INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Model-Based Testing - Automação de testes com base em modelos

MANUELA MARIA FERREIRA DA COSTA SOUSA

Outubro de 2018

POLITÉCNICO
DO PORTO

Model-Based Testing - Automação de testes com base em modelos

1121003, Manuela Maria Ferreira da Costa Sousa

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Gráficos e Multimédia

Orientador: Alberto Sampaio

Porto, outubro 2018

RESUMO

Com o surgimento de cada vez mais empresas de desenvolvimento de software, a preocupação com a realização de entregas com maior qualidade é cada vez maior. A procura pela metodologia ideal, que permita conjugar o desenvolvimento de software e a realização de testes de forma mais otimizada possível, faz com que seja necessário estudar a prática de novas metodologias que apresentem uma melhoria significativa na implementação dos processos de desenvolvimento e testes.

Esta tese de mestrado teve como principal objetivo estudar a técnica do *Model-Based Testing* que corresponde a uma técnica avançada de realização de testes. No estudo foram analisadas as notações que existem para modelação, as ferramentas, entre outros pontos. Foi também apresentado um caso prático onde a técnica foi adaptada a diferentes cenários de trabalho reais. O *Model-Based Testing* é uma metodologia relativamente recente. Apresenta uma forma de realização de testes, através da utilização de um modelo que permite gerar de forma automática, casos de teste. Dá resposta a diferentes necessidades dentro de um processo de desenvolvimento e realização de testes. Permite reaproveitar todos os artefactos utilizados, atribuindo outro significado ao tempo gasto com a realização dos testes.

Palavras-chave: *Model-Based Testing*, modelos, notações, ferramentas, testes, desenvolvimento de software.

ABSTRACT

With the emergence of more and more software development companies, the concern of delivering higher quality software is greater than ever. The search for the fitting methodology, that allows to optimize the software and test development, test plan and execution, makes it necessary to study the practice of new methodologies that present a significant improvement in the implementation of development and testing processes.

This master's thesis has as the main objective to study the Model-Based Testing technique that corresponds to an advanced test implementation technique. In this study were investigated the notations that exist for modeling, the tools, among other details. It is also presented a practical case where the technique has been adapted to different real work scenarios. Model-Based Testing is a relatively recent methodology. It presents an implementation of tests through the use of a model, that allows to automatically generate test cases. It responds to different needs within the process of developing and conducting tests. It allows reusing all the artifacts used, assigning another meaning to the time spent with the tests.

Keywords: Model-Based Testing, models, notations, tools, testing, software development

AGRADECIMENTOS

Gostaria de agradecer à minha família e amigos por todo o apoio prestado. E expressar os meus profundos e sentidos agradecimentos ao professor Doutor Alberto Sampaio, por ter sugerido este tema. E por toda a sua orientação prestada, durante a realização desta tese. Sem isso não se tornado possível a sua realização.

ÍNDICE

RESUMO I

ABSTRACT III

AGRADECIMENTOS V

LISTA DE FIGURAS X

LISTA DE TABELAS XII

LISTA DE TRECHOS DE CÓDIGO XV

NOTAÇÕES E GLOSSÁRIO XVI

CAPÍTULO 1 1

INTRODUÇÃO 1

1.1 ESTRUTURA DO DOCUMENTO 1
1.2 CONTEXTO 1
1.3 OBJETIVOS 2

CAPÍTULO 2 3

MODEL BASED TESTING 3

2.1 TÉCNICA DO *MODEL-BASED TESTING* 3
2.2 OBJETIVOS 5
2.2.1 Melhoria na eficácia *6*
2.2.2 Melhoria na eficiência *6*
2.3 PRINCIPAIS ATIVIDADES DO PROCESSO DE TESTES COM MBT 7
2.3.1 Criar modelos aplicando a técnica do Model-Based Testing *7*
2.3.2 Aplicar critérios de seleção de casos de teste *7*
2.3.3 Gerar casos de teste com ferramentas que suportam o Model-Based Testing *8*
2.3.4 Execução de casos de teste com Model-Based Testing *8*
2.4 LIMITAÇÕES DO *MODEL-BASED TESTING* 8
2.4.1 Incorreta utilização ou nenhuma de técnicas de elaboração de testes *9*
2.4.2 Não basta fazer uso da ferramenta Model-Based Testing *9*
2.4.3 Os modelos podem conter erros na sua construção *9*
2.4.4 Identificação de casos de teste em demasia *9*

