Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Análise e Teste de Software

Ano Letivo de 2022/2023

Smart Testing for Smart Houses

Cláudio Alexandre Freitas Bessa (a97063) Gabriel Alexandre Monteiro da Silva (a97363) José Fernando Monteiro Martins (a97903)

10 de maio de 2023

Indice

1 Introdução

O documento apresentado visa apresentar as metodologias praticadas e processos utilizados durante o projeto no âmbito da UC de Análise e Teste de Software.

Para o desenvolvimento deste projeto é requerido um conjunto de etapas/objetivos que visam o que vem sido lecionado ao longo deste semestre, i.e. um conjunto de ferramentas, *plugins* e/ou *frameworks* que permitem analizar e testar o código criado, assim como as suas *features* de modo a verificar a sua integridade, eligibilidade, qualidade e corretividade.

Este conjunto de processos desenvolvidos para a testagem recai sobre o projeto da UC de POO, i.e. programação orientada a objetos, realizada há 2 semestres atrás, que tinha como tema ***SmartHouses***, i.e. casas inteligentes.

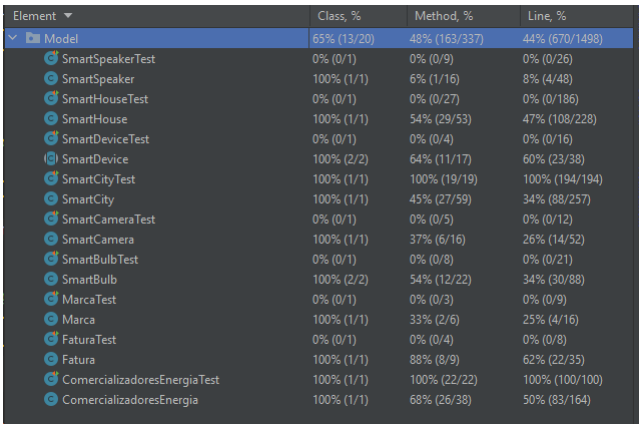
2 Objetivos

- Escrever testes **JUnit** para o sistema de *software* considerado
- Utilizar o sistema **evoSuite** para gerar testes automaticamente
- Utilizar o sistema de mutação de código para Java **PIT**
- Uitilizar o sistema de geração automática de casos de teste **QuickCheck** para gerar ficheiro de *logs*
- Analisar a qualidade do código fonte através de **sonarQube**
- Gerar automaticamente casos de teste utilizando o sistema **Hypothesis**
- Testar propriedados do projeto de POO segundo o sistema de ***Property Bases Testing***
- Analisar cobertura dos testes e a qualidade dos testes

3 *Unit Testing*

O projeto de POO, apesar do seu tema, recaía, 100% sobre os conceitos de orientação de objetos e sobre o modelo MVC, daí a existência da cadeira. Devido a isso o nosso projeto contém as diretorias, *model*, *view* e *controller*. Para a realizações dos testes unitários, efetuamos apenas a testagem nas classes existentes na diretoria *model*, pois aí é que tem todas as funções relevantes relativas a todos os componentes e atributos deste ambiente. Daí originaram-se um total de 10 classes de teste sendo elas: **ComercializadoresEnergiaTest**, **FaturaTest**, **MarcaTest**, **SmartBulbTest**, **SmartCameraTest**, **SmartSpeakerTest**, **SmartDeviceTest**, **SmartHouseTest** e **SmartCityTest**.

Através da figura 3.1 é visível que o *coverage* encontra-se por volta dos 60 a 70%, isto devido a ser um projeto com bastantes objetos, existindo a preocupação de manter a **OO**, i.e. orientação a objetos, a eminência de deixar objetos como privado faz com que tenha que existir muitos *getters*, *setters* e construtores. Analisando isso acreditamos que seria desnecessário estar a ”duplicar”ou até ”triplicar”código quando verificando certas propriedas iria fazer várias ao mesmo tempo. Outro motivo é o facto de existirem testes unitários que possuem mais do que um *assert*.[1] Algo que tentamos ao máximo não utilizar visto que se titulam testes unitários por algum motivo.



