
Smart Testing for Smart Houses



A95437
Beatriz Monteiro



A95835
Bianca Vale



A97763
Daniel Du

Conteúdo

1	Introdução	2
1.1	Contextualização	2
2	Funcionalidades a Desenvolver	2
2.1	Unit Testing	2
2.2	Mutation Testing	3
2.3	Unit Test Generation	5
2.4	Análise da cobertura e da qualidade do código	5
2.5	Hypothesis	9
3	Dificuldades	11
4	Manual de utilização	12
5	Conclusão e Análise dos Resultados	12
A	Anexos	13
A.1	JUnit no <i>pom.xml</i>	13
A.2	PITEST no ficheiro pom.xml	13
A.3	EvoSuite no ficheiro pom.xml	14
A.4	Jacoco no ficheiro pom.xml	14

Lista de Figuras

1	Relatório gerado pelo PIT página inicial	3
2	Relatório gerado pelo PIT cobertura por classes	4
3	Relatório gerado pelo PIT cobertura detalhada	4
4	Relatório gerado pelo PIT antes dos testes do evosuite	5
5	Relatório gerado pelo Jacoco antes dos testes do evosuite	6
6	Gráfico de comparação do número de mutações com e sem println	6
7	Gráfico de comparação da percentagem de mutações mortas com e sem println sem testes evosuite	7
8	Relatório gerado pelo PIT depois dos testes do evosuite	7
9	Relatório gerado pelo Jacoco depois dos testes do evosuite	8
10	Gráfico de comparação da percentagem de mutações mortas com e sem println com testes evosuite	8
11	Tempo para gerar e testar com máquina ligada ou não à corrente elétrica	9
12	PIT não reconhece testes escritos	11
13	Contraste de cobertura com o Jacoco	11

Listings

1	Exemplo de um teste unitário	2
---	--	---

1 Introdução

Este projeto foi desenvolvido no âmbito da Unidade Curricular de Análise e Testes de Software no ano letivo 2022/23. O presente relatório visa explicar a nossa solução e todo o processo inerente à formulação da resposta ao problema em questão. Para este trabalho prático, foi-nos proposto desenvolver uma pilha de testes ao trabalho prático da Unidade Curricular de Programação Orientada aos Objetos (POO) do ano letivo 2021/2022.

1.1 Contextualização

Primeiramente, de forma a produzir melhores testes precisamos de perceber em que ambiente é que estamos a trabalhar. E desta forma vamos, de forma breve, contextualizar o trabalho prático de POO.

O trabalho prático de POO baseia-se na criação de um sistema que monitoriza e regista a informação do consumo energético de casas inteligentes, que por sua vez é constituída por um conjunto de *Smart Devices*, tais como *Smart Speakers*, *Smart Camera* e por último *Smart Bulbs*, em cada divisão da casa.

Passando para a pilha de testes em si, esta é constituída por testes escritos pelo grupo com base na visualização do código fonte e face a mutações geradas pela ferramenta *PIT*. Para além desses testes escritos pelo grupo, a nossa pilha de testes também é constituída por testes gerados pela ferramenta *EvoSuite*.

Para além do que foi referido, propuseram-nos utilizar alguma das ferramentas lecionadas nas aulas para gerar automaticamente o ficheiro de *logs* que é passado como *input* no programa de *SmartHouses*. Para esta tarefa optamos por recorrer à ferramenta *Hypothesis*.

2 Funcionalidades a Desenvolver

Nesta secção iremos apresentar e explicar o raciocínio por de trás de todas as funcionalidades desenvolvidas pelo grupo de forma a termos a pilha de testes que possuímos.

2.1 Unit Testing

Nesta secção iremos abordar o Unit Testing, a sua importância no nosso trabalho e de que modo é que foi utilizado no nosso projeto.

Passando para a definição do Unit Testing, o Unit Testing é um processo de desenvolvimento de software que testa partes individuais do código. Esta prática é importante uma vez que assegura a correção e funcionalidade do código de um projeto. E uma vez que estamos a criar testes de forma a garantir a correção e funcionalidade do código, aplicar e utilizar esta ferramenta no projeto torna-se fulcral.

A nossa estratégia para o desenvolvimento de testes unitários foi com base na **White Box Testing**, uma vez que sabíamos como o *source code* foi implementado e, para além disso, à medida que íamos fazendo os testes víamos código em si.

