



UNIVERSIDADE DO MINHO

LICENCIATURA EM ENGENHARIA INFORMÁTICA

Análise e Teste de Software

Trabalho Prático

Inês Ferreira (A97372)

Joana Branco (A96584)

16 de maio de 2023

Índice

1	Introdução	4
2	Unit Testing	5
2.1	JUnit	5
2.2	Test Coverage	7
3	Unit Test Generation	10
3.1	EvoSuite	10
4	Mutation Testing	12
4.1	PIT	12
5	QuickCheck	14
5.1	Generators	14
5.2	Hypothesis	15
6	Análise de testes - PIT vs Evosuite	18
7	Conclusão	20

Lista de Figuras

1	Exemplo da <code>setCustoInstalacao</code> da classe <code>Casa</code>	6
2	Exemplo da <code>testsetCustoInstalacao</code> da classe <code>CasaTest</code>	6
3	Exemplo da <code>setDimensao</code> da classe <code>smartBulb</code>	6
4	Exemplo da <code>testsetDimensao</code> da classe <code>smartBulbTest</code>	7
5	Relatório dado por <i>Test Coverage</i> com <i>IntelliJ</i> como <i>Coverage Runner</i>	8
6	Gráfico <i>Test Coverage</i>	9
7	Configuração do PIT	12
8	Gráfico percentagens da execução de PIT	13
9	Log original do projeto	14
10	Logs criados através do <i>Quickcheck</i>	15
11	Ficheiro original de <i>logs.txt</i>	16
12	Ficheiro gerado por Hypothesis , <i>sh_logs.txt</i>	17
13	Gráfico <i>Line Coverage</i> PIT VS EvoSuite	18
14	Gráfico <i>Mutation Coverage/Scorer</i> PIT VS EvoSuite	19

1 Introdução

Este trabalho surge no âmbito da Unidade Curricular de Análise e Teste de Software com o objetivo de desenvolver testes de modo a cobrir o projeto prático de Programação Orientada aos Objetos do ano letivo anterior, sobre "Casa Inteligente". Neste caso foi utilizado como orientação o projeto prático dos elementos do grupo.

De modo a desenvolver o presente trabalho vão ser apresentados diversos testes e modos de cobertura dos mesmos tal como enunciado. Recorrendo a vários tipos de testes tais como **testes unitários**, onde vamos usar *JUnit* e *Maven*, **geração de teste unitários**, usando o *EvoSuite*, **mutações** com recurso a *PIT* e **QuickCheck** usando *Generators* e *Hypothesis*.

2 Unit Testing

Unit Testing é uma boa prática para se poder ter a certeza do quão correto o código implementado num programa está. Normalmente pode-se criar um teste que se foque apenas num método ou numa classe. Assim sendo, estes testes mencionados, testes unitários, vêm validar o *input* do código em questão através de diversas regras de testagem.

O **JUnit** é usado como *framework* para testes unitários no caso da linguagem *Java*.

2.1 JUnit

Para analisar e executar testes unitários no projeto em causa fez-se uso da ferramenta **JUnit5**, estando em causa a linguagem *Java*. O objetivo é criar um ambiente de simulação para algumas funções das diferentes classes do projeto. Com isto é possível ter a certeza da veracidade do código implementado e o quão correto é o seu resultado.

A descrição dos testes mencionados pode ser encontrada na pasta **POO_2022/src/main/_test**. Neste caso foram criadas várias classes com o sufixo *Test* de modo a desenvolvermos os testes unitários das classes principais do projeto sobre "Casa Inteligente". Nas diversas classes criadas, foram produzidos diversos testes com alvo num método em específico ou, por vezes, numa classe. Para permitir a validação do resultado obtido em relação ao esperado, fez-se uso de funções do tipo *assert* que permitem este tipo de comparação.

Fez-se uso desta ferramenta através da execução normal de uma classe em *Java*, com a configuração na classe de Teste pretendido, carregando no botão de "Play". Aqui é possível ver quantos testes passaram e não apresentam qualquer tipo de erros.

Considerando os testes unitários presentes denotou-se que, quando existe um erro é feito um *print* de uma mensagem do erro correspondente, executado pela função **erros** da classe **Menu**. Com isto, a decisão tomada pelo grupo de realização do trabalho de Programação Orientada aos Objetos não foi a melhor e pode-se afirmar que há alternativas melhores como a apresentação de uma exceção aquando aparecimento de um erro ou então, por exemplo, uma mensagem de erro exibida pelo *assertEquals*.