Element	Class, %	Method, %	Line, %
Model	65% (13/20)	48% (163/337)	44% (670/1498)
SmartSpeakerTest	0% (0/1)	0% (0/9)	0% (0/26)
SmartSpeaker	100% (1/1)	6% (1/16)	8% (4/48)
SmartHouseTest	0% (0/1)	0% (0/27)	0% (0/186)
SmartHouse	100% (1/1)	54% (29/53)	47% (108/228)
SmartDeviceTest	0% (0/1)	0% (0/4)	0% (0/16)
SmartDevice	100% (2/2)	64% (11/17)	60% (23/38)
SmartCityTest	100% (1/1)	100% (19/19)	100% (194/194)
SmartCity	100% (1/1)	45% (27/59)	34% (88/257)
SmartCameraTest	0% (0/1)	0% (0/5)	0% (0/12)
SmartCamera	100% (1/1)	37% (6/16)	26% (14/52)
SmartBulbTest	0% (0/1)	0% (0/8)	0% (0/21)
SmartBulb	100% (2/2)	54% (12/22)	34% (30/88)
MarcaTest	0% (0/1)	0% (0/3)	0% (0/9)
Marca	100% (1/1)	33% (2/6)	25% (4/16)
FaturaTest	0% (0/1)	0% (0/4)	0% (0/8)
Fatura	100% (1/1)	88% (8/9)	62% (22/35)
ComercializadoresEnergiaTest	100% (1/1)	100% (22/22)	100% (100/100)
ComercializadoresEnergia	100% (1/1)	68% (26/38)	50% (83/164)

Figura 3.1: *Testes Unitários*

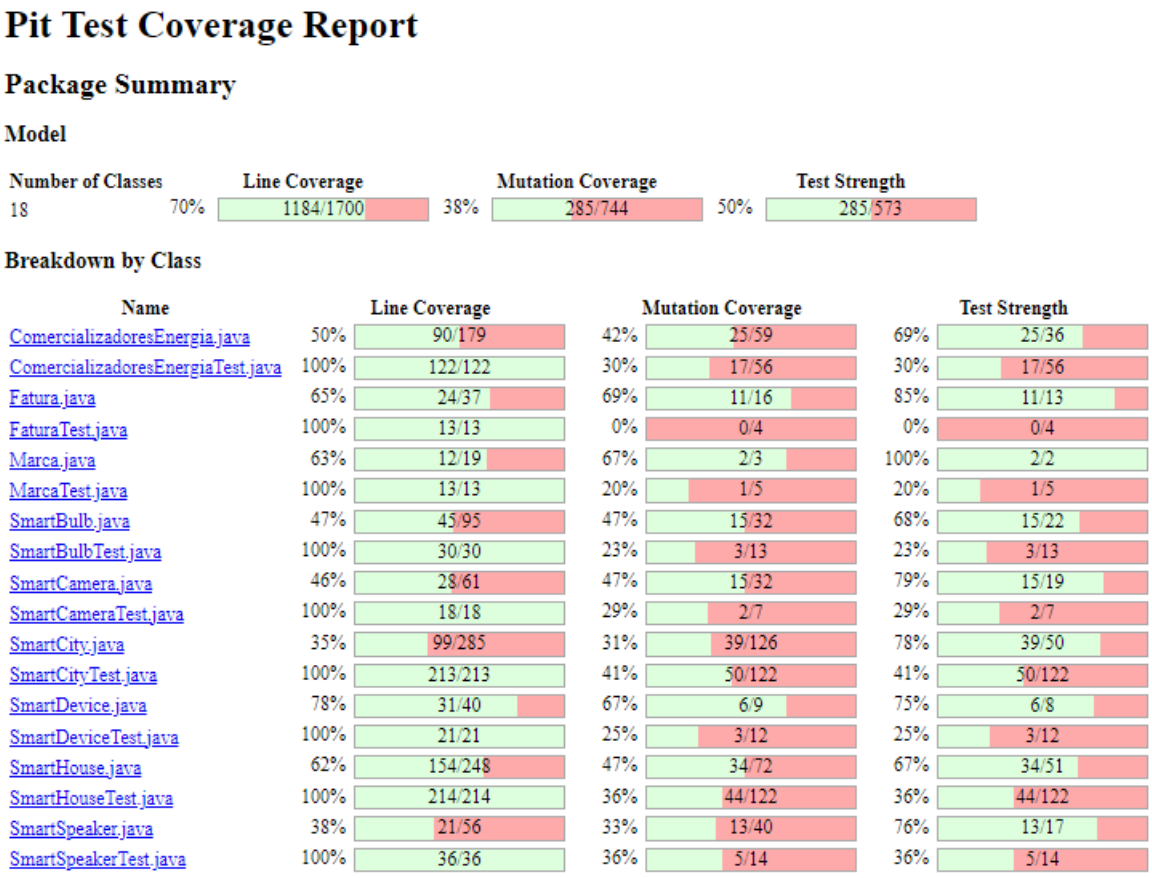
1. Antony, *Is it OK to have multiple asserts in a single unit test?*, https://softwareengineering.stackexchange.com/questions/7823/is-it-ok-to-have-multiple-asserts-in-a-single-unit-test, Accessed: 04/05/2023.