Nos anexos está demonstrada a forma como o JUnit foi configurado para o nosso projeto e abaixo fica demonstrado um teste desenvolvido pelo grupo utilizando esta ferramenta.

```
1 @Test
2 public void testSetTone(){
3     int quente = smartBulb.WARM;
4     int neutro = smartBulb.NEUTRAL;
5     int frio = smartBulb.COLD;
6     assertEquals(neutro, smartBulb.getTone(), "Valor da tonalidade da
    smartBulb n o o esperado");
```

```

7
8     smartBulb1.setTone(frio-1);
9     assertEquals(frio,smartBulb1.getTone(), "Valor da tonalidade da
smartBulb1 n o      o esperado");
10
11     smartBulb2.setTone(frio+1);
12     assertEquals(neutro,smartBulb2.getTone(), "Valor da tonalidade da
smartBulb2 n o      o esperado");
13
14     smartBulb3.setTone(quente+1);
15     assertEquals(quente,smartBulb3.getTone(), "Valor da tonalidade da
smartBulb3 n o      o esperado");
16
17     smartBulb4.setTone(quente-1);
18     assertEquals(neutro,smartBulb4.getTone(), "Valor da tonalidade da
smartBulb4 n o      o esperado");
19 }

```

Listing 1: Exemplo de um teste unitário

De forma a testarmos os nossos testes unitários procedemos a dois execuções diferentes, isto é, ou corríamos no próprio *IntelliJ* que tornava o *workflow* mais rápido pois não precisávamos de trocar de janela para testar os testes, ou então corríamos via terminal com o primeiro comando na secção 4.

2.2 Mutation Testing

Mutation Testing consistem em fazer mutações, isto é, pequenas alterações no código e, verificar se os nossos testes conseguem detetar as mutações que introduzimos.

O *PIT* foi a ferramenta deveras mais difícil de configurar, contudo foi uma ferramenta extremamente útil que nos permitiu analisar melhor a cobertura e qualidade dos nossos testes.

O nosso grande objetivo ao utilizar esta ferramenta foi analisar se os nossos testes previamente escritos estavam suficientemente bons em termos de cobertura e se matavam as mutações geradas pelo pit. Caso não matassem, escrever melhores ou mais teste capazes de matar as mutações geradas.

Nos anexos mostramos como é que configuramos o pit no nosso projeto e nas figuras abaixo um exemplo de relatório que o pit gera sobre as mutações geradas e mortas pelos nossos testes.

Nesta figura 1 mostra a página inicial do relatório que o PIT gera.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
13	77% 1142/1487	58% 519/889	82% 519/633

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
smarthouse	13	77% 1142/1487	58% 519/889	82% 519/633

Report generated by [PIT](#) 1.8.0

Figura 1: Relatório gerado pelo PIT página inicial

Nesta figura 2 mostra a página do relatório aonde mostra em mais detalhe por cada classe a cobertura dos testes, as mutações mortas e a força dos testes.

Pit Test Coverage Report

Package Summary

smarthouse

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
13	77% 1142/1487	58% 519/889	82% 519/633

Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
CasaInteligente.java	91% 173/190	41% 39/94	48% 39/81
Comercializador.java	96% 107/111	47% 22/47	51% 22/43
Estatisticas.java	66% 73/110	62% 51/82	100% 51/51
Fatura.java	100% 29/29	100% 6/6	100% 6/6
Formula.java	80% 12/15	57% 16/28	80% 16/20
Parser.java	36% 37/102	28% 14/50	100% 14/14
Periodo.java	100% 35/35	100% 29/29	100% 29/29
Programa.java	38% 39/104	36% 27/74	100% 27/27
Simulador.java	72% 337/470	67% 220/329	100% 220/220
SmartBulb.java	91% 63/69	48% 16/33	53% 16/30
SmartCamera.java	93% 68/73	69% 18/26	75% 18/24
SmartDevice.java	92% 79/86	58% 23/40	59% 23/39
SmartSpeaker.java	97% 90/93	75% 38/51	78% 38/49

Report generated by [PIT](#) 1.8.0

Figura 2: Relatório gerado pelo PIT cobertura por classes

Na figura 3 mostra já em que linhas de cada classe que os nossos testes cobrem e em que linhas as mutações foram geradas. Caso a linha esteja sublinhada com um verde mais carregado significa que os nossos testes foram capazes de matar a mutação, caso esteja vermelha significa que os nossos testes não foram capazes de matar as mutações.

```
37     public SmartBulb(Modo modo, int tone, double tamanho, double consumoDiario) {
38         super(150, modo);
39         this.tone = tone;
40         this.tamanho = tamanho;
41         setConsumoDiario(consumoDiario);
42     }
43
44     public SmartBulb(SmartBulb s) {
45         super(s);
46         this.tone = s.tone;
47         this.tamanho = s.tamanho;
48     }
49
50
51
52     public void setTone(int t) {
53         if (t>WARM) this.tone = WARM;
54         else if (t<COLD) this.tone = COLD;
55         else this.tone = t;
56     }
57
58     public int getTone() {
59         return this.tone;
60     }
61
62     public double getTamanho() {
63         return tamanho;
64     }
65
66     public void setTamanho(double tamanho) {
67         this.tamanho = tamanho;
68     }
69
70     public static SmartBulb criarSmartBulb(Scanner scanner) {
71         System.out.println("\n");
72         System.out.println("+-----+");
73         System.out.println("| -> Escreve no formato Modo,Tone,Tamanho |");
74         System.out.println("| -> Exemplo: OFF,WARM,6.3 |");
```