Como exemplo apresenta-se a função **setCustoInstalacao** que, interpretando o código apresentado na imagem seguinte podemos observar que a função verifica se o valor recebido é positivo ou não. Caso o *input* seja um valor menor que 0 (zero) é executado um *print* "Custo de instalação inválido" mas, deveria ser apresentada uma exceção aquando o aparecimento deste mesmo erro.

```

public void setCustoInstalacao(double custoInstalacao) {
    if(custoInstalacao >= 0) this.custoInstalacao = custoInstalacao;
    else Menu.error(33);
}

```

Figura 1: Exemplo da `setCustoInstalacao` da classe `Casa`

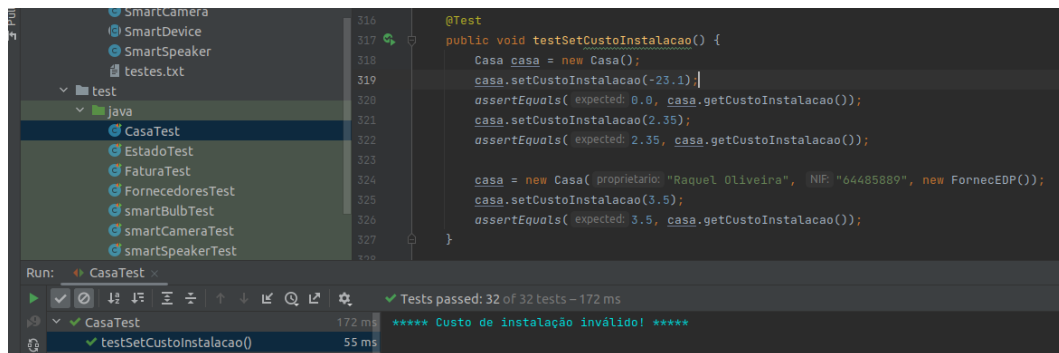


Figura 2: Exemplo da `testsetCustoInstalacao` da classe `CasaTest`

Um outro exemplo é o `setDimensao` da classe `smartBulb` que envia um *print* para o utilizador com a mensagem "Dimensao invalida!". Neste caso não deveria ser executado o *print*, que pode ser verificado na classe `smartBulbTest`, mas seria mais correto o lançamento de uma exceção para quando fosse recebido um valor negativo para definir a lâmpada (*smart-Bulb*).

```

public void setDimensao(double dimensao) {
    if(dimensao >= 0) this.dimensao = dimensao;
    else {
        System.out.println("Dimensao invalida!");
        this.dimensao = 0.0;
    }
}

```

Figura 3: Exemplo da `setDimensao` da classe `smartBulb`

Na figura seguinte é possível verificar que todos os testes passaram e, em específico, o teste `testSetDimensao` apresenta a mensagem da função `setDimensao` por causa de haver a definição do valor menor que 0 (zero), -12.9.

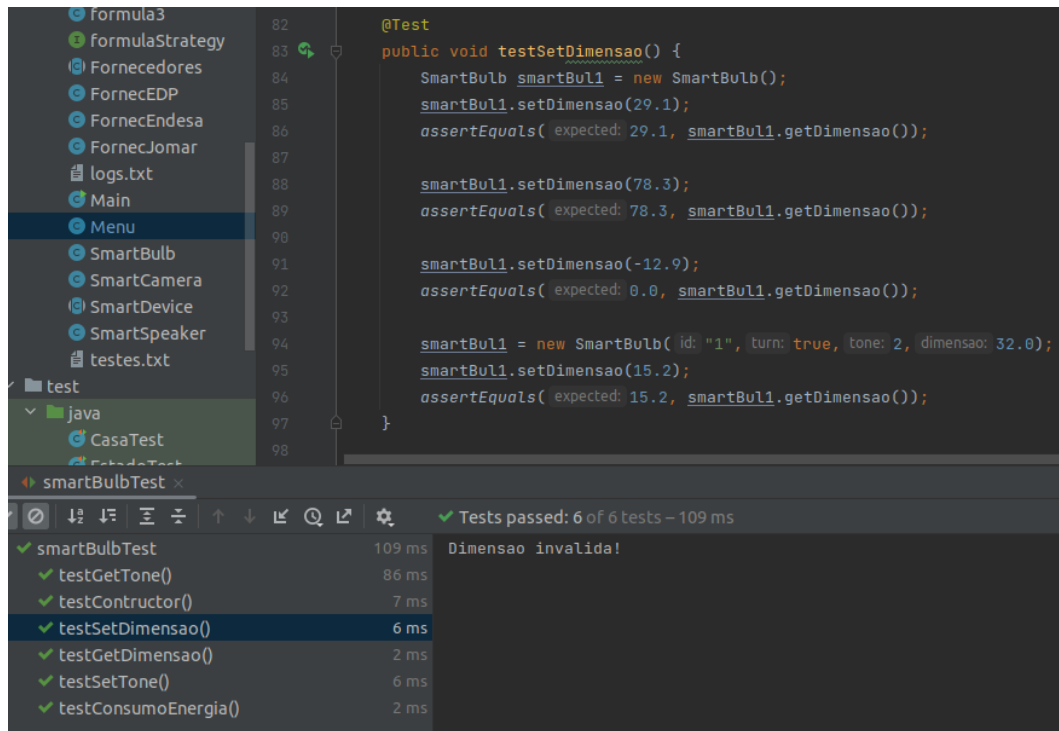


Figura 4: Exemplo da `testsetDimensao` da classe `smartBulbTest`

Contudo, depois de executadas todas as classes do tipo **Test**, onde foram especificados os testes unitários, podemos afirmar que os resultados obtidos foram todos positivos, todos os testes passaram sem erros nem *warnings*, e que os testes confirmam a correta implementação do código (à exceção do mencionado anteriormente).