CAPÍTULO 3 11

NOTAÇÕES E ALGORITMOS **11**

3.1 TIPOS DE NOTAÇÕES 11

3.2.1 Notação de Fluxogramas (Flowcharts) *12*

3.2.2 Notação de Tabelas de decisão (Decision tables) *17*

3.2.3 Notação de Máquina de estados finitos (Finite state machine) *21*

3.2.4 Notação de Redes de Petri (Petri nets) *26*

3.2.3 Notação de Redes de Petri orientadas a eventos (Event-Driven Petri nets) *28*

3.2.4 Notação de Statecharts *30*

3.2.5 Notação UML *32*

3.2.6 Notação BPMN (Business Process Modeling and Notation) *39*

CAPÍTULO 4 .... **47**

FERRAMENTAS MODEL BASED TESTING **47**

4.1 FERRAMENTAS EXISTENTES 47

4.2 COMPARAÇÃO ENTRE FERRAMENTAS 48

4.3 SELEÇÃO DE FERRAMENTA A UTILIZAR NO CASO PRÁTICO 53

4.4 FERRAMENTA SELECIONADA EM DETALHE 65

4.4.1 Funcionalidades do GraphWalker *65*

4.4.2 Regras de modelação *67*

4.4.3 Geradores e condições de paragem *69*

4.4.4 Anotações *71*

4.4.5 Plugins *71*

CAPÍTULO 5 .... **73**

COMPONENTE PRÁTICA **73**

5.1 METODOLOGIA UTILIZADA 73

5.1.1 CENÁRIO PRÁTICO UTILIZANDO WEBSITE 75

5.1.1.1 Criação de projeto com GraphWalker *75*

5.1.1.2 Verificação da criação correta do modelo *79*

5.1.1.3 Criação de classe de execução de testes *83*

5.1.1.4 Implementação dos testes automatizados *85*

5.1.1.5 Configuração do projeto no Jenkins *85*

5.1.2 CENÁRIO PRÁTICO UTILIZANDO UMA APLICAÇÃO MÓVEL 86

5.1.2.1 Criação do projeto *86*

5.1.2.2 Verificação da criação correta do modelo *88*

5.1.2.3 Criação da classe de execução de testes *90*

5.1.2.4 Implementação dos testes automatizados ... 90

5.1.2.5 Necessidades adicionais para implementação dos testes automatizados utilizando aplicação 90

CAPÍTULO 6 ... 95

ANÁLISE RESULTADOS ... 95

6.1 ANÁLISE DE RESULTADOS CASO PRÁTICO ... 95

6.2 ANÁLISE DE RESULTADOS DO CASO PRÁTICO UTILIZANDO SERVIÇOS WEB ... 99

6.3 COMPARAÇÃO *MODEL-BASED TESTING* COM METODOLOGIA *BEHAVIOR-DRIVEN DEVELOPMENT* ... 100

6.3.1 Principais vantagens e desvantagens da metodologia MBT em relação à metodologia BDD 101

CAPÍTULO 7 ... 103

7.1 CONCLUSÃO ... 103

7.2 TRABALHO FUTURO ... 105

REFERÊNCIAS ... 106

ANEXO A ... 111

ANEXO B ... 115

ANEXO C ... 118

ANEXO D ... 119

LISTA DE FIGURAS

Figura 1 - Variante de diagrama de atividades de um processo de realização de uma encomenda online
Figura 2 - Fluxograma do processo de compra numa loja física
Figura 3 - Diagrama de transições de estados de um *parser*
Figura 4 - Máquina de estados finitos de um sistema de cálculos
Figura 5 - Exemplo de uma rede de Petri
Figura 6 - Exemplo diagrama casos de uso
Figura 7 - Exemplo de diagrama de sequência
Figura 8 - Exemplo de diagrama de atividades
Figura 9 - Árvore de decisão a utilizar seguindo método *AHP*
Figura 10 - Legenda elementos do modelo
Figura 11 - Exemplo de um caminho gerado num modelo
Figura 12 - Exemplo de um modelo simplificado
Figura 13 - Resultado da criação do projeto com sucesso
Figura 14 - Implementação de métodos de forma automática a partir da interface gerada a partir do modelo
Figura 15 - Listagem dos métodos contidos pela classe da interface
Figura 16 - Modelo representativo das funcionalidades de login e pesquisa no website
Figura 17 - Classe execução de testes
Figura 18 - Execução dos testes utilizando anotação @*Test*
Figura 19 - Configurações do projeto no Jenkins
Figura 20 - Modelo representativo das funcionalidades de login e pesquisa na aplicação
Figura 21 - Configuração para início de uma nova sessão utilizando inspetor do Appium
Figura 22 - Configuração para início de uma nova sessão do inspetor do Appium Desktop utilizando a sessão corrente em execução
Figura 23 - Inspeção de elementos da aplicação utilizando inspetor do Appium Desktop
Figura 24 - Exemplo modelo construído de forma incorreta

LISTA DE TABELAS

Tabela 1 - Exemplo de um caso de teste gerado a partir do diagrama do modelo

Tabela 2 - Símbolos mais comuns da notação de fluxogramas

Tabela 3 - Casos de teste obtidos partir do fluxograma da figura 2

Tabela 4 - Exemplo da representação de tabela de decisão utilizando condições e ações em colunas separadas

Tabela 5 - Exemplo da representação de tabela de decisão utilizando condições e ações na mesma coluna