Figura 3: Relatório gerado pelo PIT cobertura detalhada

De formas a gerar as mutações e testar se os nossos testes matam as mutações corremos o quarto comando na secção 4.

2.3 Unit Test Generation

Esta ferramenta tornou-se fulcral mais no final do desenvolvimento de testes para a nossa pilha de testes, pois primeiramente haviam métodos para quais nós não sabíamos como escrever os testes e com a ajuda do evosuite conseguimos ter testes na nossa pilha de testes que cobrissem esses métodos e matassem possíveis mutações geradas pelo pit nesses métodos. Para além disso, esta ferramenta tornou-se muito útil porque nos permitiu poupar muito tempo e trabalho pois em pouco tempo o evosuite conseguiu gerar muitos e bons testes.

2.4 Análise da cobertura e da qualidade do código

Para as nossas análises tivemos muito em conta os relatórios gerados pelo *Jacoco* e pelo *PIT*.

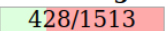
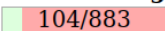
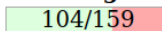
Para gerar o relatório pelo Jacoco corremos o oitavo comando na secção 4.

Para a nossa primeira análise vamos fazer uma comparação da cobertura de testes antes e depois de termos os testes gerados pelo evosuite.

Pit Test Coverage Report

Package Summary

smarthouse

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
13	28%  428/1513	12%  104/883	65%  104/159

Breakdown by Class

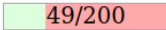
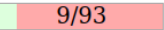
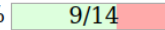
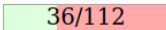
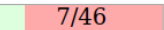

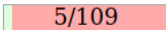
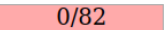
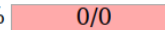
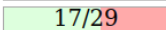
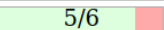
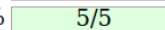
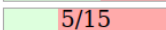
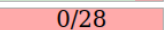
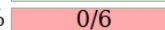
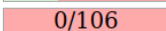
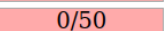
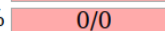
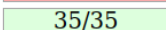
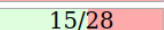
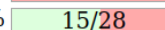
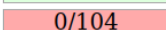
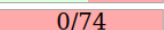
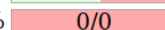
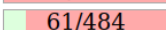
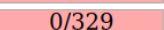
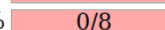
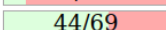
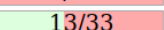
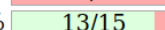
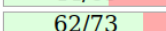
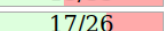
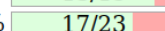
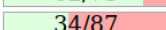
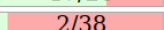
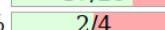
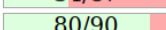
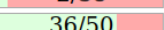
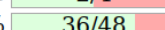
Name	Line Coverage	Mutation Coverage	Test Strength
CasaInteligente.java	25%  49/200	10%  9/93	64%  9/14
Comercializador.java	32%  36/112	15%  7/46	88%  7/8
Estatisticas.java	5%  5/109	0%  0/82	0%  0/0
Fatura.java	59%  17/29	83%  5/6	100%  5/5
Formula.java	33%  5/15	0%  0/28	0%  0/6
Parser.java	0%  0/106	0%  0/50	0%  0/0
Periodo.java	100%  35/35	54%  15/28	54%  15/28
Programa.java	0%  0/104	0%  0/74	0%  0/0
Simulador.java	13%  61/484	0%  0/329	0%  0/8
SmartBulb.java	64%  44/69	39%  13/33	87%  13/15
SmartCamera.java	85%  62/73	65%  17/26	74%  17/23
SmartDevice.java	39%  34/87	5%  2/38	50%  2/4
SmartSpeaker.java	89%  80/90	72%  36/50	75%  36/48

Figura 4: Relatório gerado pelo PIT antes dos testes do evosuite

P00_Project



Element	Missed Instructions	Cov.
 main.java		27%
Total	4,258 of 5,860	27%

Figura 5: Relatório gerado pelo Jacoco antes dos testes do evosuite

Pelas figuras acima conseguimos ver que a nossa pilha de testes antes de exportar os testes gerados pelo evosuite não cobria muitas linhas de código, métodos nem matava muitas mutações. Porém grande parte das linhas de código eram *println* e das mutações geradas eram sobre as mesmas linhas de *println*. Para sermos mais concretos, 376 mutações geradas foram em linhas de *println* pelo que o grupo considerou que não fazia sentido fazer testes de forma a cobrir ou matar as mutações nessas linhas.

