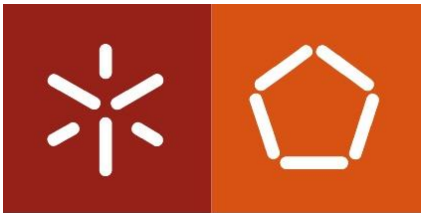


Universidade do Minho

Departamento de Informática


# Análise e Teste de Software
# Trabalho Prático

17 de maio de 2023


Afonso Magalhães
(A95250)

Rui Armada
(A90468)

Diana Teixeira
(A97516)

# Conteúdo

# Capítulo 1

# Introdução

Para a realização deste trabalho prático foram desenvolvidos vários testes e utilizadas várias ferramentas para análise de robustez de código. Após uma breve análise dos projetos realizados na disciplina de Programação Orientada a Objetos, optou-se por se tomar partido de duas soluções distintas: o **SmartHouse_1** e o **SmartHouse_2**, ambos relativos ao ano letivo 2021/2022, cujo objetivo consistia em desenvolver uma aplicação capaz de gerir e monitorizar *SmartDevices* de uma dada habitação.

A inclusão de múltiplos elementos de monitorização e de execução neste trabalho levou ao desenvolvimento de muitas classes e funções que proporcionam a necessidade de diversos testes para assegurar o bom funcionamento do programa.

Decidiu-se utilizar duas soluções distintas da mesma aplicação com o intuito de comparar os resultados obtidos entre dois trabalhos: um que, inicialmente, permaneceu inalterado e outra solução na qual foram aplicados os conhecimentos de *Unit Testing*, assim como outras ferramentas lecionadas na unidade curricular. Ter duas soluções também facilita o contorno de algumas dificuldades, como a utilização de diferentes versões do JAVA que não são compatíveis com uma solução em específico em algumas ferramentas. Esta estratégia ajudou também na realização de todos os requisitos mínimos assim como alguns extras propostos pelo enunciado.

# Capítulo 2

# Testes Unitários

## 2.1 SmartHouse_1

Nesta primeira fase do projeto, decidiu-se efetuar alguns testes unitários mais generalistas para ser possível verificar o bom funcionamento de algumas funções vitais para o modelo lógico da aplicação que foi resolvida. Com isto foram determinados os pontos vitais da aplicação, onde os primeiros testes seriam desenvolvidos de modo a garantir que todos os requisitos de uma aplicação deste género deveria cumprir. Com isto, as classes identificadas como vitais são as seguintes:

- Casa
- Fornecedor
- SmartDevice
- EstadoPrograma

### 2.1.1 Casa

A classe `Casa` possui métodos responsáveis pela alteração dos estados dos *SmartDevices*. Esta alteração pode alterar um estado de cada vez, de todos os aparelhos num determinado quarto ou até mesmo todos os aparelhos de uma habitação. Para tal, foram implementados métodos de teste para as funções `setAllDevicesStateRoom`, `setAllDevicesState` e `setDeviceState`, de modo a garantir que os estados dos *SmartDevices* são alterados corretamente. Adicionalmente, foi testado o método `consumoDispositivos`, responsável pelo cálculo do consumo total de todos os dispositivos presentes numa habitação.

Além disso, cada `Casa` inclui uma lista de faturas emitidas que detalham o período de consumo referente à mesma, o consumo energético total nesse período e o custo monetário resultante. De modo a garantir que os cálculos de consumo em função do tempo estão a ser efetuados eficientemente, foi construído um teste unitário para verificar o bom funcionamento do método `consumoPeriodo` de duas formas diferentes:

1. É especificado o período específico que se pretende obter o consumo
2. É considerado o instante de emissão da fatura até ao tempo percorrido `LocalDate`

Finalmente, optou-se ainda por testar dois métodos presentes na classe `Casa`. Ambos estes métodos são utilizados no projeto original para facilitar a manipulação dos diferentes quartos em cada casa, algo que não é necessariamente realista, mas útil para testar *setups* distintos. Estes métodos são `mudaDeviceRoom` e `juntaRooms`, que alteram um dispositivo de uma divisão para outra e agregam divisões distintas para que sejam identificadas pelo mesmo nome, respetivamente.

