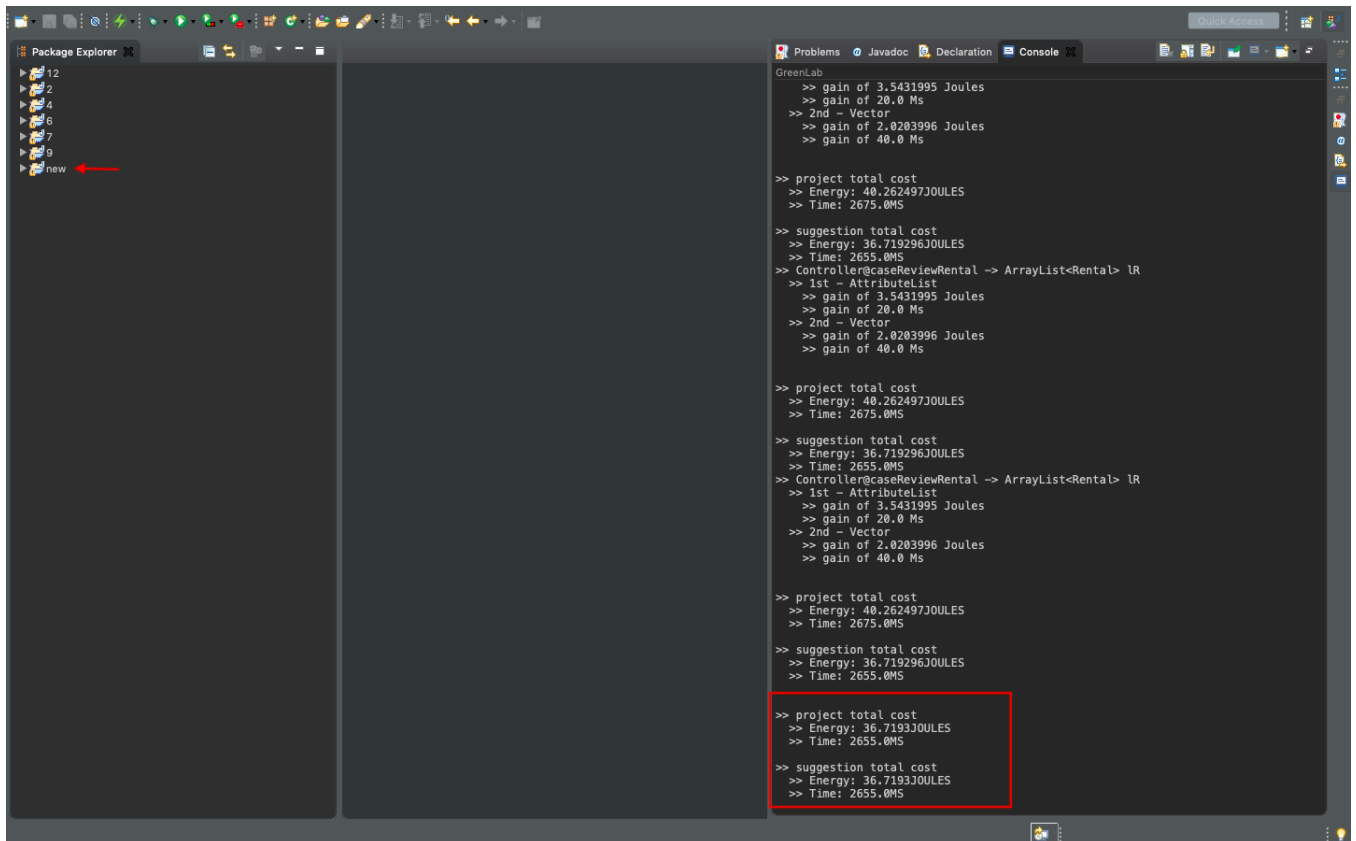


Figura 29. Resultados JStanley pós-correção de redSmells

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

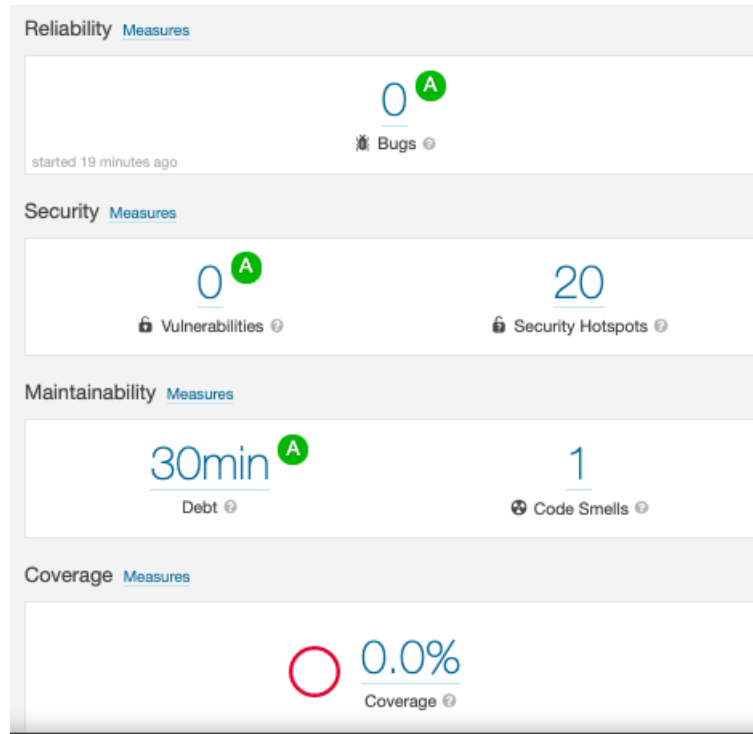


Figura 30. Todo o código foi corrigido com sucesso

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);
    }
}
```

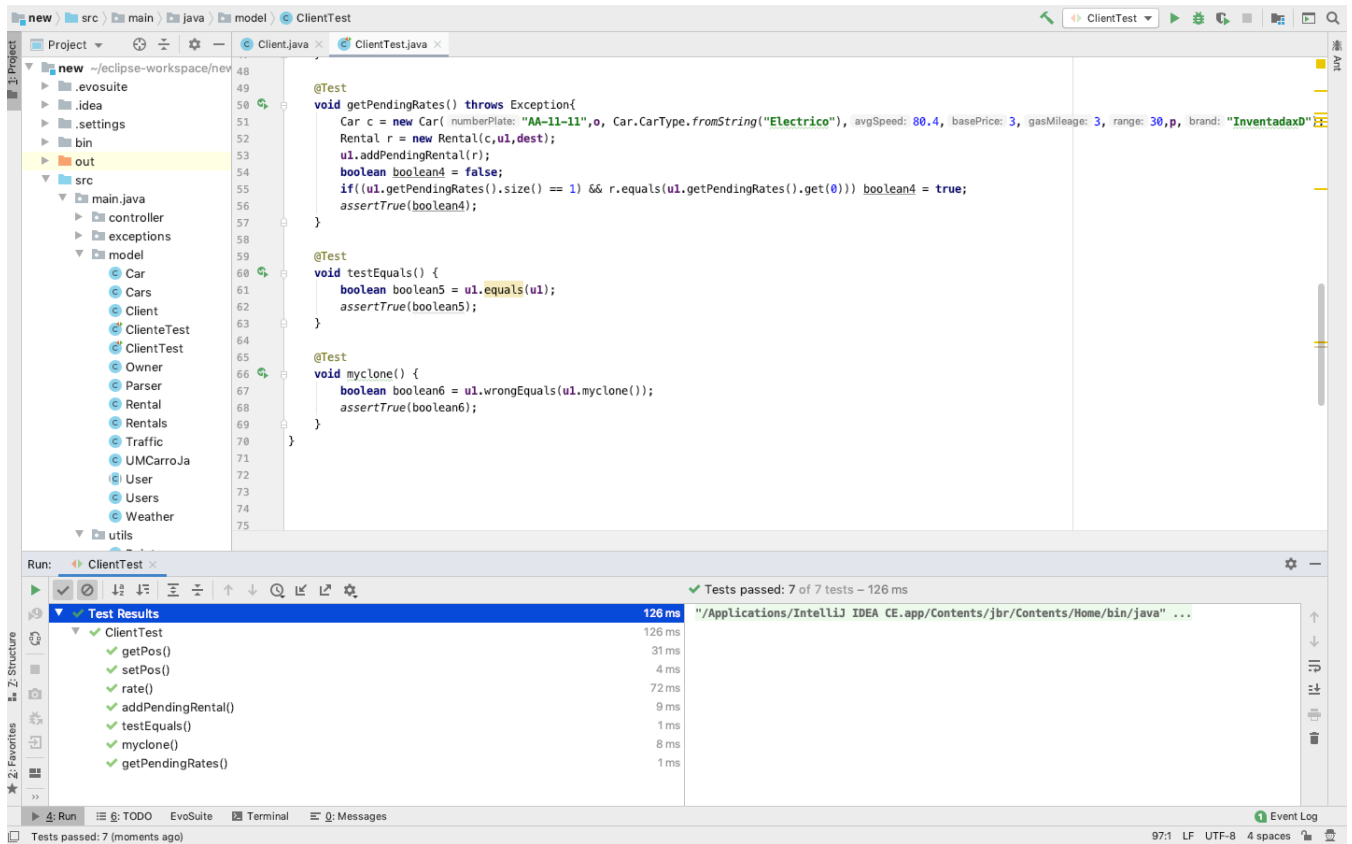
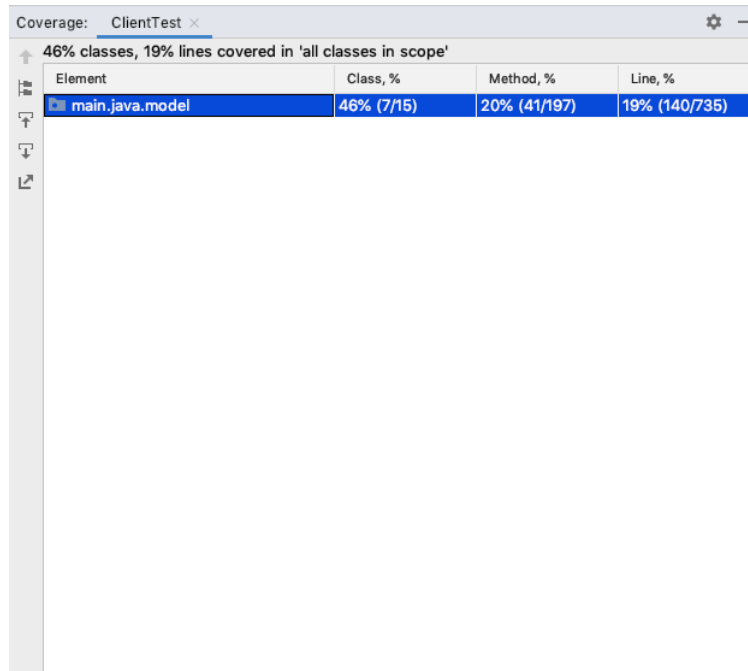


Figura 31. Classe ClientTest.java e resultado da sua aplicação