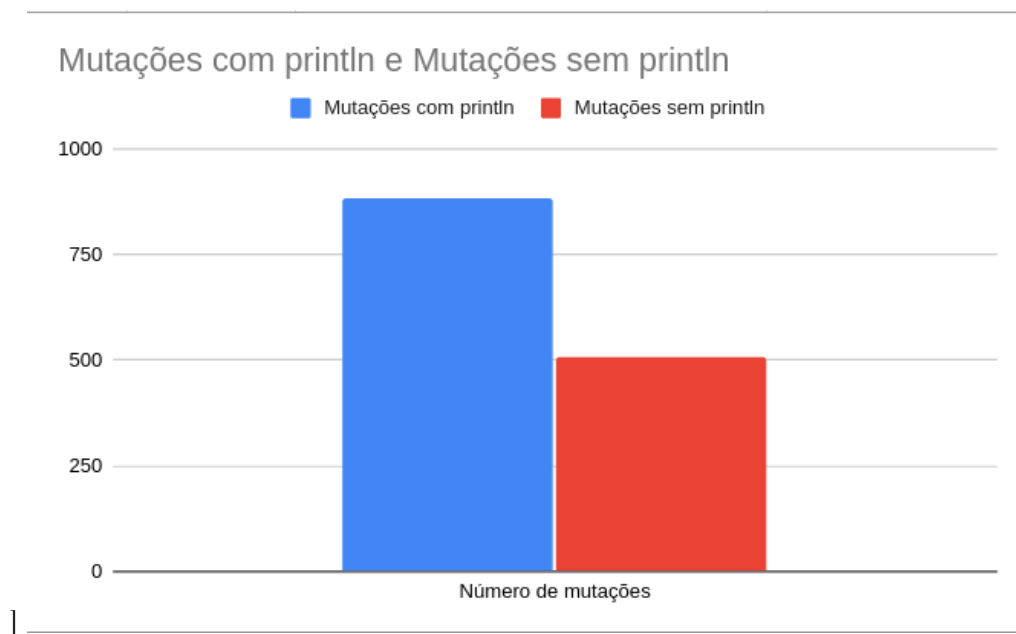


Figura 6: Gráfico de comparação do número de mutações com e sem println

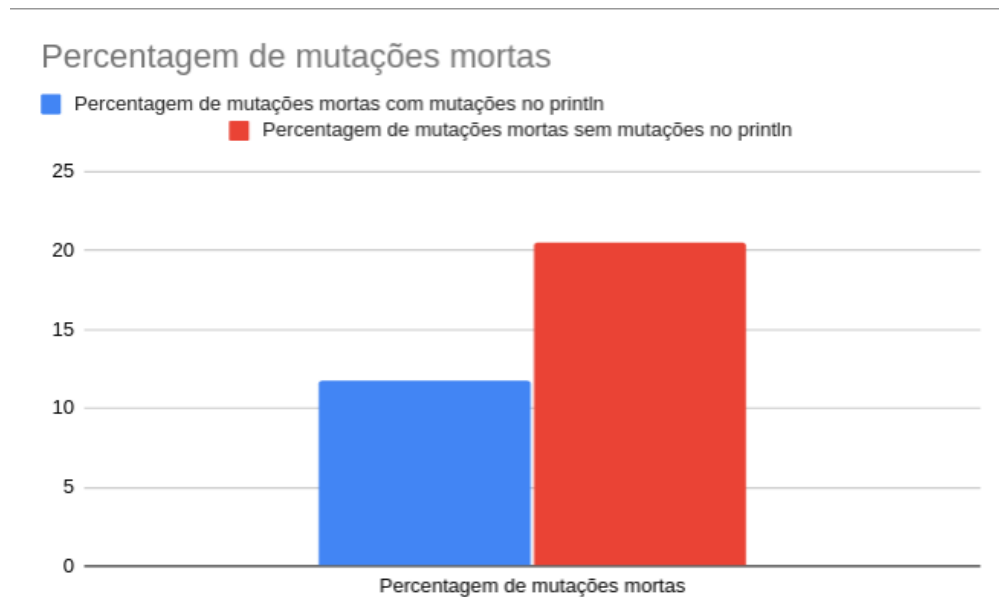


Figura 7: Gráfico de comparação da percentagem de mutações mortas com e sem println sem testes evosuite

Pela figura acima conseguimos analisar que se não houvesse mutações nas linhas de println, no total teríamos cerca de 500 mutações e a percentagem de mutações mortas aumentava cerca de 8%.