2.2 Test Coverage

A *Test Coverage* permite analisar a cobertura dos testes unitários feitos. Com isto é possível termos uma noção de quantos métodos e linhas de código estão a ser cobertos pelos testes unitários feitos.

Utilizar a ferramenta exige a necessidade de termos a configuração da classe em questão (entre as de Teste) e executando o botão *"Run with Coverage"*. Após a apresentação de uma tabela com todas as classes do projeto onde podemos aceder a *"Generate Coverage Report"* que gera uma página *html* que apresenta em específico que partes do código foram cobertas pelos testes e as percentagens do quão estes abrangem o código em questão.

Usando a ferramenta de configuração do *IntelliJ*, definiu-se este mesmo como *Coverage Runner*. O relatório resultante pode ser observado na figura seguinte.

Package	Class, %	Method, %	Line, %
<empty package>	63% (17/27)	72,8% (150/206)	41,5% (533/1284)


Class 	Class, %	Method, %	Line, %
Automatizacao	0% (0/1)	0% (0/4)	0% (0/56)
Casa	100% (3/3)	97,6% (40/41)	88% (161/183)
Controller	0% (0/1)	0% (0/2)	0% (0/25)
ControllerAutomatizacao	0% (0/1)	0% (0/2)	0% (0/4)
ControllerCasa	0% (0/1)	0% (0/4)	0% (0/224)
ControllerEstado	0% (0/1)	0% (0/2)	0% (0/26)
ControllerEstatistica	0% (0/1)	0% (0/2)	0% (0/93)
ControllerFornecedores	0% (0/1)	0% (0/2)	0% (0/58)
ControllerSimulacao	0% (0/1)	0% (0/2)	0% (0/58)
Estado	100% (1/1)	79,3% (23/29)	90,1% (155/172)
Faturas	100% (1/1)	90% (9/10)	79,4% (27/34)
FornecEDP	100% (1/1)	100% (5/5)	100% (5/5)
FornecEndesa	100% (1/1)	100% (5/5)	100% (5/5)
FornecJomar	100% (1/1)	100% (5/5)	100% (5/5)
Fornecedores	100% (2/2)	88,9% (16/18)	84,6% (33/39)
Main	0% (0/1)	0% (0/2)	0% (0/2)
Menu	100% (1/1)	6,7% (1/15)	24,7% (37/150)
SmartBulb	100% (1/1)	83,3% (10/12)	71,9% (23/32)
SmartCamera	100% (1/1)	83,3% (10/12)	68,6% (24/35)
SmartDevice	100% (1/1)	80% (8/10)	75% (18/24)
SmartSpeaker	100% (1/1)	87,5% (14/16)	75,6% (34/45)
formula1	0% (0/1)	0% (0/2)	0% (0/3)
formula2	100% (1/1)	100% (2/2)	100% (3/3)
formula3	100% (1/1)	100% (2/2)	100% (3/3)

Figura 5: Relatório dado por *Test Coverage* com *IntelliJ* como *Coverage Runner*

Através da análise da relatório anterior consegue-se perceber que maior parte das classes (63%) são cobertas pelos testes unitários previamente feitos e, há uma grande cobertura tanto dos métodos como linhas de código. Há que ter em conta que não foram implementados testes para todas as classes, excluimos as que não consideramos relevantes para verificação dos seus métodos. Estas são **Controller**, e suas demais, e **Main**.

Para uma melhor percepção visual, construiu-se um gráfico que apresenta as percentagens de *Test Coverage* apenas para as classes que foram cobertas pelos testes unitários, ou seja, as que contém pelo menos um teste para um dos seus métodos.