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.



The screenshot shows the 'Coverage: ClientTest' window in IntelliJ IDEA. It displays a table with test coverage data. The table has four columns: 'Element', 'Class, %', 'Method, %', and 'Line, %'. The first row shows 'main.java.model' with 46% class coverage (7/15), 20% method coverage (41/197), and 19% line coverage (140/735). The window title bar indicates '46% classes, 19% lines covered in 'all classes in scope''.

| Element | Class, % | Method, % | Line, % |
|-----------------|------------|--------------|---------------|
| main.java.model | 46% (7/15) | 20% (41/197) | 19% (140/735) |

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

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.

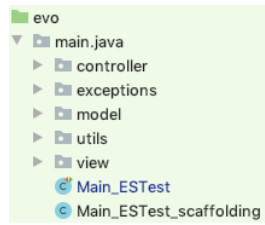


Figura 33. Testes class Main

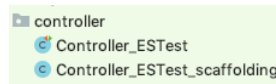


Figura 34. Testes package Controller

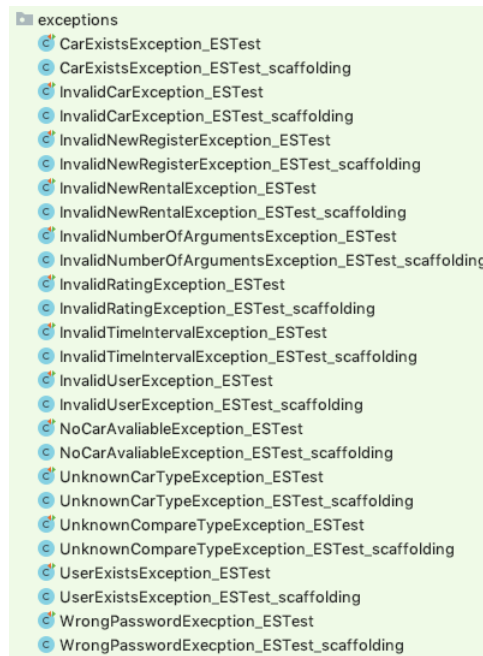


Figura 35. Testes do package exceptions

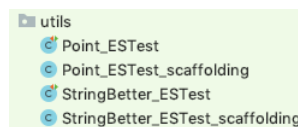


Figura 36. Testes do package utils

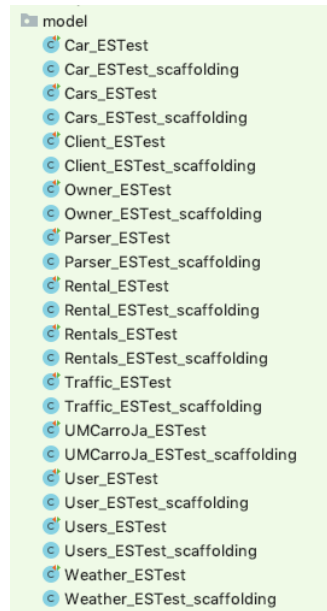


Figura 37. Testes do package model

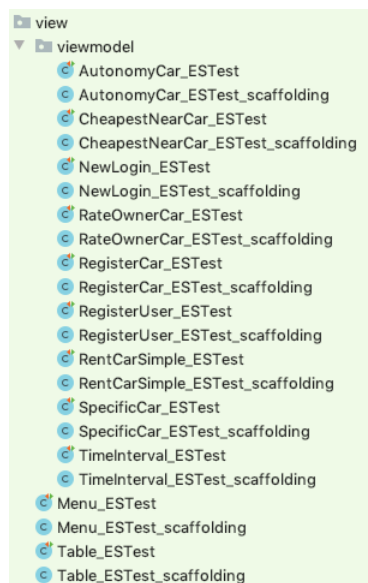
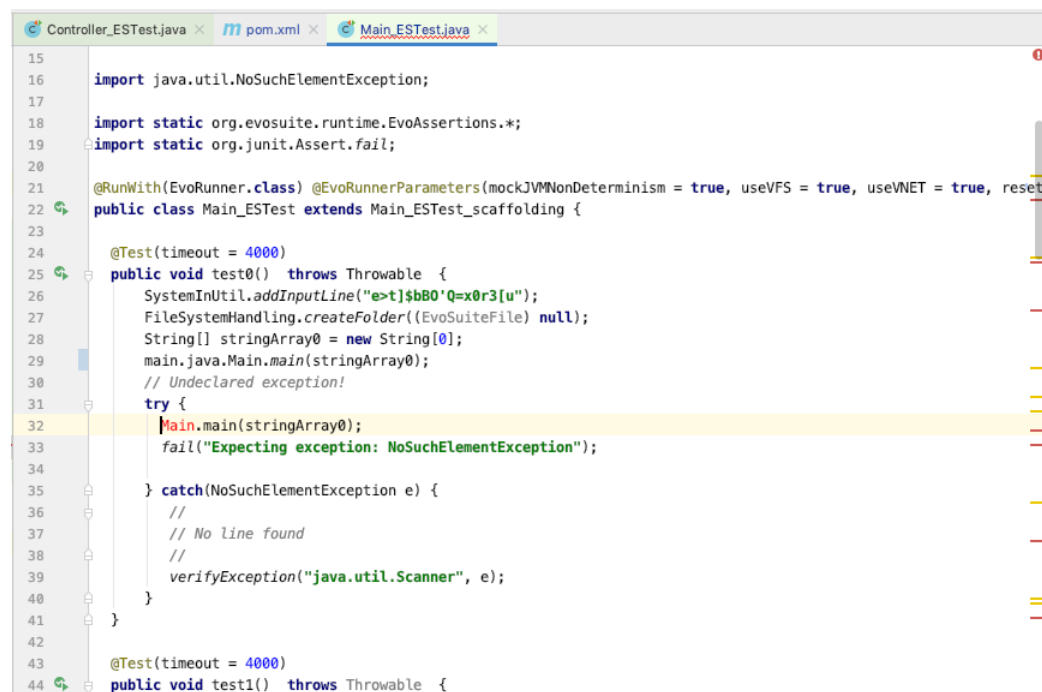


Figura 38. Testes do package view

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 {

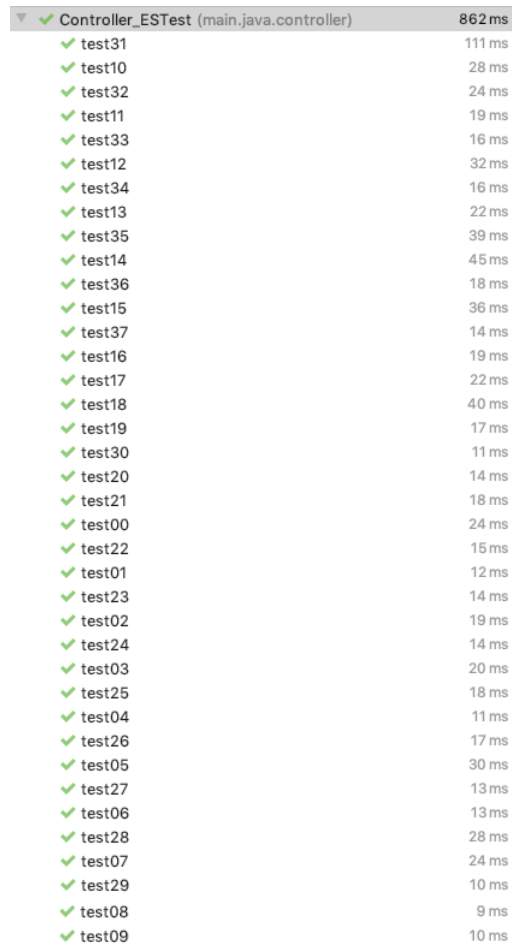
```