Pit Test Coverage Report

Package Summary

smarthouse

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
13	78% 1167/1487	39% 351/889	53% 351/658

Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
CasaInteligente.java	93% 177/190	45% 42/94	51% 42/82
Comercializador.java	96% 107/111	47% 22/47	51% 22/43
Estatisticas.java	67% 74/110	63% 52/82	100% 52/52
Fatura.java	100% 29/29	100% 6/6	100% 6/6
Formula.java	87% 13/15	61% 17/28	71% 17/24
Parser.java	36% 37/102	28% 14/50	100% 14/14
Periodo.java	100% 35/35	100% 29/29	100% 29/29
Programa.java	38% 39/104	36% 27/74	100% 27/27
Simulador.java	76% 356/470	14% 45/329	19% 45/239
SmartBulb.java	91% 63/69	48% 16/33	53% 16/30
SmartCamera.java	93% 68/73	69% 18/26	75% 18/24
SmartDevice.java	92% 79/86	63% 25/40	64% 25/39
SmartSpeaker.java	97% 90/93	75% 38/51	78% 38/49

Figura 8: Relatório gerado pelo PIT depois dos testes do evosuite

POO_Project



Element	Missed Instructions	Cov.
 smarthouse		75%
Total	1,444 of 5,845	75%

Figura 9: Relatório gerado pelo Jacoco depois dos testes do evosuite

Da mesma forma que na nossa pilha de teste sem os testes gerados pelo evosuite os testes gerados pelo evosuite também não foram direcionados a matar as mutações geradas nas linhas de `println`. Desta forma conseguimos observar que a percentagem de mutações mortas aumentou em cerca de 30%.

Percentagem de mutações mortas

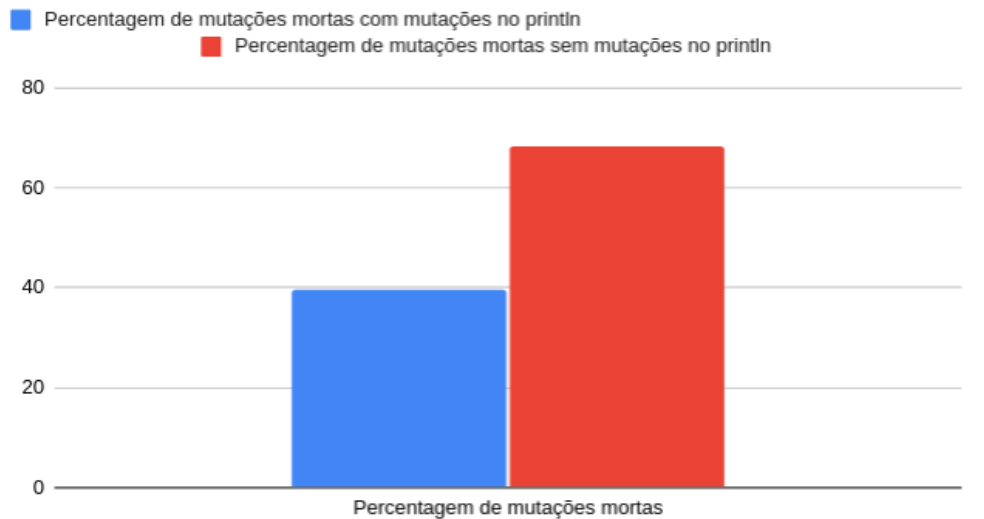


Figura 10: Gráfico de comparação da percentagem de mutações mortas com e sem `println` com testes evosuite

É de notar que a percentagem de linhas cobertas na classe programa podiam ser mais pois ao testar os testes com o pit dá erro apesar de testarmos somente com o maven os testes passam. E dessa forma tivemos que descartar cerca de 10 testes direcionados à classe `programa.java`.

Outras análises feitas pelo grupo foi o tempo gasto para gerar os testes do evosuite e o tempo levado para gerar as mutações e correr os testes na máquina ligada ou não à corrente elétrica.