Tendo conhecimento que o projeto continha algumas fragilidades e/ou *bugs*, à medida que fomos criando as diferentes funções para testes fomos corrigindo algumas fragilidades. É de destacar a alteração do método **addCasa** da classe **ComercializadoresEnergia**, que sofreu uma redefinição visto que duplicava os clientes que tinha na sua memória, método **setDivisaoOn** na classe **SmartHouse** que não trocava o estados dos diferentes *SmartDevices* de forma independente, apenas o seu estado quando acedidos pela classe da casa. Por fim a class principal, i.e. **SmartCity**, continha alguns problemas, originando algumas dificuldades na criação de funções de testagem, métodos esses sendo: **merge**, **addDevicesDivisao**, **states**. Identificamos assim que a criação de testes unitários foi bastante simples mas foi aumentando o seu nível de dificuldade consoante o número de ligações que cada classe podia ter em relação a outras.

4 PIT

Para o sistema do **PIT** o processo acaba por ser automatizado, utilizando os métodos que tinhamos definido, incluindo agora alguns mutantes também pre-definidos, entre outros testes que o próprio sistema acaba por desenvolver.[1]

Através da figura 4.1 é visível que existe uma cobertura de 60-70% mas grande parte das mutações sobrevivem. Isto devido a fragilidades já mencionadas , sendo maioritarimente fragilidades relativas aos grandes objetos, i.e. **SmartHouse** e **SmartCity**.



Pit Test Coverage Report

Package Summary

Model

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
18	70%	1184/1700	38%	285/744	50%	285/573

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
ComercializadoresEnergia.java	50%	90/179	42%	25/59	69%	25/36
ComercializadoresEnergiaTest.java	100%	122/122	30%	17/56	30%	17/56
Fatura.java	65%	24/37	69%	11/16	85%	11/13
FaturaTest.java	100%	13/13	0%	0/4	0%	0/4
Marca.java	63%	12/19	67%	2/3	100%	2/2
MarcaTest.java	100%	13/13	20%	1/5	20%	1/5
SmartBulb.java	47%	45/95	47%	15/32	68%	15/22
SmartBulbTest.java	100%	30/30	23%	3/13	23%	3/13
SmartCamera.java	46%	28/61	47%	15/32	79%	15/19
SmartCameraTest.java	100%	18/18	29%	2/7	29%	2/7
SmartCity.java	35%	99/285	31%	39/126	78%	39/50
SmartCityTest.java	100%	213/213	41%	50/122	41%	50/122
SmartDevice.java	78%	31/40	67%	6/9	75%	6/8
SmartDeviceTest.java	100%	21/21	25%	3/12	25%	3/12
SmartHouse.java	62%	154/248	47%	34/72	67%	34/51
SmartHouseTest.java	100%	214/214	36%	44/122	36%	44/122
SmartSpeaker.java	38%	21/56	33%	13/40	76%	13/17
SmartSpeakerTest.java	100%	36/36	36%	5/14	36%	5/14