### 2.1.2 Fornecedor

A principal função dos fornecedores envolve a criação e emissão de faturas. Como já foi mencionado, estas faturas contêm um conjunto de informações que são essenciais para o funcionamento do programa. Dito isto, foi testado o método `criarFatura` presente na classe fornecedor, com o intuito de garantir que os parâmetros definidos nas faturas são registados corretamente. Estes parâmetros incluem:

- Data de Início
- Data de Fim
- Fornecedor
- NIF do cliente
- Consumo

Adicionalmente, verificou-se o bom funcionamento do método `faturacao` que, em função do conjunto de casas para as quais um fornecedor fornece energia calcula o valor monetário faturado por um dado fornecedor num dado período.

### 2.1.3 SmartDevice

A principal particularidade da classe *SmartDevice* baseia-se na necessidade de calcular o consumo dos três possíveis aparelhos de forma distinta. Para se testar estes consumos, foi implementado um método que testa a função `Consumption` para as *SmartBulbs*, as *SmartCameras* e os *SmartSpeakers* para verificar se estes *Devices* são adicionados ao programa correta para garantir que cada habitação possui a informação necessária para identificar cada dispositivo constituinte de uma determinada divisão.

### 2.1.4 EstadoPrograma

Para concluir este primeiro ponto, foram implementados métodos de teste para a classe *EstadoPrograma*. Esta classe é responsável pelo cálculo das estatísticas sobre o estado do programa, estas sendo:

- Qual é a casa que mais gastou naquele período?
- Qual o comercializador com maior volume de faturação?
- Listar as faturas emitidas por um comercializador.
- Ordenar os maiores consumidores de energia durante um período a determinar.

Portanto, os métodos que foram testados são:

- `getCasaMaisGastadora`
- `maiorConsumidorPeriodo`
- `getFornecedorMaiorFaturacao`
- `podiumDeviceMaisUsado`

O processo utilizado para testar estes métodos é semelhante na maior parte dos casos. Numa fase inicial, começou-se por inicializar alguns elementos que vão influenciar o resultado da estatística pretendida. Estes elementos podem variar entre *Faturas*, *Fornecedores*, *SmartDevices* e *Casas*. Em seguida, colocaram-se estes elementos nos seus locais respetivos, adicionando *SmartDevices* em casas, designando fornecedores a casas, aplicando o custo às faturas, etc. Assim que a preparação se deu como concluída, foi iniciada uma nova instância da classe *EstadoPrograma*, garantindo que o resultado da estatística aplicada aos elementos adicionados corresponda ao esperado.

## 2.2 SmartHouse_2

Respetivamente à segunda solução que foi utilizada, é importante referir que já se encontravam alguns testes unitários desenvolvidos do ano letivo anterior, alterando-se apenas algumas secções de código de forma a que a versão de JAVA utilizada pela solução correspondesse com o SDK JAVA 1.8.

Assim, tal como referido anteriormente, nota-se que este projeto foi apenas desenvolvido para utilizar o *plug-in* do IntelIJ **Evosuite**, o qual não foi possível utilizar na solução no *SmartHouse_1*, mesmo após vários esforços e tentativas com outras versões de `Evosuite` e de JAVA. A única forma de utilizar o `Evosuite` seria fazer um conjunto de modificações relativamente extensivo a uma solução de código mais complexa e de mais difícil alteração comparando com a segunda solução utilizada. É de notar que também se aplicou o **PITest** à comparação e análise dos resultados das ferramentas de análise e teste de *software* relativos à cobertura e à robustez dos testes utilizados durante este projeto.

Portanto, as classes que são testadas nesta solução são:

- SmartSpeaker
- Casa Inteligente
- Invoice
- Energy Supplier
- SmartBulb

# Capítulo 3

# Geração Automática de Testes com EvoSuite

Com o intuito de garantir o bom funcionamento do `EvoSuite`, tal como foi mencionado anteriormente, optou-se por utilizar a solução *SmartHouse_2*, devido a problemas que surgiram devido à elevada complexidade da primeira solução. Estes problemas acabaram por gerar conflitos de versões do JAVA que não possibilitaram a utilização da ferramenta desejada.