Tempo para gerar e testar com máquina ligada ou não à corrente elétrica

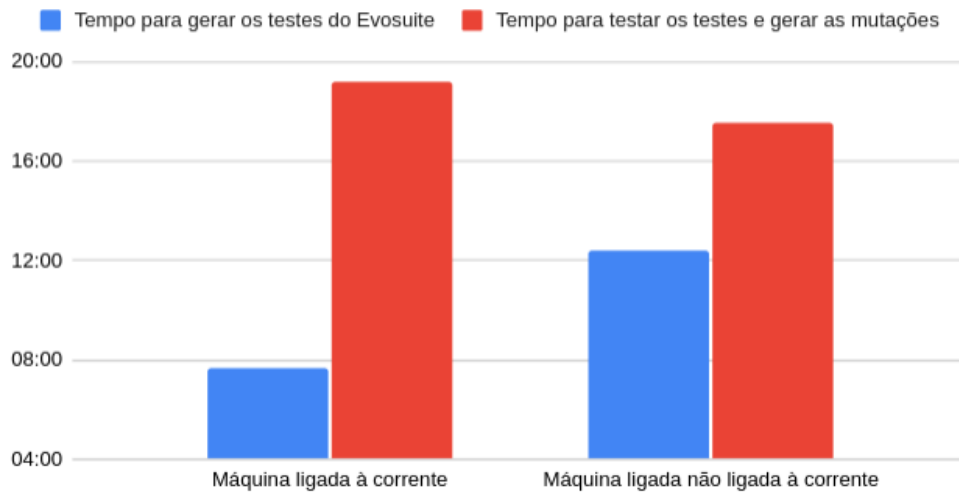


Figura 11: Tempo para gerar e testar com máquina ligada ou não à corrente elétrica

É notório que o tempo gasto com a máquina ligada à corrente é muito menor. Contudo, é interessante de ver que o tempo gasto para testar os testes e gerar as mutações com a máquina não ligada à corrente foi menor uma vez que a máquina estava somente a executar aquele processo, enquanto que mesmo estando a máquina ligada à corrente o tempo que levou a gerar foi ligeiramente superior uma vez que a máquina estava a gastar mais recursos, como por exemplo páginas no browser abertas, bluetooth ligado entre outras.

2.5 Hypothesis

Apesar desta ferramenta ser bastante útil na geração automática de testes, neste projeto recorremos ao *hypothesis* para gerar o ficheiro *logs.txt* que a aplicação de *SmartHouses* recebe como *input*.

Após analisarmos o ficheiros de *logs*, apercebemo-nos que o ficheiro teria de seguir o seguinte formato:

Casa: Proprietario,NIF,Fornecedor

Divisão: Nome

Tipo de Equipamento {

SmartBulb: Type,Tamanho,Consumo

SmartCamera: Resolução,TamanhoFicheiro,Consumo

SmartSpeaker: Volume,Radio,Marca,Consumo

}

Inicialmente criamos duas classes (Casa e Divisão), uma superclasse (Equipamentos) e três subclasses(*SmartBulb*,*SmartSpeaked* e *SmartCamera*). Cada classe tem os parâmetros que se encontram acima. De realçar que dentro de cada casa temos uma lista de divisões que a compõe e, por sua vez, cada divisão tem uma lista de Equipamentos.

Antes de passar para a geração da casa e dos seus componente começamos por definir o número de casas que serão geradas, este número é, também, gerado aleatoriamente

e varia entre 10 e 100. Estes números foram escolhidos para que fosse possível testar e por em prática a geração automática do ficheiro input, apesar de não serem valores "realistas".

Geração dos diferentes componentes:

SmartSpeaker:

gerar__consumo gera um double com 3 casas decimais

gerar__Volume gera um inteiro

gerar__Marca escolhe uma marca de uma lista definida previamente de marcas de Colunas

gerar__Radio escolhe uma rádio de uma lista de rádios nacionais, sendo que nesta lista foram escolhidas as rádios portuguesas mais populares

gerar__SmartSpeaker utiliza todos os geradores acima para gerar uma SmartSpeaker

SmartCamera:

gerar__consumo

gerar__Resolucao escolhe uma resolução de uma lista de resoluções criada previamente

gerar__TamanhoFicheiro gera um inteiro em 0 e 150

gerar__SmartCamera utiliza todos os geradores acima para gerar uma SmartCamera

SmartBulb:

gerar__consumo

gerar__Tamanho gera um inteiro entre 5 e 20

gerar__Type gera um tipo de lâmpada da lista ["Warm", "Neutral", "Cold"]

gerar__SmartBulb utiliza todos os geradores acima para gerar uma SmartBulb

Divisao:

gerar__nomeDivisao escolhe um nome da seguinte lista de divisões ["Quarto", "Sala de Estar", "Sala de Jantar", "Entrada", "Garagem", "Escritorio", "Casa de Banho", "Jardim", "Cozinha", "Cave"]

gerar__Equipamentos gera uma lista de Equipamentos, o número de elementos da lista também é gerado aleatoriamente e varia entre 2 e 10. Recorre às funções **gerar__SmartBulb**, **gerar__SmartCamera** e **gerar__SmartSpeaker** para gerar os diferentes equipamentos.