Figura 4.1: *PIT Test Coverage Report*

1. Matt Kirk, *Why PIT?*, https://pitest.org/, Accessed: 04/05/2023.

5 *QuickCheck*

Geradores são um ponto crucial no que toca no processo de testagem de um *software*. De igual forma, são uma ferramenta importante no *QuickCheck*, uma biblioteca de testes automatizados, incialmente criada para *Haskell*. São utilizados para gerar dados aleatórios que serão usados, neste caso, como ficheiros *logs* de modo a conseguirmos definir uma ***SmartCity*** de forma completamente aleatória, gerando ***SmartHouses***, **Comercializadores de Energia** e todos os tipos de ***SmartDevices***.

Todas as funções inicializadas por **"gen"** acabam por ter a funcionalidade de um gerador. Para criar um, pode-se usar diversos tipos de funções de alto nível, e.g. *choose*, *elements*, etc. Sendo totalmente personalizáveis podemos adaptar a todas as circunstâncias.

Neste caso em específico, a geração de uma ***SmartCity*** acaba por seguir um conjunto de regras, que já tinham sido definidas pelos criadoress do projeto de POO que estamos a utilizar. Os gerados tem como objetivo gerar strings com o seguinte formato:

- **ComercializadoresEnergia** -> "Fornecedor:<Nome Comercializador de Energia>"
- ***SmartHouse*** -> "Casa:<Nome Proprietario>,<NIF>,<Nome Comercializador de Energia>"
- **Divisão** -> "Divisao:<Nome Divisão>"
- ***SmartBulb*** -> "SmartBulb:<Modo>,<Dimensão>,<Consumo>"
- ***SmartCamera*** -> "SmartCamera:<Resolução>,<Dimensão>,<Consumo>"
- ***SmartSpeaker*** -> "SmartSpeaker:<Volume>,<Nome Rádio>,<Marca>,<Consumo>"

É de igual forma de realçar que, a leitura do ficheiro de *logs* necessita de uma ordem específica para que possam ser efetuada um *parse* do mesmo de forma correta. Isto obviamente foi um dos pontos cruciais, e consequentemente com um grau de complexidade superior no que toca a criar um bom gerador. Essa hierarquia é, portanto, inicialmente define-se os fornecedores de energia, uma vez que as casas necessitam de ter um comercializador de energia que já existe. Seguidamente é respeitada uma hierarquia entre casas, e os retantes componentes, uma vez que uma casa existe, seguidamente as suas divisões e posteriormente os dispositivos inteligentes que vivem na mesma.