Com isto, de modo a testar a eficiência desta ferramenta, foram analisados os valores de *Coverage* dos testes que estavam presentes originalmente, em comparação com os valores obtidos após a geração automática de testes efetuada pelo `EvoSuite`. Como foi referido anteriormente, os testes presentes nesta solução da aplicação, foram realizados aquando o desenvolvimento da mesma. Visto isto, para possibilitar uma melhor compreensão do funcionamento e resultados produzidos pelo `EvoSuite`, estes testes permaneceram inalterados durante a total duração deste projeto prático.

De seguida foram construídos alguns gráficos de forma a comparar os resultados obtidos pela utilização do `EvoSuite` e pela análise dos testes originais com o recurso à ferramenta `PITest`. Estes gráficos podem ser encontrados na secção de Anexos, Figura 6.1.

Portanto, ao comparar os testes já presentes na solução com aqueles gerados pelo `EvoSuite`, viu-se que os testes automatizados apresentam uma cobertura mais abrangente do código e possuem uma capacidade de deteção de falhas muito mais eficiente e precisa. Tendo isso em conta, pode-se concluir que o `EvoSuite` consegue identificar casos de teste que podem não ter sido considerados inicialmente ou que poderiam ter sido negligenciados durante o desenvolvimento da aplicação. A quantidade e qualidade considerável dos testes gerados é razão suficiente para justificar a utilização da ferramenta, pelo que foram obtidas percentagens elevadas de *Line Coverage* e *Mutation Coverage* que comprovam a existência de testes capazes de abranger o código de tal forma a evitar alterações maliciosas de código e assegurar o bom funcionamento do serviço desenvolvido.

Em suma, com base nos resultados obtidos, pode-se concluir que o `EvoSuite` é uma ferramenta eficiente, tendo em conta que a sua abordagem automatizada não só é capaz de aprimorar a qualidade dos testes, como também poupar tempo e esforço aos desenvolvedores de *Software*.

# Capítulo 4

# Mutação de Código

## 4.1 Resultados iniciais

Com o intuito de avaliar a cobertura dos testes unitários mencionados anteriormente, optou-se por utilizar o *plugin* de **PITest** do InteliJ nas quatro classes vitais identificadas na secção 2.1.

Após a análise dos resultados, é possível aferir que, na maior parte das classes, o *Test Strength* obtido é bastante positivo. Isto por si só induz que os testes unitários inicialmente adicionados são eficazes em validar o comportamento expectável do programa. Contudo, tal como se pode observar na Tabela 4.1, a *Line Coverage* e a *Mutation Coverage* possuem percentagens demasiado baixas, o que, por sua vez, indica que o programa não se encontra capaz de detetar pequenas falhas e alterações ao código original.

| | Line Coverage | Mutation Coverage | Test Strength |
|---|---|---|---|
| Casa | 50% | 45% | 79% |
| EstadoPrograma | 27% | 17% | 93% |
| SmartDevice | 24% | 8% | 50% |
| Fornecedor | 33% | 32% | 92% |

Tabela 4.1: Resultados obtidos ao utilizar o PITest

## 4.2 Alterações efetuadas

Face a isto, com o objetivo de melhorar esta vertente, chegou-se à conclusão de que seria necessário implementar uma nova vaga de testes, com um maior foque na deteção de mutantes, visto que dessa forma poder-se-ia melhorar a eficiência geral dos testes presentes na solução.

Optando por realizar uma nova vaga de testes unitários mais concisos e minuciosos, a fim de garantir que todos os métodos das quatro classes, mencionadas na secção 2.1, possuem alguma cobertura e apresentem uma maior robustez face a alterações introduzidas pelas mutações do `PITest`, foram criados mais testes direcionados a testar os *getters*, *setters*, *equals*, *toStrings*, construtores, etc.