Tabela 6 - Exemplo de tabela de decisão com classes de equivalência mutuamente exclusivas

Tabela 7 - Levantamento de saldo e atribuição de crédito

Tabela 8 - Exemplo de matriz de transições de estados de um contador da máquina de vendas

Tabela 9 - Forma alternativa da matriz de transições para o contador da máquina de vendas

Tabela 10 - Tabela de transição de estados

Tabela 11 - Tabela com elementos da linguagem para eventos

Tabela 12 - Tabela com elementos da linguagem para condições

Tabela 13 - Tabela com elementos da linguagem para ações

Tabela 14 – Casos de teste obtidos a partir da figura 7

Tabela 15 – Casos de teste obtidos a partir da figura 8

Tabela 16 - Símbolos da notação BPMN

Tabela 17 - Listagem de ferramentas MBT

Tabela 18 - Comparação de características entre ferramentas

Tabela 19 - Escala de *Saaty* ou escala fundamental do método AHP

Tabela 20 - Matriz de comparação de critérios

Tabela 21 - Matriz de comparação de critérios com soma por colunas

Tabela 22 - Matriz de comparação de critérios normalizada com média

Tabela 23 - Matriz de comparação de alternativas com critério eficiência

Tabela 24 - Matriz de comparação de alternativas com critério eficiência com soma por colunas

Tabela 25 - Matriz de comparação de alternativas com critério eficiência normalizada com média

Tabela 26 - Matriz de comparação de alternativas com critério usabilidade

Tabela 27- Matriz de comparação de alternativas com critério usabilidade com soma por colunas

Tabela 28 - Matriz de comparação de alternativas com critério eficiência normalizada com média

Tabela 29 - Matriz de comparação de alternativas com critério Multiplataforma

Tabela 30 - Matriz de comparação de alternativas com critério Multiplataforma com soma por colunas

Tabela 31 - Matriz de comparação de alternativas com critério Multiplataforma normalizada com média

Tabela 32 - Matriz de comparação de alternativas com critério Escalabilidade

Tabela 33 - Matriz de comparação de alternativas com critério Escalabilidade com soma por colunas

Tabela 34 - Matriz de comparação de alternativas com critério Escalabilidade normalizada com média

Tabela 35 - Matriz de comparação de alternativas com critério Integração contínua

Tabela 36 - Matriz de comparação de alternativas com critério Escalabilidade com soma por colunas

Tabela 37 - Matriz de comparação de alternativas com critério integração contínua normalizada com média

Tabela 38 - Matriz de comparação de alternativas com critério Suporte

Tabela 39 - Matriz de comparação de alternativas com critério Suporte com soma por colunas

Tabela 40 - Matriz de comparação de alternativas com critério Suporte normalizada com média

Tabela 41 - Matriz de obtenção de prioridades compostas

Tabela 42 - Comparação de metodologias

LISTA DE TRECHOS DE CÓDIGO

Trecho de código 1 - Dependência a ser alterada para projeto assumir utilização do *GraphWalker*

Trecho de código 2 - Estrutura da classe de teste após geração das dependências necessárias

Trecho de código 3 - Classe LoginTest.java após implementação de métodos

Trecho de código 4 - Comando para validação do modelo em modo offline

Trecho de código 5 - Classe LoginTest.java com gerador *GraphWalker* embutido

Trecho de código 6 - Comando para validação do modelo em modo online

Trecho de código 7 - Resultado da validação do modelo

Trecho de código 8 - Conteúdo da classe de execução de testes

Trecho de código 9 - Resultado da execução dos testes

Trecho de código 10 - Implementação testes automatizados

Trecho de código 11 - Classe LoginTest.java após implementação dos métodos

Trecho de código 12 - Conteúdo da classe de execução de testes

Trecho de código 13 - Resultado de execução de modelo construído incorretamente

NOTAÇÕES E GLOSSÁRIO

Termo	**Descrição**
MBT	*Model-Based Testing*
SUT	*System Under Testing*
BPMN	*Business Process Modeling Notation*
UML	*Unified Modeling Language*
DSL	*Domain Specific Language*
EFSM	*Finite State Machine Extended*
PrT net	*Petri Net*
FSM	*Finite State Machine*
BDD	*Behavior Driven Development*

CAPÍTULO 1

INTRODUÇÃO

O objetivo do capítulo 1, é servir de introdução relativamente ao contexto aos objetivos a serem cumpridos com a realização desta tese de mestrado. O presente documento corresponde à entrega final da tese de mestrado desenvolvida no âmbito do mestrado em engenharia informática pelo instituto superior de engenharia do porto.