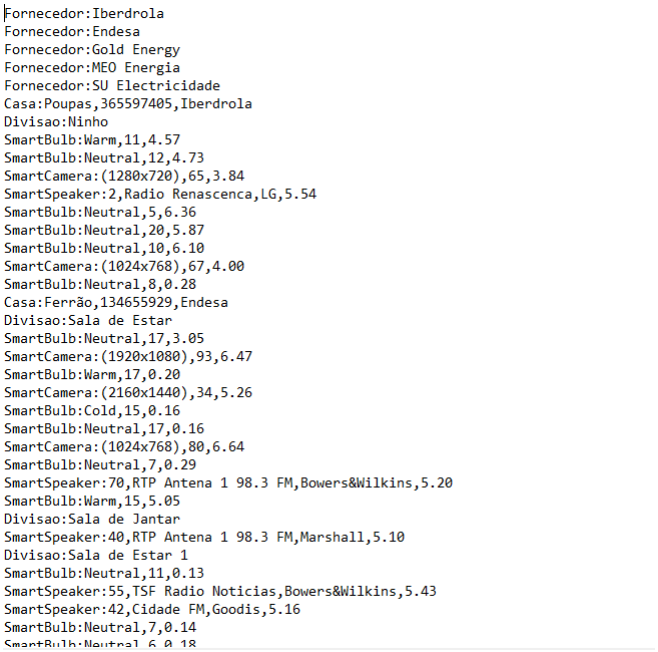


Figura 5.1: Exemplo de ficheiro *Logs*

6 Extras

6.1 *SonarQube*

SonarQube é uma ferramenta poderosa usada para testar e analisar a qualidade do código. Ela fornece uma ampla gama de recursos que ajudam os desenvolvedores a identificar e corrigir potenciais problemas cedo no processo de desenvolvimento. A ferramenta realiza análise de código estático para detectar *bugs*, vulnerabilidades, além de fornecer *insights* sobre a complexidade e confiabilidade do código. O *SonarQube* é uma ferramenta crucial no desenvolvimento de *software* moderno, pois ajuda a garantir que a base de código seja segura, escalável e mantida, o que é essencial para fornecer *software* de alta qualidade.[1]

Apesar de sempre ser útil, pouco conseguimos concluir através da utilização desta ferramenta, apenas nos identificou alguns *code smells* que ao fim ao cabo nem o são, apenas são más práticas, como duplicação de código e *casts* sem necessidades a objetos, e.g. figura 6.1

1. Mili Ali, *Mastering Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers*, Packt Publishing (2021), Accessed: 05/05/2023.