Dito isto, foram adotadas as seguintes estratégias:

- Múltiplos testes para um só método, onde cada um tem em consideração possíveis comportamentos diferentes do método em questão. Por exemplo, considerar o comportamento de uma das estatísticas da classe EstadoPrograma onde múltiplos elementos possuem o mesmo "Maior valor"
- Testar métodos abstratos considerando cada classe que o implementa
- Testar todos os tipos de construtores, sejam parametrizados e consequentemente clonados
- Garantir que todos os métodos possuam pelo menos um teste que avalie o seu comportamento, certificando que mutantes inseridos em métodos simples são facilmente detetáveis

## 4.3 Resultados finais

Portanto, após adicionar a nova vaga de testes foi utilizado mais uma vez o `PITest` para observar os resultados dos testes. De forma a verificar se o processo de geração de novos testes foi satisfatório, foram construídos os seguintes gráficos que comparam os resultados originais com os resultados após a inserção de novos testes.Evosuite

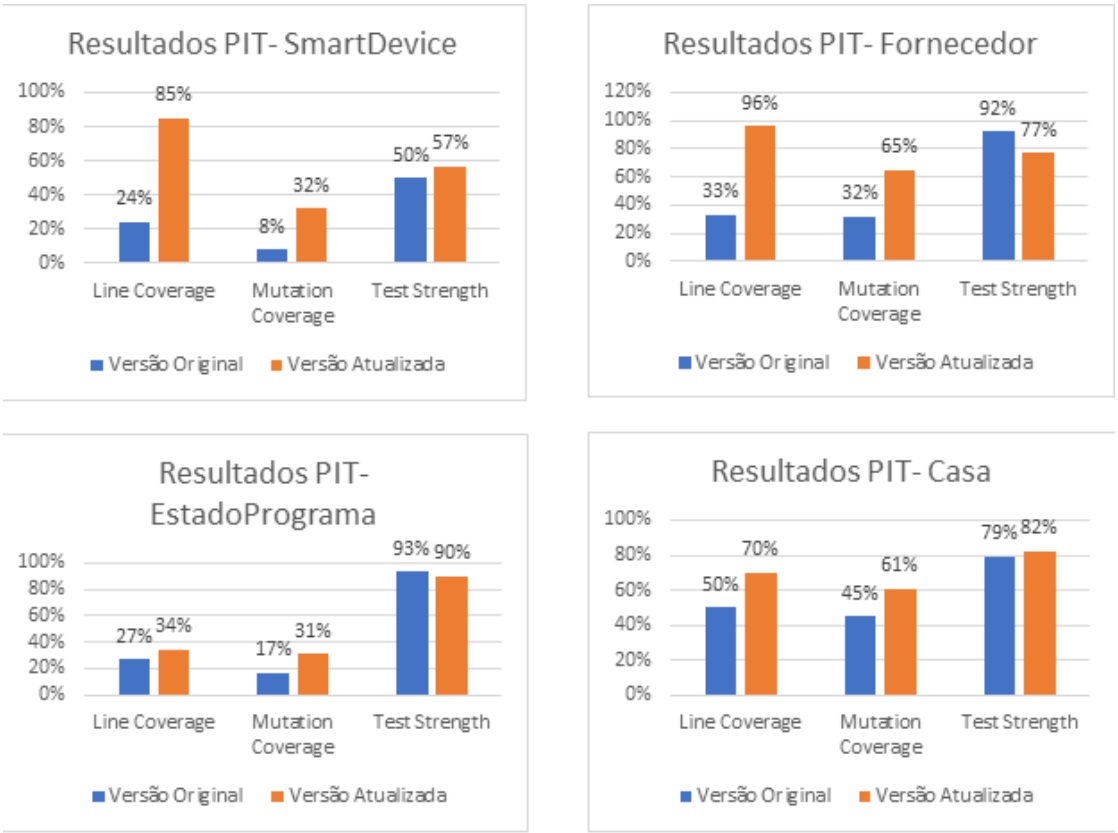


Figura 4.1: Comparação dos resultados relativos ao PIT.

Analisando os resultados obtidos pela execução do *plugin* do `PITest`, é possível ver que, após a introdução de uma nova vaga de testes unitários, algo que melhorou significativamente em todos os casos foi a *Line Coverage* e a *Mutation Coverage*. Este resultado é a consequência direta do aumento do número de testes em cada classe, visto que a maior abrangência da nova cobertura leva a que as diferenças subtis introduzidas pelos mutantes gerados no `PIT` sejam detetadas com mais frequência. Apesar disso, é possível ver que no parâmetro *Test Strength* os valores tendem a diminuir, estagnar ou subir de forma insignificante. Crê-se que esta flutuação dos valores se deve à valorização da quantidade de testes ao invés da qualidade dos mesmos pela parte da equipa de trabalho. É de notar que na classe `EstadoPrograma` não se desenvolveram tantos testes como o pretendido; como se trata de uma classe que junta diversas outras classes para testar os seus métodos, garantir uma abrangência elevada seria um processo demorado.