1.1 Estrutura do documento

O documento está organizado em seis capítulos. O capítulo 1, corresponde à introdução, o capítulo 2, corresponde à descrição da metodologia *Model-Based Testing*. O capítulo 3, corresponde à descrição das diferentes notações e algoritmos, utilizadas para modelar sistemas. O capítulo 4, corresponde ao detalhe relativamente às ferramentas existentes que atualmente suportam a metodologia, à comparação de algumas e à escolha de uma, utilizando o método AHP, para utilizar no caso prático. Apresenta também em detalhe as potencialidades da ferramenta selecionada. O capítulo 5, apresenta, o funcionamento da técnica do *Model-Based Testing* em conjunto com a ferramenta selecionada de forma prática. O capítulo 6, corresponde à análise dos diferentes resultados obtidos.

1.2 Contexto

O tema de realização desta tese é sobre uma técnica de realização de testes o *Model-Based Testing*. Esta técnica de realização de testes é avançada e permite, através da construção de modelos, que descrevem os aspetos normalmente funcionais de um sistema, a geração automática de casos de teste.

A técnica do *Model-Based Testing (MBT)* tem como objetivo a criação de casos de teste a partir de modelos de *software*. O modelo *Model-Based Testing (MBT)* pode ser usado desde que exista algum modelo criado. A área do *Model-Based Testing* está a crescer, mas ainda não é muito conhecida e o seu uso é limitado.

1.3 Objetivos

Os objetivos a cumprir com a realização desta tese são a realização de uma análise da relevância do *Model-Based Testing* e das suas diferentes abordagens, incluindo das ferramentas existentes que suportam o seu funcionamento. Sintetização de todo o conhecimento relacionado com o *Model-Based Testing* através da revisão sistemática da literatura e do levantamento das ferramentas disponíveis. Comparação entre as diferentes ferramentas disponíveis e abordagens para implementação da técnica do *Model-Based Testing*. Realização de uma seriação de abordagens e ferramentas disponíveis para a aplicação da técnica *do Model-Based Testing*. Implementação e apresentação de um caso de estudo para exemplificação da utilização do *Model-Based Testing*, e das ferramentas necessárias. Realização de uma descrição detalhada do processo de teste e de outros processos considerados relevantes. Apresentação de todos os resultados obtidos a partir do estudo e da sua análise crítica.

Em sumário, esta tese deverá responder a duas importantes questões:

Questão 1: Como pode ser aplicada a metodologia *Model-Based Testing* utilizando cenários de trabalho real?

Questão 2: Quais as vantagens e as desvantagens da metodologia MBT em relação à metodologia BDD?

CAPÍTULO 2

MODEL BASED TESTING

O *Model-Based Testing* (MBT) é uma técnica de realização de testes avançada que permite, através da construção de modelos, que descrevem os aspetos normalmente funcionais de um sistema, a geração automática de casos de teste [1].

O sistema a ser alvo de testes, também conhecido por *SUT - System Under Test,* durante a fase de construção dos modelos não existe. Existem diferentes linguagens de modelação disponíveis para a realização dos modelos e que são suportadas por várias ferramentas de modelação existentes. Os modelos descrevem o comportamento esperado do SUT com detalhe suficiente que torne percetível o que se vai testar e como será feito. Possuem um nível de abstração que pode variar de acordo com as especificidades de cada projeto.

Os modelos podem ser representados através de linguagens de modelação textual, modelação gráfica ou modelação textual em conjunto com modelação gráfica. A modelação textual combinada com modelação gráfica é utilizada quer para visualizar o modelo textual num diagrama quer para completar um modelo gráfico com informações textuais. Um modelo pode ser constituído por vários diagramas, estruturados e detalhados de forma diferente, que representem ou não diferentes partes do mesmo SUT.

2.1 Técnica do *Model-Based Testing*

Existem duas abordagens relacionadas com a utilização da técnica do *Model-Based Testing*, a *Online* e *Offline*. A abordagem *online* refere que a geração dos casos de teste ocorre durante a execução dos testes e a *Offline* refere que a geração dos casos de teste é feita antes da execução dos casos de teste [4].

Esta técnica de testes é normalmente enquadrada no conjunto de métodos de teste conhecido por testes de caixa preta (*black-box*) que pressupõe a elaboração dos casos de teste sem existir conhecimento da estrutura interna do sistema [2].

A aplicação da técnica do MBT envolve sempre a criação de um modelo para um sistema a ser alvo de testes e a obtenção de casos de teste a partir desse modelo. A figura 1 representa um

exemplo de como poderia ser construído um diagrama de atividades pertencente a um modelo, representante de uma parte de vendas *online* de um SUT.