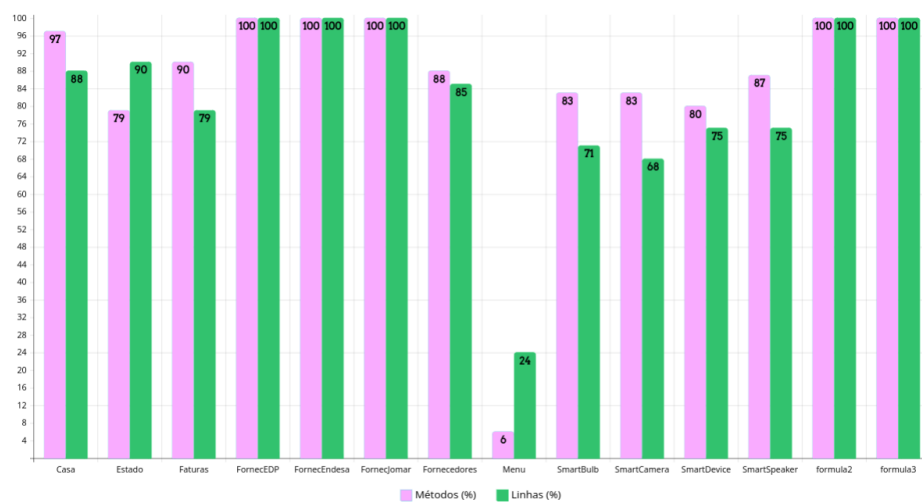


Figura 6: Gráfico *Test Coverage*

3 Unit Test Generation

3.1 EvoSuite

Evosuite é uma ferramenta que gera testes unitários que usa algoritmos de busca baseados em otimização para gerar casos de teste que proporcionam uma cobertura abrangente para o nosso código. Esta ferramenta analisa o código-fonte de uma classe ou de *packages* e, em seguida, gera testes que cobrem diversos cenários de entrada e comportamentos esperados. Esses testes são criados automaticamente sem a necessidade de intervenção manual.

Visto que a ferramenta **EvoSuite** apenas é funcional para código em java 8, e como o nosso trabalho estava codificado em *java 16*, mudaram-se **todas as classes** do nosso código com dependências ligadas a versões de *java* superiores à 8, como *switches* e sequência de caracteres multi-linhas em formato de *string* delimitada por três aspas duplas (`"""`), e modifica-las para código operacional *java 8*.

Depois de todas as modificações e *set-up* necessários, executou-se a ferramenta *evosuite* nas classes do projeto. Após a execução foi apresentada as classes teste unitários e as suas estatísticas geradas. Na tabela a baixo segue-se as estatísticas avaliadas, de cada classe, que consideramos mais relevantes:

Resultados de EvoSuite		
Classe	Line Coverage	Weak Mutation Scorer
Automatização	0.109375	0.034782608695652174
Casa	0.9900990099009901	0.9512893982808023
Controller	0.7857142857142857	0.7111111111111111
Estatística	0.5	0.3696682464454976
Estado	0.5	0.3696682464454976
Faturas	1.0	1.0
Formula1	1.0	1.0
Formula3	1.0	1.0
Fornecedores	1.0	0.9841269841269841
FornecEdp	1.0	1.0
FornecEndesa	1.0	1.0
FornecJomar	1.0	1.0
Main	0.666666666	1.0
Menu	1.0	0.9815242494226328
SmartBulb	1.0	1.0
SmartBulb	1.0	1.0
SmartSpeaker	1.0	0.98125
SmartDevice	0.8275862068965517	0.7

Line Coverage - Corresponde à cobertura sobre o código(alto se for bem analisado baixa caso contrário).

Weak Mutation Scorer - Corresponde ao *Mutation Test* no *PIT*.

4 Mutation Testing

Este procedimento é utilizado essencialmente para avaliar a qualidade e eficácia de um caso de teste. Neste caso são criados mutações ou falhas propositadas no código em questão e, os de testes devem ser capazes de decodificar e "matar" o mutante em específico. Desta forma, é possível detetar erros ou falhas que o humano possa ter cometido na escrita e implementação do seu código.

Esta é uma estratégia complementar a **Unit Testing** e *Code Coverage* que permite a identificação dos pontos fracos no código e a viabilidade dos testes em causa.

4.1 PIT

O **PIT** é uma ferramenta que permite geração de mutações automaticamente de modo a podermos analisar se os testes atuais são eficientes.

Fez-se uso desta ferramenta através do *plugin* do *IntelliJ*.

Para a utilização da ferramenta em questão, decidiu-se alterar a estrutura do projeto de modo a facilitar a execução do mesmo. Neste caso, também houve a necessidade de se editar a configuração de execução do programa para cada uma das classes ao qual têm testes. Assim sendo, para cada uma das classes que têm o sufixo *Test* fizemos a seguinte configuração, por exemplo, as *Target classes* correspondem as classes principais do projeto e *Target tests* o nome da classe onde foram criados os testes correspondentes.

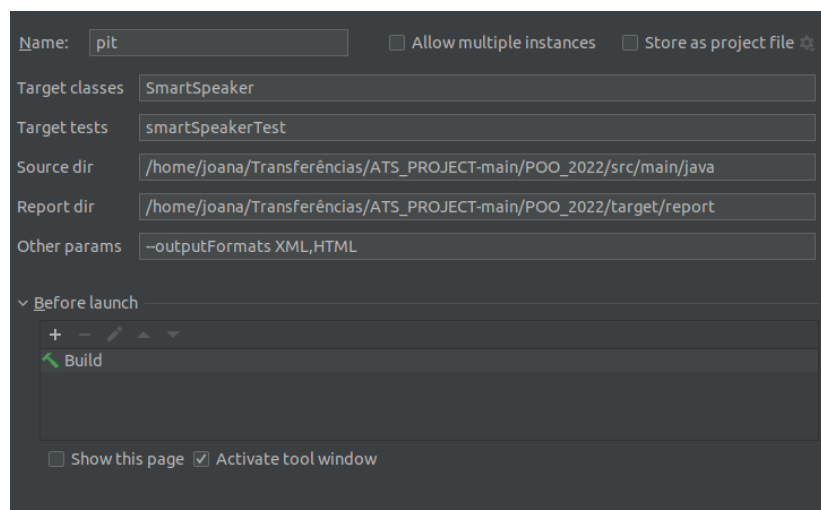


Figura 7: Configuração do **PIT**

Após a execução da ferramenta **PIT** e obtidas as diferentes percentagens para *line coverage*,