Tendo isto em conta, tornou-se notório que a utilização do `PITest` ajudou bastante na compreensão e análise da qualidade dos testes construídos originalmente. Por sua vez, esta ferramenta também auxiliou o estudo do código, com recurso aos relatórios detalhados que esta gera, assim como a consequente construção de testes capazes de captar e lidar com mutantes, o que tornou a aplicação mais robusta e segura contra possíveis alterações maliciosas do código base.

# Capítulo 5

# Geradores de Logs

De acordo com os requisitos do enunciado do presente trabalho prático e com base no conhecimento obtido das aulas da unidade curricular, foram utilizados o `Quickcheck` e o sistema `Hypotesis` para gerar ficheiros de *log* que fossem não só semelhantes aos fornecidos para a unidade curricular *Programação Orientada a Objetos*, como também possíveis de aplicar nas soluções utilizadas. Para isto, foi necessário criar um ficheiro em Haskell e outro em Python capazes de gerar um ficheiro com o seguinte formato:

```
Fornecedor:EDP Comercial
Fornecedor:Galp Energia
Fornecedor:Iberdrola
Fornecedor:Endesa
Fornecedor:Gold Energy
Fornecedor:Coopernico
Fornecedor:Enat
Fornecedor:YIce
Fornecedor:MEO Energia
Fornecedor:Muon
Fornecedor:Luzboa
Fornecedor:Energia Simples
Fornecedor:SU Electricidade
Fornecedor:EDA
Casa:Vicente de Carvalho Castro,365597405,Iberdrola
Divisao:Sala de Jantar
SmartBulb:Warm,11,4.57
SmartBulb:Neutral,12,4.73
Divisao:Sala de Jantar 1
SmartBulb:Neutral,7,9.35
SmartCamera:(1280x720),65,3.84
SmartSpeaker:2,Radio Renascenca,LG,5.54
SmartBulb:Neutral,5,6.36
```

## 5.1 Quickcheck

Em primeiro lugar foi necessário desenvolver um ficheiro em Haskell capaz de gerar um ficheiro `.txt` constituído por todas as entradas de informação que seriam inseridas na aplicação. Para isto foi utilizado o `Quickcheck` de modo a ser possível gerar dados de forma aleatória, de forma a repetir a sintaxe apresentada anteriormente.

Assim, para tornar a sua realização mais fácil, optou-se por implementar as seguintes características:

- Uma função `roundTo` que arredonda os *doubles* para um certo número de casas pedidas — neste método presume-se que todos os números decimais têm apenas duas casas decimais;
- Uma função para gerar NIF'S, que gera um número entre 100000000 e 999999999, não assumindo a possibilidade de que este pode ser duplicado, visto a probabilidade de isto acontecer ser muito reduzida e não se pretender gerar muitas linhas;
- Foram também implementadas duas outras funções, uma que gera um inteiro após receber o intervalo a qual este tem que pertencer e outra que gera um *double* aleatório;
- Com isto, após serem ponderados diferentes cursos de ação, decidiu-se organizar a informação sobre dos Fornecedores, Nomes, Cores, Divisões, e outras *Strings* em listas. Estas listas contêm algumas *strings* que são retiradas aleatoriamente e inseridas no ficheiro de logs. Desta forma, é possível retirar nomes, cores ou divisões de uma forma aleatória de maneira a conseguir gerar o ficheiro de *logs* mais eficientemente.
- Assim, e com base em tudo o que foi dito, foi também possível a implementação das funções `formatRegisto`, `generateRegisto` e `generateFile` que visam formatar a informação a ser escrita nas *logs* de forma a que o resultado final seja idêntico ao que é pretendido.

Com isto, o ficheiro gerado pelo `Quickcheck` apresenta o seguinte formato:


Figura 5.1: Ficheiro gerado pelo `QuickCheck`

## 5.2 Hypothesis

Por sua vez, também foi sugerida a utilização do `Hypothesis` em Python, para construir um ficheiro de *logs*, algo semelhante ao que foi descrito na secção 5.1.