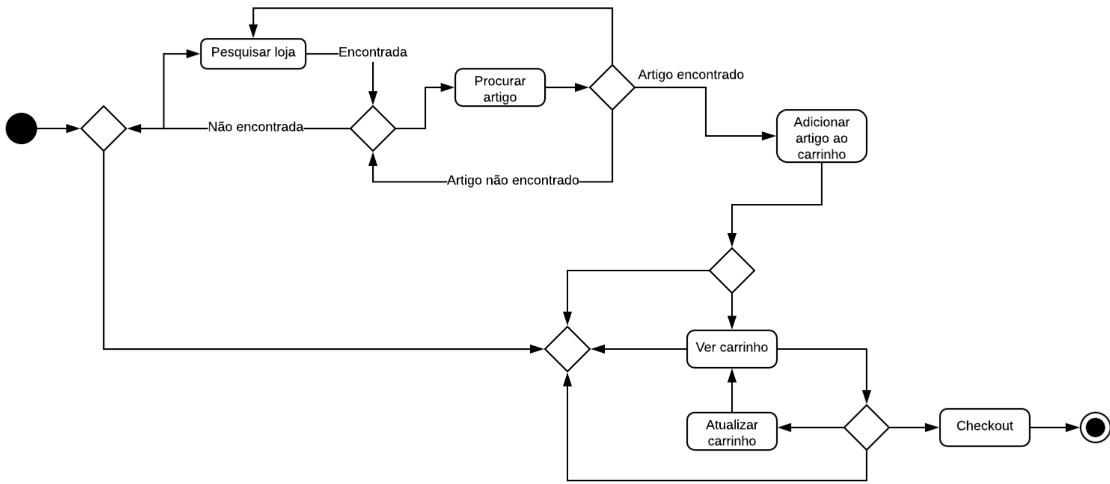


Figura 1 - Variante de diagrama de atividades de um processo de realização de uma encomenda online.

Aplicando a técnica do MBT, seguindo o caminho indicado do início ao fim e colecionando informação durante o percurso é possível obter facilmente a partir do diagrama pelo menos um caso de teste representado na Tabela 1.

Tabela 1 - Exemplo de um caso de teste gerado a partir do diagrama do modelo

Requisito: Deverá conseguir efetuar uma encomenda numa loja *online*
Caso de teste - Efetuar encomenda numa loja *online*
1. **Procurar loja** 2. **Procurar artigo** 3. **Adicionar artigo ao carrinho de compras** 4. **Ver carrinho de compras** 5. **Efetuar *checkout*** 6. **Finalizar a compra**

A técnica MBT a partir dos modelos permite, desta forma, automatizar completamente o processo de elaboração de testes e produzir fluxos de testes que posteriormente podem ser transformados em scripts executáveis [3].
Dado a técnica MBT ser incorporada no método de testes de caixa preta (*black-box)* estende e suporta as técnicas de elaboração de testes já existentes, tais como:

- **classes de equivalência (*equivalence partitioning*);**
- **valores limite (*boundary value analysis*);**
- **tabelas de decisão (*decision table testing*);**
- **casos de uso (*use case testing*);**

É possível criar modelos onde podem ser incluídos casos de teste gerados através das técnicas de elaboração de testes mencionadas ou combinar os modelos com outras representações, tais como, tabelas de decisão e outros diagramas [2].
Podemos, portanto, assumir que a técnica MBT é a automação da elaboração de testes de caixa-preta (*black-box*). Sendo a principal diferença do método tradicional para o método com a técnica do MBT a criação de um modelo que represente o comportamento esperado de partes de um sistema em vez da escrita manual dos casos de teste a partir da análise de requisitos. Após esta fase ferramentas que suportam a técnica do MBT são utilizadas para gerar automaticamente os testes a partir do modelo.
Existem várias ferramentas disponíveis atualmente no mercado que já suportam a técnica do MBT e possuem várias diferenças em relação às entradas necessárias e às saídas geradas. São ferramentas baratas, rápidas e mais precisas que a execução manual e humana [3].

2.2 Objetivos

A técnica do MBT melhora a eficácia e a eficiência da conceção, e da implementação dos testes [3]. Desta forma fornecem um melhor controlo na identificação de casos de teste a serem gerados para um projeto, permitem manter e gerir o conhecimento de um SUT [3].

2.2.1 Melhoria na eficácia

- Ajuda a gerir a complexidade, a melhorar a comunicação entre os responsáveis pelos testes e outras pessoas da organização e a facilitar o envolvimento de todos os interessados.
- Proporciona um entendimento comum e a deteção de possíveis falhas de entendimento sobre os requisitos a serem implementados.
- Suporta a melhoria contínua e a valorização dos profissionais de testes de *software* devido ao seu envolvimento antecipado na análise de requisitos e na gestão da complexidade exigida durante a criação e análise dos modelos.
- O nível de abstração do modelo permite identificar possíveis partes do sistema suscetíveis de serem alvo de falhas.
- Permite a análise e geração de casos de teste antes da existência do sistema real.