mutation coverage e *test strength* organizaram-se estes mesmos valores no seguinte gráfico para uma melhor compreensão visual.

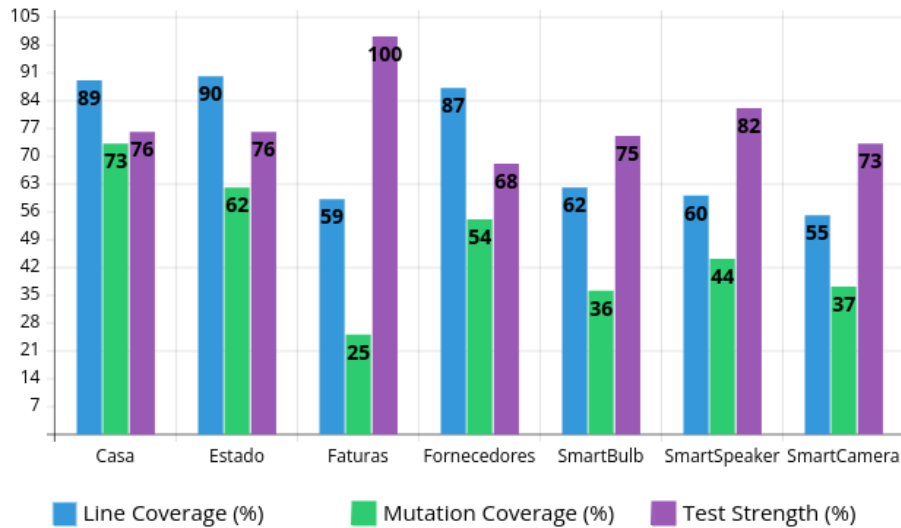


Figura 8: Gráfico percentagens da execução de **PIT**

O que se pode concluir é que, por exemplo, a classe Casa apresenta um grande equilíbrio entre os três valores. Em geral consegue-se observar valores relativamente bons (testes abrangem bastante código e estão bem construídos), acima de 50%. Em Faturas, *mutation coverage* é o que é considerado valor mais baixo em comparação às outras. Outro caso é Faturas que dispõe de 100% de *Test Strength* porque em comparação às outras tem um menor número de testes, Faturas tem 4/4 que, por exemplo, Estado tem no total 54/71.

Apesar dos valores resultantes da execução do **PIT**, não se alterou qualquer teste unitário previamente feito por isso, apenas é possível comparar estes resultados com o **EvoSuite** como é feito na secção **Análise de Testes - PIT vs EvoSuite**.

5 QuickCheck

5.1 Generators

Foi proposta uma implementação de ficheiro similar ao ficheiro *logs* de entrada, do projeto de POO do ano passado. Para realizar esta tarefa usou-se biblioteca **QuickCheck** e a linguagem *Haskell*, onde geradores são implementados usando a classe *Arbitrary*. Esta classe define a função *arbitrary*, que é usada para criar instâncias de geradores para tipos personalizados. Ao criar uma instância de *Arbitrary* para um tipo personalizado, define-se como gerar valores aleatórios desse tipo.

Foi necessário criar três tipos de instâncias *Arbitrary*, uma para a Casa, outra para Divisão, e para *SmartDevices*. Esta ultima é geral para *SmartBulb*, *SmartSpeaker* e *SmartCamera*. A partir daí criou-se uma função *generateLog* que utiliza a função *generate* do módulo *Test.QuickCheck* para gerar valores aleatórios com base nas instâncias *Arbitrary* definidas para as diferentes estruturas de dados.

Ficheiros logs do projeto inicial e o que foi gerado através do *Quickcheck*:

```
logs > ≡ logsooriginal.txt
1  Fornecedor:EDP,formula2
2  Fornecedor:Jomar,formula1
3  Fornecedor:Endesa,formula3
4  Casa:Joao Pedro Malheiro da Costa,707666276,EDP
5  Divisao:Sala de Jantar
6  SmartBulb:7906985,Neutral,60.0
7  SmartBulb:6559790,Neutral,14.0
8  Divisao:Garagem
9  SmartSpeaker:8891467,55,RFM Oceano Pacifico,Marshall
10 SmartBulb:659226,Cold,40.0
11 SmartCamera:9417540,3840,2160
12 Divisao:Jardim
13 SmartSpeaker:6659156,35,RTP Antena 1 98.3 FM,Marshall
14 SmartBulb:8981844,Warm,60.0
15 SmartSpeaker:2819614,42,RFM,Sennheiser
16 SmartSpeaker:3437522,30,M80 Radio,LG
17 SmartSpeaker:2191142,65,RTP Antena 1 98.3 FM,Sony
18 SmartSpeaker:9334471,52,Radio Renascenca,LG
19 SmartSpeaker:8814376,45,MEGA HITS,BOSE
20 Casa:Miguel Velho Raposo,134655929,Endesa
```