Assim, foram utilizadas algumas estratégias para tornar o processo de geração do ficheiro o mais simples e eficiente possível. As estratégias utilizadas foram as seguintes:

- Várias listas declaradas, no início do ficheiro, das quais se poderão gerar os Fornecedores, a Resolução das Câmaras, Nomes, Cores, Divisões, Marcas e Estações a partir de exemplos previamente inseridos
- A construção de funções **_strategy* que se encarregam de formatar a informação a ser inserida no ficheiro `.txt`

- Utilização de métodos como *round*, para limitar número de casas decimais (neste caso apenas duas casas decimais)
- Utilização do `@given` de forma a indicar o tamanho da amostra a ser colocada no ficheiro e para garantir a existência de inserções de linhas duplicadas

Portanto, o ficheiro gerado pelo `Hypothesis` apresenta o seguinte formato:

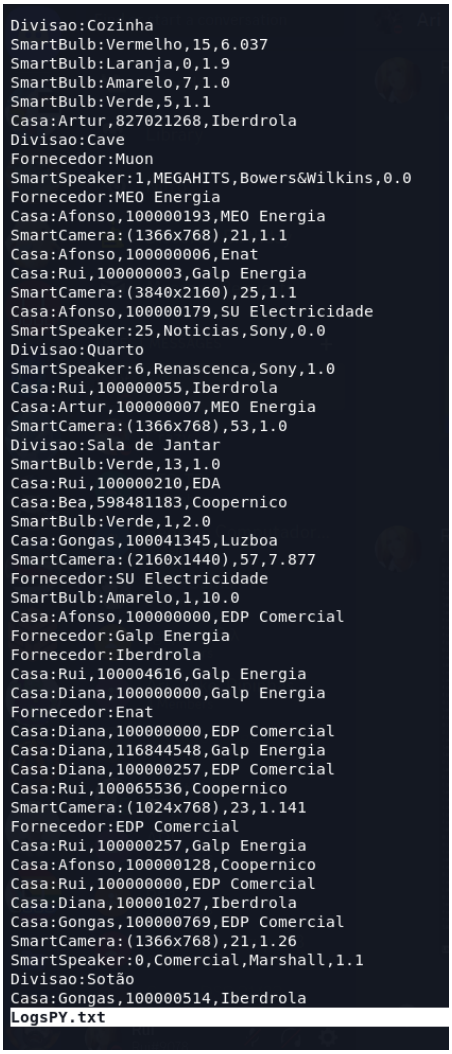


Figura 5.2: Ficheiro gerado pelo `Hypothesis`

# Capítulo 6

# Conclusão

Após a utilização de grande parte das diferentes ferramentas que foram lecionadas ao longo do semestre, pode-se concluir que as funcionalidades oferecidas por estas foram utilizadas de forma eficiente e concisa, o que levou a uma análise detalhada dos projetos mencionados para se poderem atingir os melhores resultados possiveis.

Através do uso do `PITest`, foi proporcionada uma nova perspetiva dos testes que se desenvolveu, permitindo a melhoria da sua cobertura. Por sua vez, o uso do `EvoSuite` tornou evidente a importância da geração automática de testes unitários, proporcionando uma alternativa à criação de testes de forma manual. Já o uso de `QuickCheck` e de `Hypotheis` levou à descoberta de uma forma de facilitar a criação de ficheiros de logs carregados no início da aplicação abordada.

Contudo, não foi possível utilizar a ferramenta `SonarQube` de forma correta, tendo em conta que não se conseguiu completar a sua configuração inicial. Ainda assim, foi fornecida a pasta gerada pela ferramenta juntamente com os ficheiros de entrega do trabalho prático.

Em suma, a equipa de trabalho atingiu todos os objetivos e metas espetáveis durante o desenvolvimento da solução do trabalho prático. Quanto a dificuldades durante o projeto, a equipa deparou-se com alguns problemas provenientes do `EvoSuite` funcionar apenas com o `JDK8`, visto que esse nível de linguagem JAVA não suportava muitas expressões de sintaxe utilizadas frequentemente pelos membros da equipa de trabalho, especialmente no *SmartHouse_1*, o que levou à utilização do *SmartHouse_2* em alternativa.