2.2.2 Melhoria na eficiência

- Modelação, validação e verificação do modelo ajuda a evitar defeitos na fase inicial do processo de desenvolvimento.
- Os artefactos dos modelos anteriores podem ser reutilizados para posterior desenvolvimento de testes.
- A técnica MBT suporta a automação, através da geração dos casos de teste, e ajuda a reduzir os defeitos que podem ser introduzidos quando os testes são definidos e especificados manualmente.
- Diferentes *suites* de testes podem ser geradas a partir do mesmo modelo, com diferentes critérios de teste, promovendo uma fácil manutenção dos testes e dos seus objetivos.
- A técnica MBT pode ser usada para vários testes com diferentes objetivos, cobrir vários níveis de testes e diferentes tipos durante a fase de testes.
- Ajuda a reduzir o custo de manutenção do desenho dos testes quando os requisitos são alterados proporcionando apenas um único ponto de manutenção.

2.3 Principais atividades do processo de testes com MBT

A técnica MBT substitui uma definição manual de casos de teste pela elaboração de um modelo abstrato de um sistema a ser validado. Pressupõe o uso de uma ferramenta que suporta a técnica e permite, dessa forma, a geração automática de casos de teste para determinado sistema.

Integrando a técnica no processo clássico de testes obtemos uma divisão em quatro principais atividades [2]:

2.3.1 Criar modelos aplicando a técnica do *Model-Based Testing*

Os modelos facilitam o entendimento e tornam possível a identificação explícita de casos de teste a efetuar num sistema. É uma representação estruturada dos aspetos de um sistema que queremos testar, onde são descritos pontos de controlo e de verificação do seu comportamento e representados diferentes casos de uso. A informação para construção do modelo é obtida com base nos requisitos do sistema por forma a garantir a existência de coerência entre todas as partes, os requisitos, o modelo criado e os testes gerados. As informações utilizadas para construção dos modelos deverão ser suficientemente precisas e completas para que seja possível, a partir deles a geração automática dos testes onde deverão ser inclusos os resultados esperados de cada comportamento do sistema.

2.3.2 Aplicar critérios de seleção de casos de teste

Normalmente o uso da técnica MBT dá origem a um grande número ou infinito de casos de teste. Por esse motivo existe a necessidade de serem aplicados critérios de seleção de casos de teste. Durante a escolha de critérios de seleção de casos de teste para aplicar é por vezes assumido incorretamente que para haver uma correta seleção dos casos de teste deverão ser utilizados apenas critérios considerados os melhores na identificação. Na realidade o que é necessário é combinar vários critérios de seleção de casos de teste para atingir os objetivos de testes pretendidos e evitar a geração de casos de teste em demasia.

Com base nos critérios combinados deverão ser escolhidos os casos de teste mais importantes e apropriados a serem realizados no sistema, que garantam um determinado nível de cobertura de testes em determinadas partes ou funcionalidades específicas.

Existem seis tipos de critérios de seleção de casos de teste que correspondem aos mais utilizados que são, seleção de casos de teste por requisitos, por elementos do modelo, através de dados, de forma aleatória, por cenários ou padrões e por projeto.

2.3.3 Gerar casos de teste com ferramentas que suportam o *Model-Based Testing*

A geração de casos de teste é um processo automático que é realizado a partir de ferramentas que suportam a técnica do *Model-Based Testing*. A ferramenta utiliza o modelo como input, em conjunto com os critérios de seleção de casos de teste. A partir dessa informação gera os casos de teste, dados de teste e possivelmente scripts executáveis caso suporte essa funcionalidade. Os casos de teste gerados correspondem a sequências de atividades com parâmetros de entrada e valores de saída possíveis em relação a cada ação. Os testes gerados deverão ser fáceis de entender e de executar.

Quando a ferramenta suporta a geração automática de scripts executáveis a partir dos testes, para execução totalmente automática deverá ser implementada uma camada de adaptação aos testes. Esta camada deverá utilizar as funções definidas no modelo de forma a ser compatível e permitir uma execução completamente integrada com a ferramenta *Model-Based Testing* efetuando a leitura dos scripts executáveis e proporcionando dessa forma a criação automática dos testes.

2.3.4 Execução de casos de teste com *Model-Based Testing*

Os testes podem ser executados manualmente ou a partir de um ambiente que permita a sua execução automática. Os testes que falham durante a execução podem indicar uma discrepância entre o comportamento atual do sistema a ser testado e os resultados esperados descritos no modelo. Deverá ser analisada a sua causa e decidir se é uma falha no comportamento, um erro no modelo, nos requisitos ou na camada de adaptação aos testes gerados a partir da ferramenta *Model-Based Testing*. A análise de resultados de execução de testes e dos scripts é uma parte importante durante a tarefa de execução de testes. Se for possível rastrear um erro de volta ao modelo existirá um melhor contexto sobre o que poderá ter causado o problema. Ajudará a identificar mais testes que irão falhar se não estiverem de acordo com o definido e que poderão ser removidos do modelo impedindo dessa forma a sua execução desnecessária.

2.4 Limitações do *Model-Based Testing*

Apesar de apresentar vários benefícios e ser uma solução que permite automatizar os casos de testes desde a sua especificação até à sua execução a técnica do *Model-Based Testing* apresenta também algumas falsas expetativas [1].