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.

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 for the 'Controller_ESTest' class. The class name is at the top with a green checkmark and a total execution time of 862 ms. Below it, a list of 31 individual tests (test00 to test31) are shown, each with a green checkmark and its own execution time in milliseconds. The tests are sorted in descending order of execution time.

| | |
|----------------------------------------------|--------|
| ▼ ✓ 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.

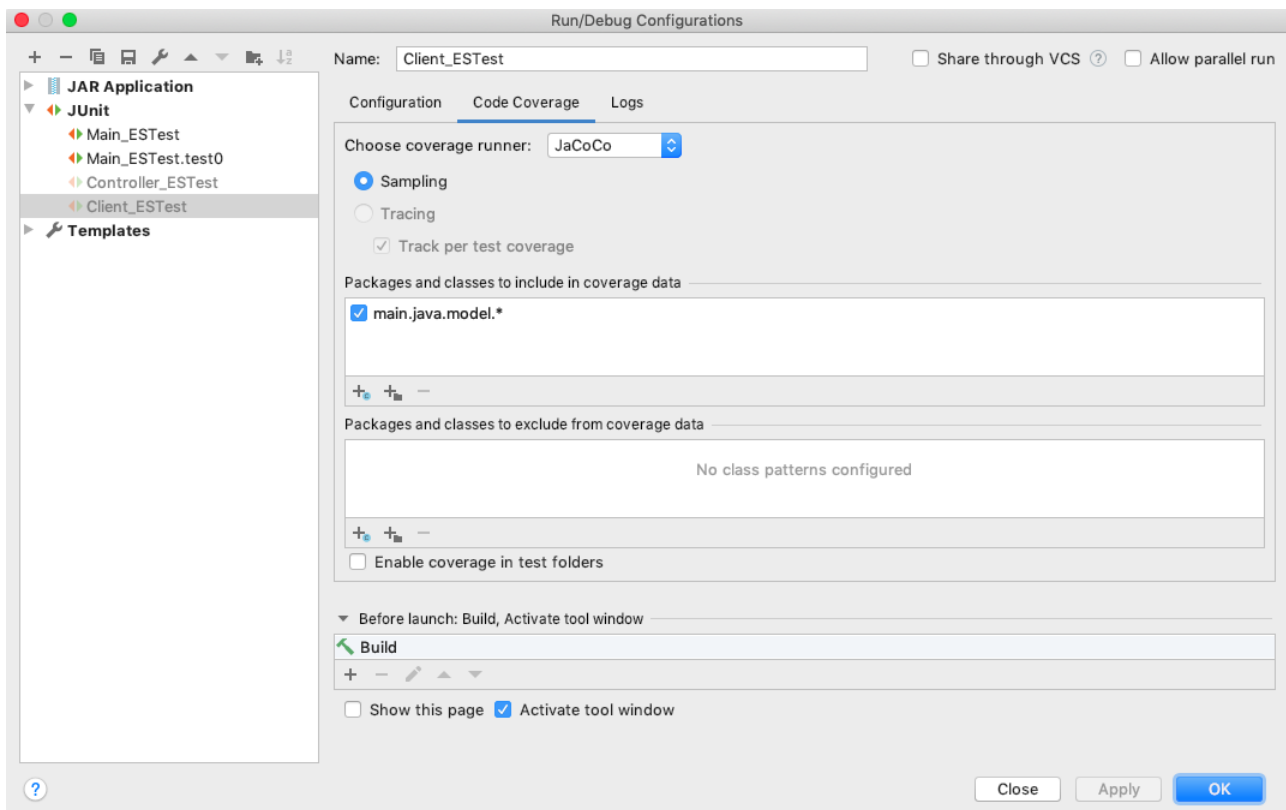


Figura 43. Configuração do Code Coverage para usar o plugin JaCoCo

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.