## 6.1 Anexos

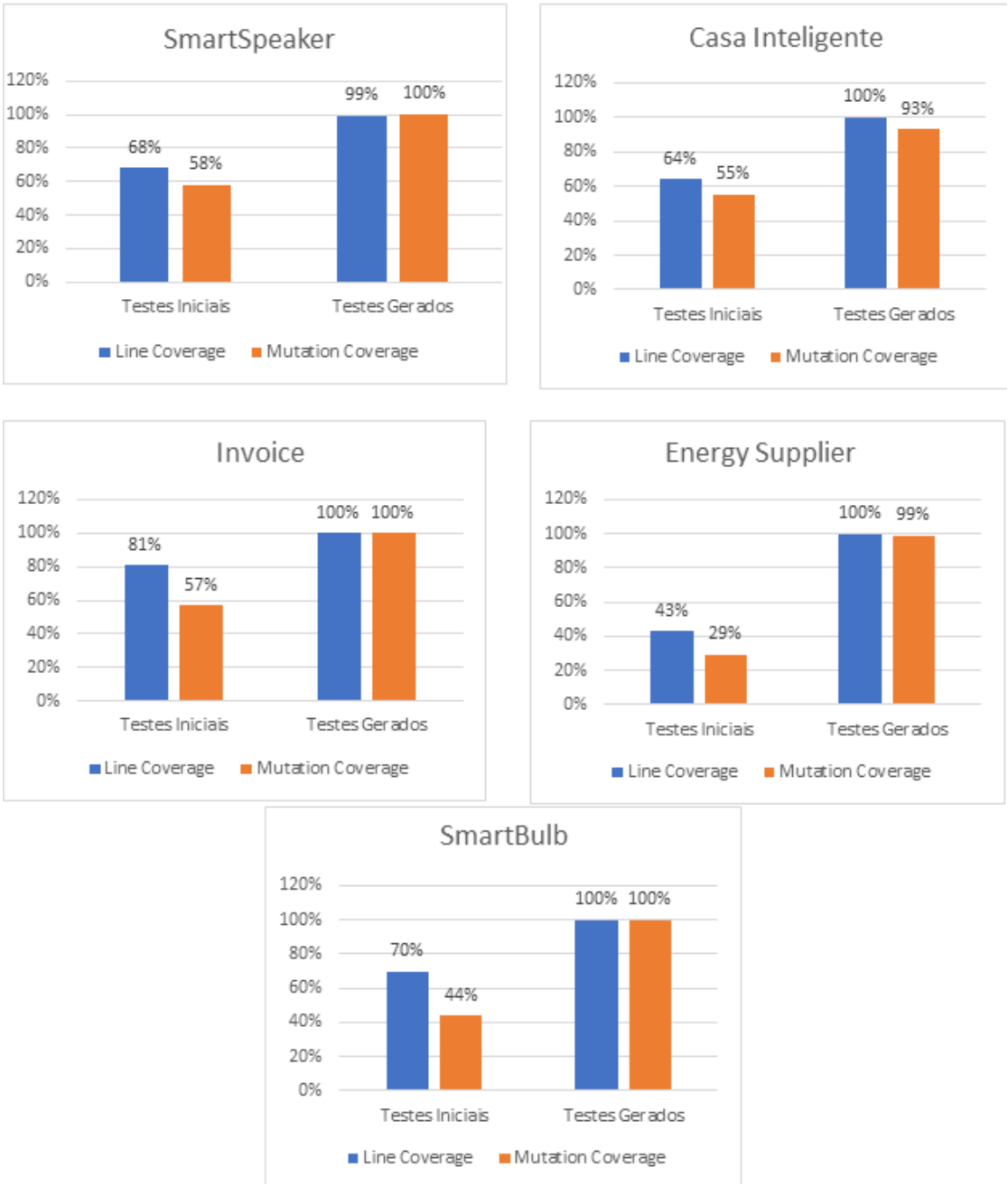


Figura 6.1: Comparação dos resultados obtidos entre os testes originais e os testes do *EvoSuite*.