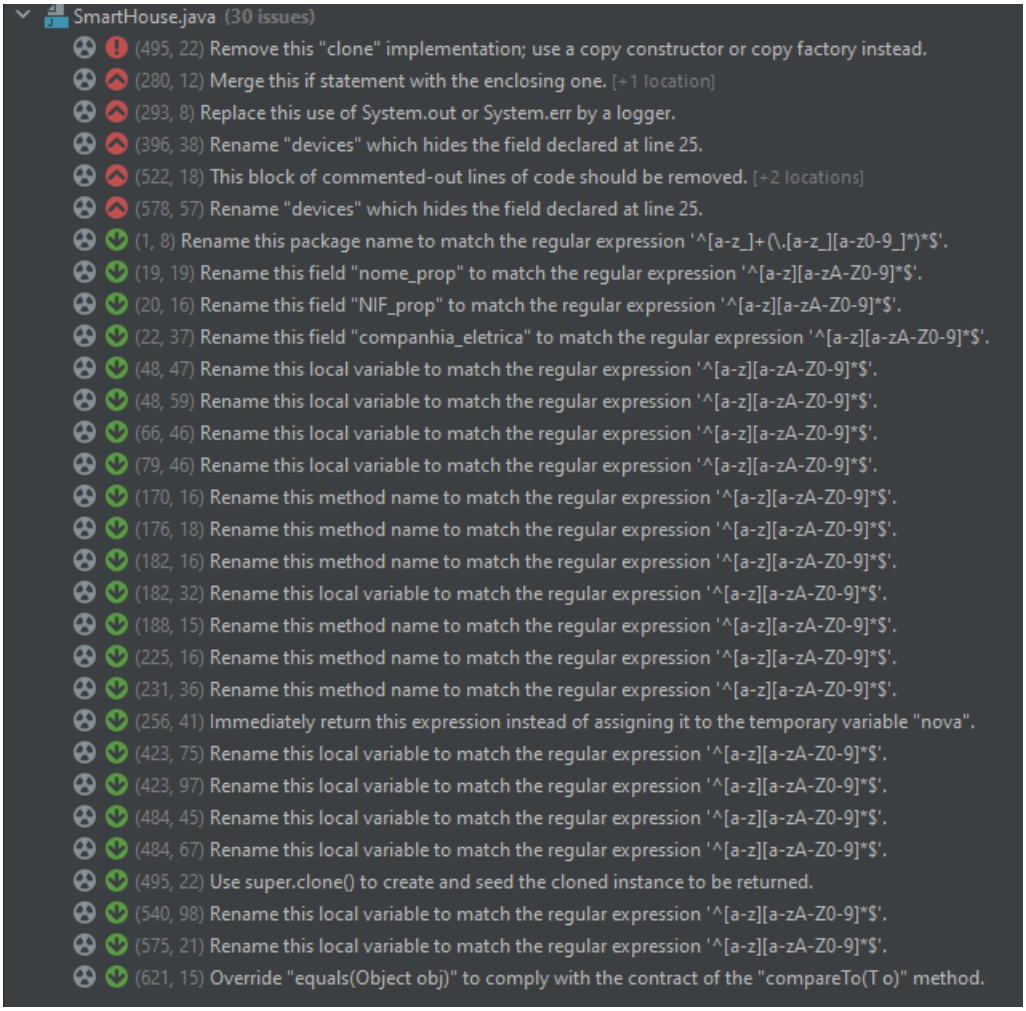


Figura 6.1: *SmartHouse SonarQube Feedback*

6.2 *Hypothesis*

Da mesma forma como foi mencionado na secção do *QuickCheck*, a geração de dados de teste de forma automática é crucial para bons resultados no que toca a testagem. Sendo assim ***Hypotesis*** é apenas uma outra versão desta biblioteca de testes, mas para *python*. Todos os procedimentos e cuidados realizados e mencionados *a priori*, mantêm-se nesta mesma secção, de forma menos complexa porque a linguagem em questão assim o facilita.[2] Apesar de ser uma linguagem mais fácil, a criação dos geradores tornou-se um processo mais lento comparativo aos outros uma vez que não é uma biblioteca que tivessemos explorado muito anteriormente.

2. David R. Maclver, *What you can generate and how*, https://hypothesis.readthedocs.io/en/latest/data.html, Accessed: 08/05/2023.

7 Conclusão

Em suma, conseguimos verificar que o projeto que decidimos utilizar de POO, obtinha alguns *bugs* ao longo da sua extensão, estando alguns deles agora tratados, e outros como identificadores da existência para a análise crítica de tal.

Apesar de extensa, é notável que a criação de testes unitárias é a que se demonstra mais eficiente a nível de corrigir e identificar pequenos erros ao longo do *software* uma vez que tem que ser o próprio utilizador a escrever tanto o teste como a *feature*, daí o seu porém de puder ter um extenso periodo até a sua identificação. Nesse ponto os mutantes tomam um papel crucial no que se troca na análise de código.

Contudo, a automação de todo o trabalho, poupando bastante tempo, consegue trazer a mesma, se não mais cuidados consoante o pretendido na análise do código. Reconhecemos assim que as diferentes *frameworks* e *plugins* de automação que foram utilizados acabam por ter um balanço muito mais positivo do que propriamentes os testes feitos pelo ”programador”.

Como é visível ao longo deste documento há secções que faltam, i.e. indo de acordo ao solicitado no enunciado do trabalho prático, e.g. *evoSuite*, *Property Based Testing*. Isto uma vez que o projeto de POO que está a ser utilizado foi criado na versão de **Java 18** faz com que estas ferramentas ou *frameworks* não possuam um suporte para esta mesma versão. Uma vez que a alteração do projeto inicial, ia remover uma grande parte da integridade inicial do projeto decidimos não o fazer e optamos por fazer outras secções extras.

Finalizando, consideramos que o grupo esteve de acordo o esperado face ao grau de dificuldade apresentado para este projeto e consoante o lecionado e aprendido na UC.

Referências

Ali, Mili. *Mastering Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers*. Packt Publishing (2021). Accessed: 05/05/2023.

Antony. *Is it OK to have multiple asserts in a single unit test?* https://softwareengineering.stackexchange.com/questions/7823/is-it-ok-to-have-multiple-asserts-in-a-single-unit-test. Accessed: 04/05/2023.

Kirk, Matt. *Why PIT?* https://pitest.org/. Accessed: 04/05/2023.

Maclver, David R. *What you can generate and how*. https://hypothesis.readthedocs.io/en/latest/data.html. Accessed: 08/05/2023.