2.4.1 Incorreta utilização ou nenhuma de técnicas de elaboração de testes

Dado ser uma forma diferente de trabalhar que requer uma grande remodelação numa organização, num processo de testes e desenvolvimento de *software* a técnica do *Model-Based Testing* é vista como uma técnica que substituirá todos os passos de testes existentes no processo clássico. Esta presunção pode levar a que não sejam sequer utilizadas técnicas de elaboração de casos de testes ou que não sejam devidamente aplicadas. Apesar da técnica do *Model-Based Testing* suportar diferentes técnicas de especificação existe sempre a possibilidade de a equipa escolher não fazer uso delas.

2.4.2 Não basta fazer uso da ferramenta *Model-Based Testing*

A técnica do *Model-Based Testing* pressupõe o uso de uma ferramenta, mas a sua aquisição não deverá ser a primeira tarefa a executar dentro de uma organização. Deverá ser analisada a decisão de introdução da técnica com base na definição de objetivos mesuráveis relacionados com melhorias no processo de testes existente na organização e a partir tomar a decisão da escolha de uma ferramenta e da mais apropriada ao sistema e às suas tecnologias.

2.4.3 Os modelos podem conter erros na sua construção

A construção dos modelos não é sempre perfeita, tal como no processo clássico de especificação de testes, durante a sua construção, podem ser introduzidos erros no seu desenho. Os modelos construídos utilizando a técnica do *Model-Based Testing* requerem uma verificação e validação completas. Qualquer alteração no modelo propaga-se para todos os dados e casos de teste gerados relacionados ao elemento do modelo modificado. Cada alteração deve ser cuidadosamente reavaliada antes da implementação.

2.4.4 Identificação de casos de teste em demasia

A construção do modelo pode levar à identificação de um número de casos de teste muito grande. Nessa situação, poderá tornar-se complicado selecionar os casos de teste mais importantes a serem executados.

CAPÍTULO 3

NOTAÇÕES E ALGORITMOS

No contexto da técnica do *Model-Based Testing* a construção de modelos é uma atividade obrigatória e essencial. Os modelos construídos permitirem a identificação e geração de casos de teste utilizando informação necessária, suficiente e adequada aos objetivos de testes definidos para um projeto. A qualidade do modelo construído tem um grande impacto na eficácia do processo de testes seguindo a técnica do MBT. Um modelo que não se encontre bem construído pode conduzir à identificação de casos de teste que não são úteis ou válidos para um projeto. Os modelos são interpretados por ferramentas e devem seguir uma sintaxe rigorosa consoante a notação de modelação utilizada para a sua construção [1].

3.1 Tipos de Notações

Os tipos de notações que permitem a construção de modelos MBT estão diretamente ligados ao tipo de diagrama que se pretende utilizar para representar a informação. Normalmente os modelos são de dois tipos, estruturais ou comportamentais [5]. Os modelos estruturais ajudam a verificar a arquitetura de um sistema, em particular a decomposição do sistema em componentes e as interfaces entre os componentes, e não descrevem aspetos dinâmicos do sistema. A linguagem que mais se adequa para a sua representação é a linguagem UML (*Unified Modeling Language*) que possibilita a construção de diagramas de pacotes para condensar as várias classes de *software* e descrever a relação entre os pacotes. Os modelos comportamentais são utilizados para testar casos de uso diretamente relacionados com regras de negócio, fluxos de trabalho (*workflows*) ou ciclos de vida de um produto, representando todas as ações possíveis do SUT. A maioria dos modelos MBT corresponde a modelos comportamentais [2].

Para alguns dos tipos de notações descritas serão apresentados pequenos casos práticos de como se podem obter casos de teste a partir da descrição existente nos diagramas das notações.

3.2.1 Notação de Fluxogramas (*Flowcharts*)

A notação de fluxogramas possui elementos e características que têm como principal objetivo ajudar-nos a entender a uniformizar e a melhorar um qualquer processo. Os fluxogramas são importantes para o desenvolvimento do pensamento orientado ao processo e possibilitam a identificação de uma sequência de passos necessária para a execução de uma determinada tarefa. A notação de fluxogramas torna os processos visíveis. Ajuda a identificar as áreas que necessitam de melhoria e através da análise e concordância sobre o funcionamento de um processo a eliminar as ineficiências. Ajuda a manter o histórico das alterações aos processos e a escrita de uma sequência de passos ajuda à sua aprendizagem e ao seu entendimento comum. Previamente à construção de um fluxograma para representação de um processo é essencial que exista conhecimento sobre qual o produto ou serviço que será representado, bem como, quais serão os seus principais intervenientes e quais as atividades que irão realizar. Um fluxograma é, portanto, um diagrama que contém uma sequência de passos que representa um processo de trabalho e é construído através de caixas ou outros símbolos que podem representar os passos ou as ações. A figura 2 representa um fluxograma de uma compra realizada numa loja física [7].