Figura 9: Log original do projeto

```

logs > cat log.txt
1 Fornecedor:EDP,formula2
2 Fornecedor:Jomar,formula1
3 Fornecedor:Endesa,formula3
4 Casa: Jacinta, "519253457", Endesa
5 Casa: Ines, "991152782", Jomar
6 Casa: Jacinta, "918693835", EDP
7 Casa: Miguel, "667959374", Endesa
8 Casa: Rafael, "623982874", Endesa
9 Casa: Manel, "926892894", Jomar
10 Divisao: Sala
11 SmartBulb: "0621572", Warm, 4.15693820072508
12 SmartSpeaker: "7829197", 82, RTP Marshall
13 SmartSpeaker: "4203862", 54, NovaEra Marshall
14 SmartBulb: "0176694", Neutral, 2.2122800630099926
15 SmartCamera: 4.455576909848231e7, 85134.60264100615, 69402.46317710575
16 SmartSpeaker: "4896124", 64, RTP Sony
17 SmartCamera: 8.018010365884246e7, 72766.43591780431, 86419.68790825845
18 SmartCamera: 6.255118043254765e7, 32128.39025256642, 11420.695658342358
19 SmartCamera: 6.00735043134857e7, 78893.30698475012, 42374.33002457739
20 SmartBulb: "5357138", Neutral, 2.1866207273617153
21 SmartCamera: 5.5435015695665255e7, 32226.85144909883, 8883.079836282059
22 SmartBulb: "1278226", Cold, 5.238904451991106
23 SmartSpeaker: "0651663", 43, RFM Marshall
24 SmartBulb: "4757103", Warm, 6.2608486140273865
25 SmartBulb: "4638155", Cold, 4.841556307704896
26 SmartCamera: 5.525018542436593e7, 81783.7618799634, 99444.10416137944
27 SmartSpeaker: "9029951", 70, RadioPopular LG
28 SmartBulb: "4210733", Warm, 3.599249367308259
29 SmartBulb: "0979887", Neutral, 5.321740087006637
30 SmartSpeaker: "0258477", 36, RTP Marshall
31 SmartBulb: "2745931", Warm, 3.0814562882579155
32 SmartSpeaker: "4025196", 69, RTP LG
33 SmartSpeaker: "9269066", 16, NovaEra LG
34 SmartCamera: 4699461.281615329, 26264.131789782114, 41546.92083259773
35 SmartSpeaker: "1696316", 37, RFM Marshall
36 SmartCamera: 9.802201973535445e7, 3376.240605557343, 60473.10741300226
37 SmartSpeaker: "7882604", 48, RFM Marshall

```

Figura 10: Logs criados através do *Quickcheck*

5.2 Hypothesis

O *software* em causa serve como biblioteca de testes e é usado na linguagem *Python*. Aqui é possível escrever casos de teste onde se define as propriedade e regras que desejamos que o código execute. Tendo em conta a estrutura do ficheiro de *logs.txt* já usados no projeto original de Programação Orientada aos Objetos, **Hypothesis** gerou automaticamente um ficheiro de *logs* semelhante, chamado **sh.logs.txt**.

Tendo em conta o ficheiro original e a estrutura de apresentação de *logs.txt*, criaram-se diversas propriedades que sejam capazes de construir este mesmo padrão, tal como se pode ver na figura seguinte.