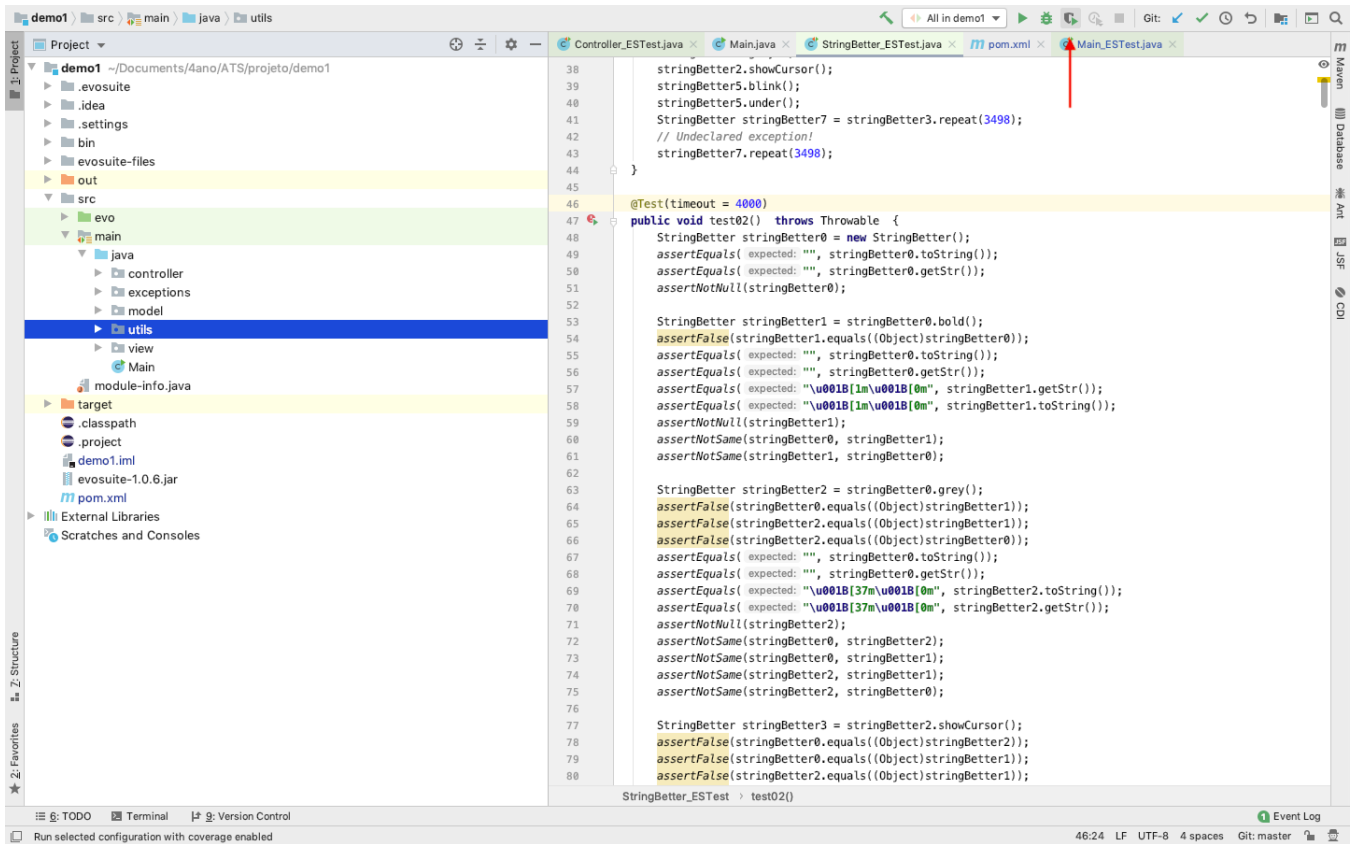


Figura 44. Execução dos testes usando coverage com o plugin JaCoCo

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

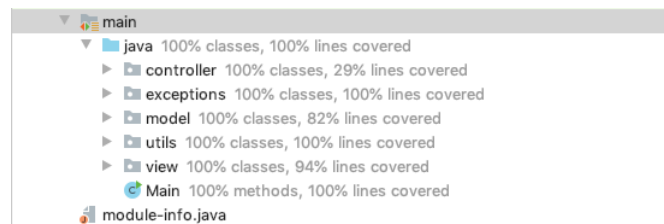


Figura 45. Resultados do coverage utilizando o plugin JaCoCo

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.

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*.

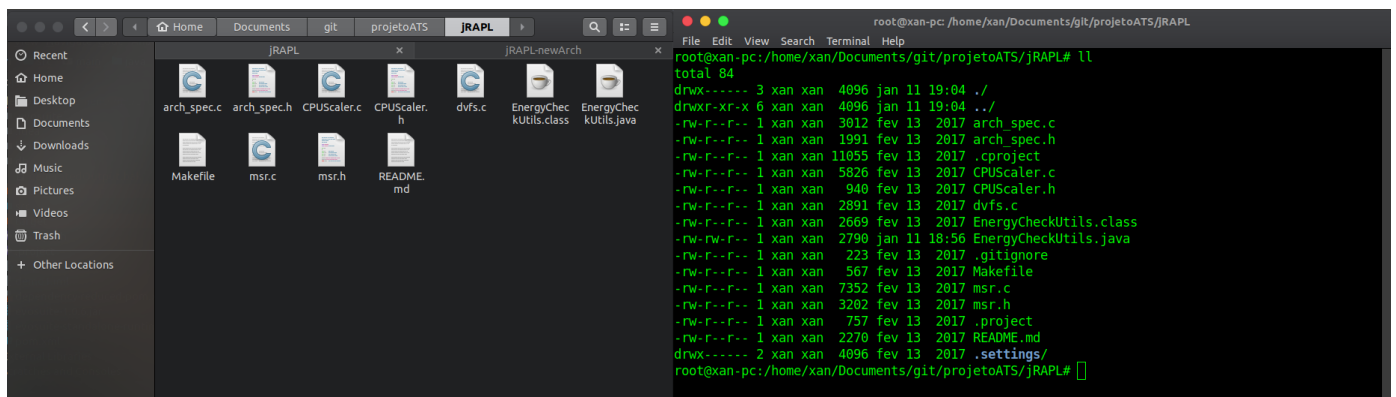
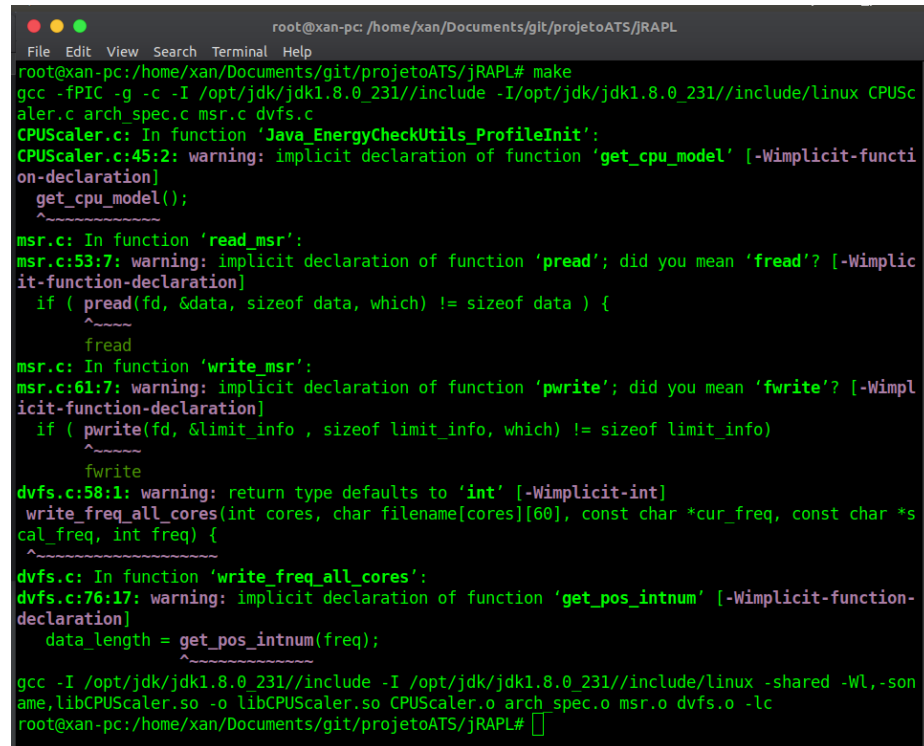