gerar__Divisao utiliza os 2 geradores acima para gerar uma Divisão

Casa:

gerar__Fornecedor escolhe, aleatoriamente, o nome de uma empresa de eletricidade da lista fornecida. Nesta lista estão apenas empresas portuguesas.

gerar__NIF gera um número de 9 dígitos

gerar__Divisoies gera uma lista de divisões, o número de divisões é gerado aleatoriamente. Utiliza a função **gerar__Divisao** para gerar as divisões individualmente

gerar_Proprietario gera um tuplo com, na posição 0, a lista com os nomes próprios (pode variar entre 1 e 2 elementos); a posição 1 tem a lista dos apelidos sendo que o número de apelidos varia entre 2 e 5. A lista de onde foram escolhidos os nomes próprios tem 50 nomes masculinos e 50 nomes femininos portugueses misturados e a dos apelidos tem 100 apelidos portugueses.

gerar_Casa utiliza os geradores acima para gerar uma Casa

Primeiro abre-se o ficheiro logs.txt que se encontra na pasta "presets_txt" e, após se gerar as casas todas, estas são escritas para o ficheiro.

É, ainda, apresentado um output no terminal a dizer quantas casas foram geradas.

Como executar o ficheiro:

Para executar o ficheiro é necessário, estando na diretoria "POO_Project" executar o seguinte comando no terminal:

```
python3 logs.py
```

3 Dificuldades

De todas as dificuldades que enfrentamos durante o desenvolvimento deste projeto, a maior dificuldade e a que nos roubou mais tempo foi definitivamente a configuração do projeto em si, isto é, conceber um ficheiro *pom.xml* capaz de suportar todas as ferramentas, sem dar erro ao compilar devido à incompatibilidade das versões.

Outra das dificuldades que tivemos foi a configuração do *PIT*, quer seja no ficheiro *pom*, quer seja no *IntelliJ*. Primeiramente, quando tentamos correr o *pit* com o ficheiro *pom* ele não reconhecia os testes já feitos e não era capaz de matar as mutações geradas, nem sequer cobria qualquer linha do código apesar de ao analisarmos com o *Jacoco* verificarmos que tínhamos alguma cobertura, como podemos aferir através das figuras abaixo apresentadas.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
13	0% 0/1510	0% 0/894

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
src.main.java	13	0% 0/1510	0% 0/894

Figura 12: PIT não reconhece testes escritos

POO_Project

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
src.main.java	<div><div></div></div>	16%	<div><div></div></div>	6%	387 461	1,223 1,513	177 245	5 15
Total	4,865 of 5,853	16%	401 of 427	6%	387 461	1,223 1,513	177 245	5 15

Created with [JaCoCo](#) 0.8.5.201910111838

Figura 13: Contraste de cobertura com o Jacoco

Outro erro que levou a ser considerada dificuldade e que nos fez perder muito tempo foi o facto de termos "confiado" no *IntelliJ*, isto é, ao estarmos a trabalhar segundo as

configurações do *IntelliJ* tínhamos a estrutura dos *packages* errada levando a não conseguirmos correr por exemplo o *PIT* e o *Maven* nem através da linha de comandos nem pelo *IntelliJ*.

4 Manual de utilização

Nesta secção iremos mostrar os comandos que devem ser utilizados para correr o nosso programa.

1. **mvn clear** - Reset ao maven
2. **mvn compile** - Compilar o maven
3. **mvn test** - Correr os testes com o maven
4. **mvn test pitest:mutationCoverage** - Correr os testes com o maven e gerar o relatório da cobertura dos testes e das mutações mortas. O caminho para o relatório é: *target/pit-report/index.html*.
5. **mvn evosuite:generate** - Gerar testes unitários com o evosuite
6. **mvn evosuite:export** - Dar merge dos testes gerados pelo evosuite para a nossa pilha de testes
7. **python3 logs.py** - Gerar um ficheiro logs.
8. **mvn test jacoco:report** - Correr os testes com o maven e gerar o relatório de cobertura com o Jacoco. O caminho para o relatório é: *target/site/jacoco/index.html*.

5 Conclusão e Análise dos Resultados

Em modo de conclusão, após a finalização do Trabalho Prático da Unidade Curricular *Análise e Teste de Software*, sentimos que fomos capazes de aprofundar e aplicar os nossos conhecimentos adquiridos ao longo das aulas teóricas e práticas da respetiva UC, tais como escrever testes *JUnit*, utilizar o sistema *evoSuite*, analisar a cobertura dos testes e a qualidade dos mesmos, utilizar o sistema de mutação de código para *Java PIT* e utilizar o sistema *Hypotesis*.