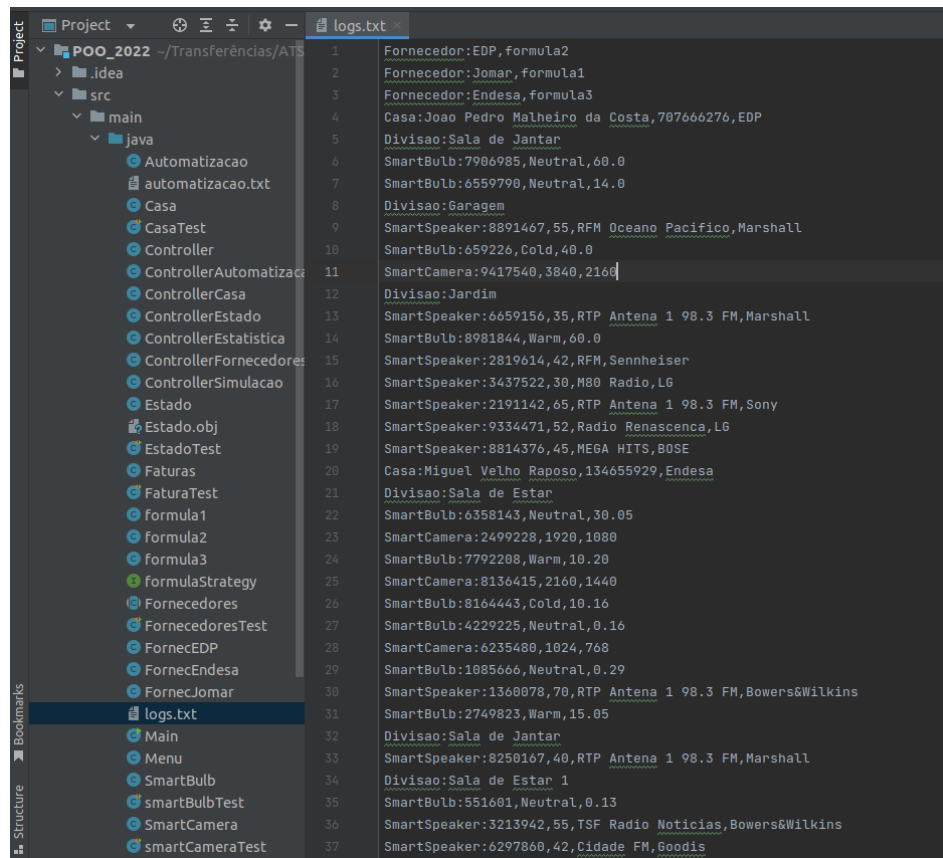


Figura 11: Ficheiro original de *logs.txt*

Iniciou-se o processo por identificar quais os fornecedores que existem, apenas existem 3 tipos, as fórmulas de cálculo de forma semelhante. Também para os nomes e divisões da casa decidiu-se criar uma lista de elementos em que podem ser escolhidos qualquer um deles. De forma semelhante foi feito para as estações de rádio e as marcas dos *speakers*. Já para o valor de nif, id, dimensão da lâmpada, volume da coluna, resolução e tamanho da câmara, fez-se variar entre dois valores atribuindo-lhe um mínimo e máximo.

Com estas atribuições, procedeu-se à construção das linhas como era suposto e da recursividade necessária para que sejam gerados no mínimo 30 e 150 linhas.

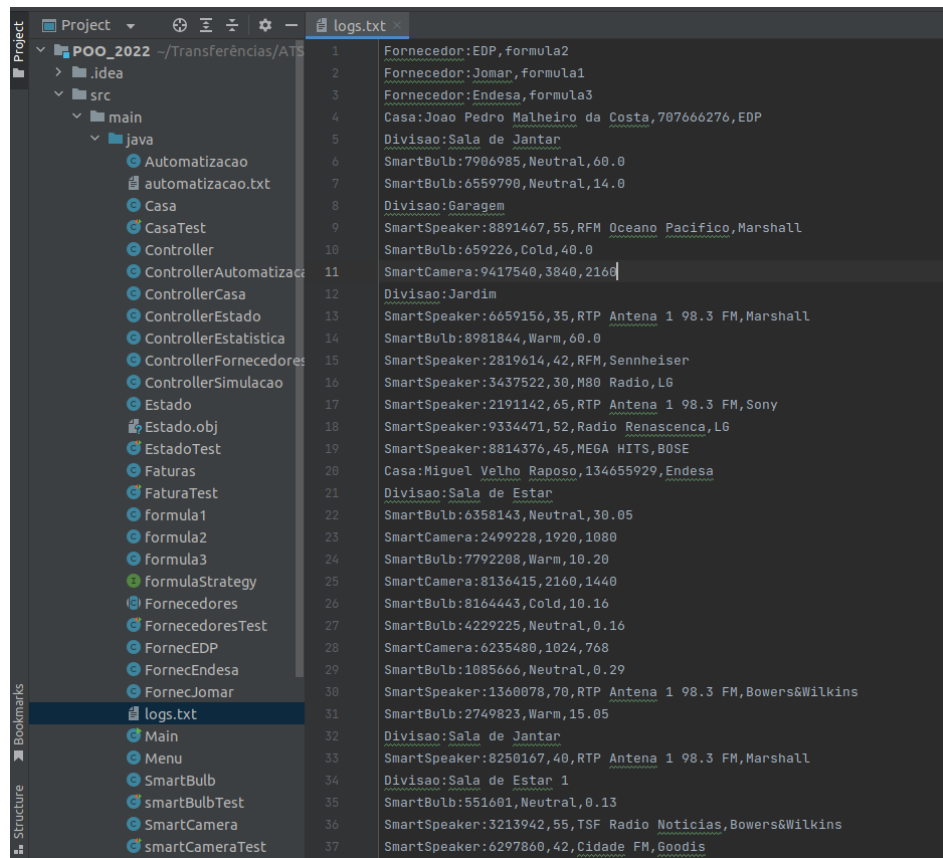


Figura 12: Ficheiro gerado por **Hypothesis**, **sh_logs.txt**

Assim, pode-se verificar que o ficheiro gerado corresponde ao ficheiro original que foi tido como modelo.