Figura 47. Pasta inicial do jRAPL

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



```

root@xan-pc: /home/xan/Documents/git/projetoATS/jRAPL
File Edit View Search Terminal Help
root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# make
gcc -fPIC -g -c -I /opt/jdk/jdk1.8.0_231//include -I/opt/jdk/jdk1.8.0_231//include/linux CPUScaler.c arch_spec.c msr.c dvfs.c
CPUScaler.c: In function 'Java_EnergyCheckUtils_ProfileInit':
CPUScaler.c:45:2: warning: implicit declaration of function 'get_cpu_model' [-Wimplicit-function-declaration]
    get_cpu_model();
    ^~~~~~
msr.c: In function 'read_msr':
msr.c:53:7: warning: implicit declaration of function 'pread'; did you mean 'fread'? [-Wimplicit-function-declaration]
    if ( pread(fd, &data, sizeof data, which) != sizeof data ) {
        ^~~~~
        fread
msr.c: In function 'write_msr':
msr.c:61:7: warning: implicit declaration of function 'pwrite'; did you mean 'fwrite'? [-Wimplicit-function-declaration]
    if ( pwrite(fd, &limit_info, sizeof limit_info, which) != sizeof limit_info)
        ^~~~~
        fwrite
dvfs.c:58:1: warning: return type defaults to 'int' [-Wimplicit-int]
    write_freq_all_cores(int cores, char filename[cores][60], const char *cur_freq, const char *scal_freq, int freq) {
    ^~~~~~
dvfs.c: In function 'write_freq_all_cores':
dvfs.c:76:17: warning: implicit declaration of function 'get_pos_intnum' [-Wimplicit-function-declaration]
    data_length = get_pos_intnum(freq);
                  ^~~~~~
gcc -I /opt/jdk/jdk1.8.0_231//include -I /opt/jdk/jdk1.8.0_231//include/linux -shared -Wl,-soname,libCPUScaler.so -o libCPUScaler.so CPUScaler.o arch_spec.o msr.o dvfs.o -lc
root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL#

```

Figura 48. jRAPL make

Para além de executar o comando *make* falta ir ao projeto e apagar o ficheiro *.tmp* da pasta para no código no *try/catch*, em vez de entrar no *.tmp* e assim poder executar o programa a partir do ficheiro *generatedOutput.bak*.

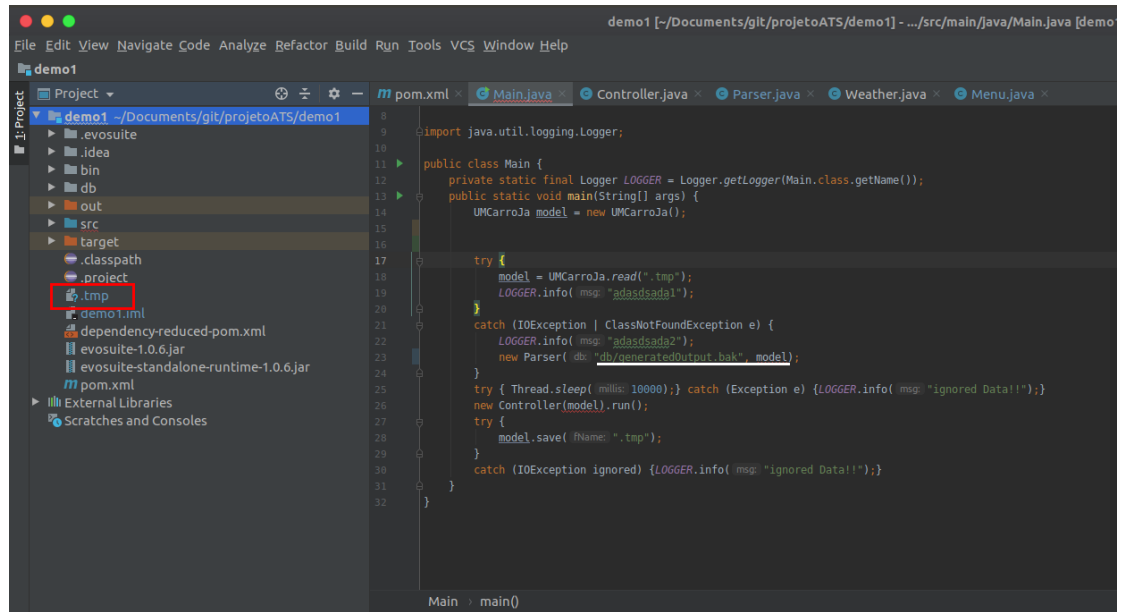


Figura 49. Remover .tmp do programa UmCarroJá

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, ver imagem em baixo.