Deste modo, finalizado este projeto, o grupo está satisfeito com o produto final, considerando ir de encontro ao que foi solicitado, apesar das dificuldades encontradas ao longo da execução do projeto.

A Anexos

A.1 JUnit no *pom.xml*

```
1 <dependencies>
2   <dependency>
3     <groupId>junit</groupId>
4     <artifactId>junit</artifactId>
5     <version>4.13.1</version>
6     <scope>test</scope>
7   </dependency>
8   <dependency>
9     <groupId>org.junit.jupiter</groupId>
10    <artifactId>junit-jupiter</artifactId>
11    <version>5.8.1</version>
12    <scope>test</scope>
13  </dependency>
14  <dependency>
15    <groupId>org.junit.jupiter</groupId>
16    <artifactId>junit-jupiter-api</artifactId>
17    <version>5.9.2</version>
18  </dependency>
19  <dependency>
20    <groupId>org.junit.vintage</groupId>
21    <artifactId>junit-vintage-engine</artifactId>
22    <version>5.9.2</version>
23  </dependency>
24 </dependencies>
```

A.2 PITEST no ficheiro pom.xml

```
1 <plugin>
2   <groupId>org.pitest</groupId>
3   <artifactId>pitest-maven</artifactId>
4   <version>1.8.0</version>
5   <dependencies>
6     <dependency>
7       <groupId>org.pitest</groupId>
8       <artifactId>pitest-junit5-plugin</artifactId>
9       <version>0.16</version>
10    </dependency>
11  </dependencies>
12  <configuration>
13    <verbose>true</verbose>
14    <outputFormats>
15      <outputFormat>XML</outputFormat>
16      <outputFormat>HTML</outputFormat>
17    </outputFormats>
18    <exportLineCoverage>true</exportLineCoverage>
19    <!--We want each report to override the former one-->
20    <timestampedReports>>false</timestampedReports>
21    <excludedMethods>
22      <excludedMethod>java.lang.System.println</excludedMethod>
23      <excludedMethod>java.io.PrintStream.println</excludedMethod>
24    </excludedMethods>
25    <coverageIgnore>java/io/PrintStream println</coverageIgnore>
26  </configuration>
27 </plugin>
```

A.3 EvoSuite no ficheiro pom.xml

```
1 <dependencies>
2   <dependency>
3     <groupId>org.evosuite</groupId>
4     <artifactId>evosuite-standalone-runtime</artifactId>
5     <version>1.0.6</version>
6     <scope>test</scope>
7   </dependency>
8 </dependencies>
9 <build>
10   <plugins>
11     <plugin>
12       <groupId>org.evosuite.plugins</groupId>
13       <artifactId>evosuite-maven-plugin</artifactId>
14       <version>1.0.6</version>
15       <executions><execution>
16         <goals> <goal> prepare </goal> </goals>
17         <phase> process-test-classes </phase>
18       </execution></executions>
19       <configuration>
20         <extraArgs> -Duse_separate_classloader=false</extraArgs>
21       </configuration>
22     </plugin>
23     <plugin>
24       <groupId>org.apache.maven.plugins</groupId>
25       <artifactId>maven-surefire-plugin</artifactId>
26       <version>3.0.0-M6</version>
27       <configuration>
28         <properties>
29           <property>
30             <name>listener</name>
31             <value>
32               org.evosuite.runtime.InitializingListener
33             </value>
34           </property>
35         </properties>
36       </configuration>
37     </plugin>
38     <plugin>
39       <groupId>org.codehaus.mojo</groupId>
40       <artifactId>build-helper-maven-plugin</artifactId>
41       <version>1.8</version>
42       <executions>
43         <execution>
44           <id>add-test-source</id>
45           <phase>generate-test-sources</phase>
46           <goals>
47             <goal>add-test-source</goal>
48           </goals>
49           <configuration>
50             <sources>
51               <source>$.evosuite/evosuite-tests</source>
52             </sources>
53           </configuration>
54         </execution>
55       </executions>
56     </plugin>
57   </plugins>
58 </build>
```

A.4 Jacoco no ficheiro pom.xml

```
1 <plugin>
2   <groupId>org.jacoco</groupId>
3   <artifactId>jacoco-maven-plugin</artifactId>
4   <version>0.8.5</version>
5   <executions>
6     <execution>
7       <goals>
8         <goal>prepare-agent</goal>
9       </goals>
10    </execution>
11    <execution>
12      <id>report</id>
13      <phase>test</phase>
14      <goals>
15        <goal>report</goal>
16      </goals>
17    </execution>
18  </executions>
19 </plugin>
```