E, com isto, é possível confirmar que o ficheiro gerado pode ser inserido no projeto de *Smart Houses* em causa sem qualquer erro.

6 Análise de testes - PIT vs EvoSuite

Para se ter uma melhor observação visual sobre as capacidades do **EvoSuite** em relação ao **PIT** e vice-versa apresentam-se os seguintes gráficos abaixo, onde comparamos a *Line Coverage* e *Mutation Coverage/Scorer* de ambas ferramentas.

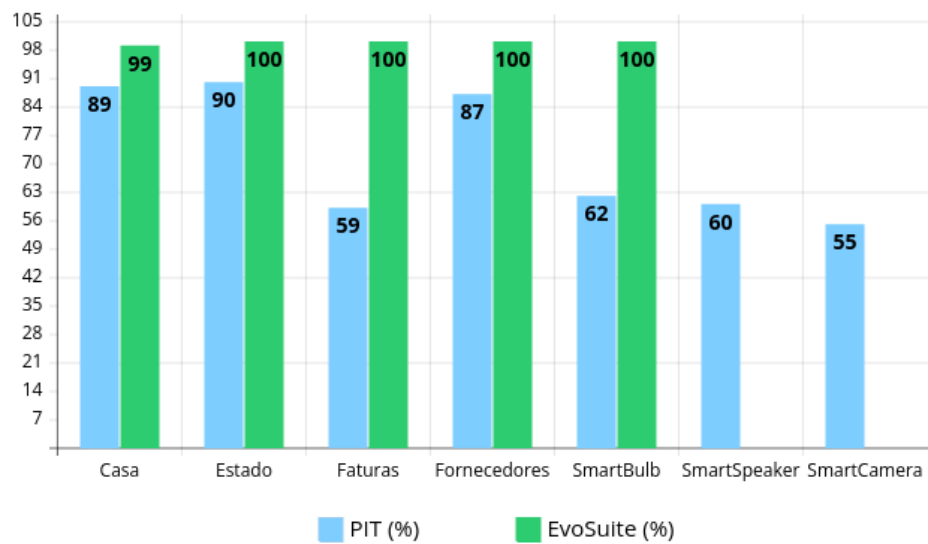


Figura 13: Gráfico *Line Coverage* PIT VS EvoSuite

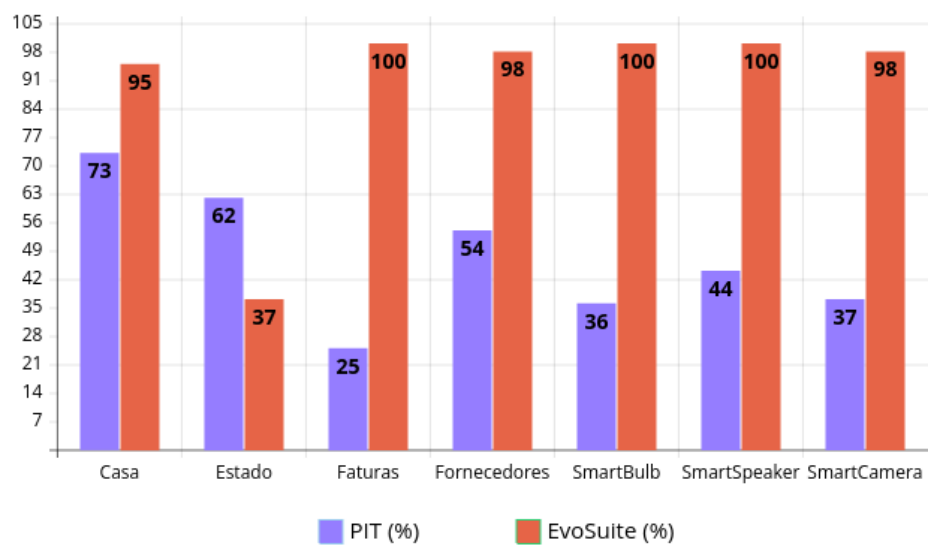


Figura 14: *Gráfico Mutation Coverage/Scorer PIT VS EvoSuite*

7 Conclusão

Apesar que se concluir que foi um projeto bem conseguido, apresentamos algumas dificuldades com as ferramentas utilizadas. Verificou-se que os testes feitos através do **Evosuite** apresentaram, em geral, altas percentagens de cobertura do código e de *Weak Mutation Scorer*, infere-se então que é um bom utensílio para testar código de maneira rápida e eficaz, embora seja necessário ter o projeto configurado para a versão *java 8*, ficando assim o código mais rudimentar.

Em relação às demais ferramentas usadas é visível o melhor conhecimento do funcionamento das mesmas e, por exemplo, o **PIT** permitiu uma melhor análise dos testes unitários feitos e, oportunidade de melhoramento dos mesmos.