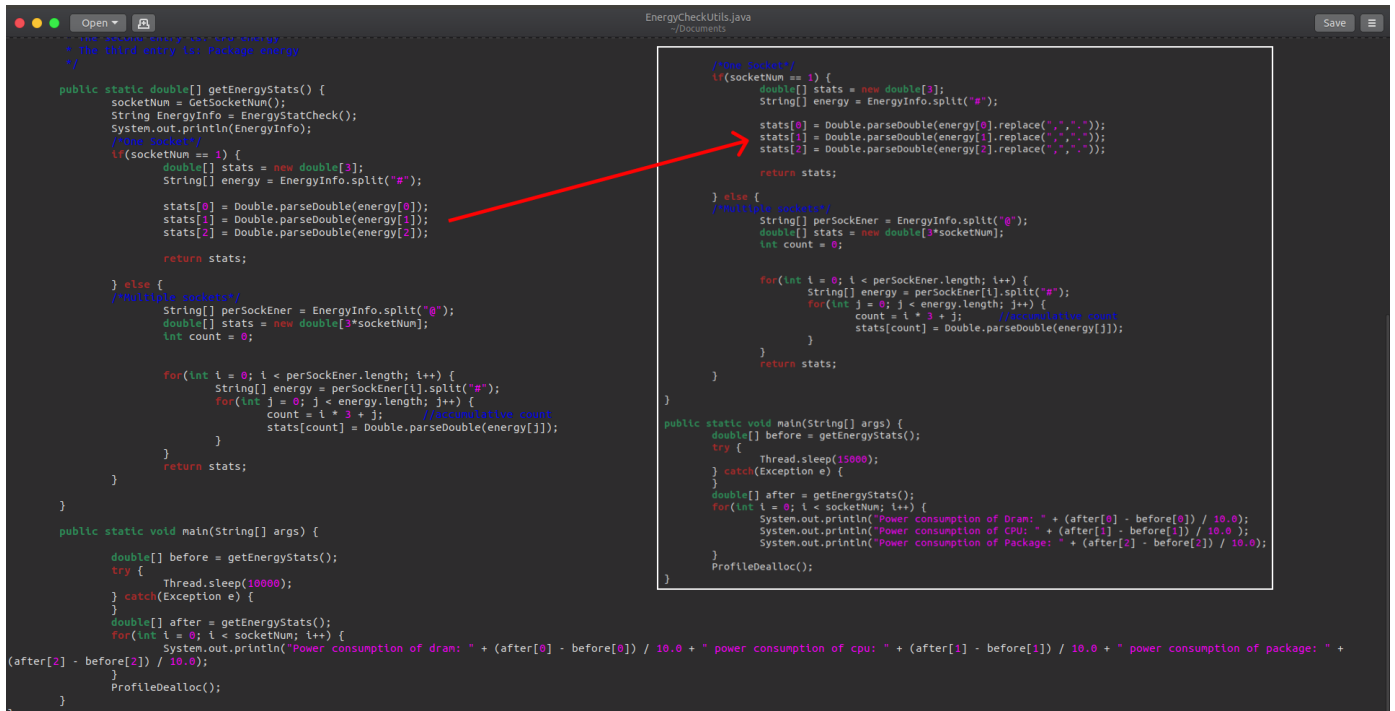


Figura 50. Alterações no ficheiro EnergyCheckUtils

Depois de corrigir o EnergyCheckUtils procedemos à execução dos comando necessário para obter os dados de energia. Comandos, *sudo modprobe msr*, *javac EnergyCheckUtils.java* e *java EnergyCheckUtils*.

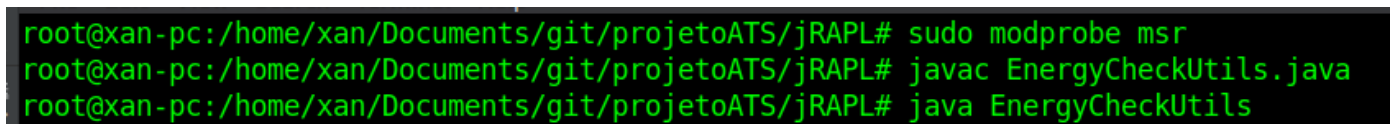


Figura 51. Comandos para correr o RAPL

Para testar o UmCarroJá é primeiro necessário correr o EnergyCheckUtils e depois correr o UmCarroJá e assim obter os dados de energia tendo em consideração a solução UmCarroJá.

Efectuamos quatro testes e conseguimos os seguintes resultados:

```
root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
Power consumption of Dram: 2.9056579000000006
Power consumption of CPU: 2.30901490000000566
Power consumption of Package: 5.777264400000059
root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
Power consumption of Dram: 2.798205600000074
Power consumption of CPU: 3.522912600000018
Power consumption of Package: 6.285589599999973
root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
Power consumption of Dram: 2.7814086999998837
Power consumption of CPU: 2.3450072999999976
Power consumption of Package: 5.53177489999998
root@xan-pc:/home/xan/Documents/git/projetoATS/jRAPL# java EnergyCheckUtils
Power consumption of Dram: 2.98714600000003
Power consumption of CPU: 3.8177185000000464
Power consumption of Package: 7.014196699999957
```

Figura 52. Testes elaborados com RAPL à solução UmCarroJá

Referências