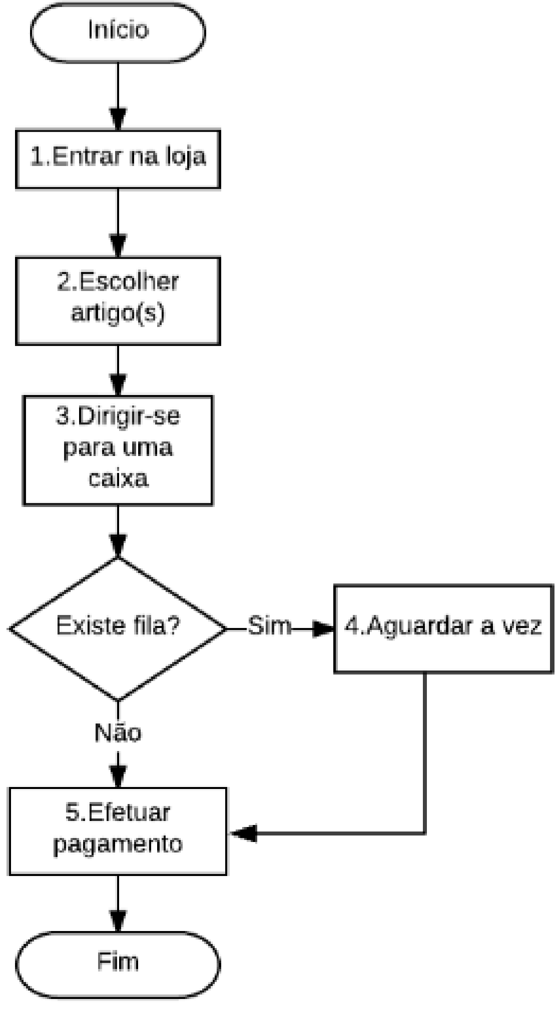


Figura 2 - Fluxograma do processo de compra numa loja física

O fluxograma da figura 2 está assinalado pelo início e fim e é constituído por caixas que representam atividades, uma decisão e por setas que indicam a sequência a ser seguida para se efetuar uma compra numa loja física.

Os diferentes passos de um processo podem ser representados através de diferentes símbolos de fluxogramas. Durante a utilização da notação é importante utilizar sempre os mesmos símbolos para permitir que outros utilizadores consigam interpretar e fazer uso dos fluxogramas. Na tabela 2 é possível visualizar alguns dos símbolos cuja utilização é mais comum e mais conhecida na notação de fluxogramas.

Apesar de os fluxogramas serem muito úteis para expor problemas eles apenas apresentam melhorias num processo se os problemas identificados forem resolvidos caso contrário não cumpriram esse objetivo. Esta tarefa normalmente envolve a recolha de informação relativamente ao problema, as causas e a identificação sobre qual o seu impacto [7].

Tabela 2 - Símbolos mais comuns da notação de fluxogramas

Símbolos	Descrição
	Os passos de início e de fim são representados através de um símbolo oval.
	O símbolo com o formato de um retângulo representa as ações.
	O símbolo com formato de um losango representa as decisões.
	A sequência de passos dos processos é representada através de setas.
	O símbolo com formato de um quadrado com borda arredondada é utilizado para representar documentos.
	O símbolo de um círculo fechado vazio é utilizado em gráficos mais complexos para conectar elementos separados numa página.
	O símbolo com formato de um retângulo com uma ponta é utilizado em diagramas complexos para conectar elementos através de múltiplas páginas com um número. Normalmente colocado em cima ou dentro do símbolo para fácil referência.
	O símbolo com formato de um retângulo inclinado representa informação disponível para funcionar como entradas e saídas e/ou recursos usados ou gerados.

3.2.1.1 Técnicas de utilização da notação

Existem algumas técnicas de construção de fluxogramas que os tornam fáceis de utilizar e de entender. A existência de um nome para identificar o processo ajuda a manter os processos criados sempre identificados. A utilização de uma data de criação ou de atualização ajudará a identificar sempre qual a versão mais recente do diagrama. É boa prática conter o nome do responsável pela criação do fluxograma para deixar claro quem deverá ser contactado para esclarecer possíveis dúvidas. Na construção do fluxograma deverá ser bem assinalado o começo e o fim do processo para ajudar outros durante a sua utilização e para estabelecer quais os seus limites. A direção de construção deverá ocorrer de cima para baixo e da esquerda para a direita para facilitar o seguimento do fluxo do processo e para facilitar a identificação de possíveis problemas. Os passos do processo deverão ser numerados para facilitar a sua identificação e referência.

Deverão ser utilizados os símbolos da notação mais comuns e mais conhecidos, durante a construção de fluxogramas para facilitar a sua interpretação por parte de outros intervenientes [7]. A construção poderá incluir o uso de setas que geralmente terminam na parte de cima ou lateral de outro símbolo. As caixas de decisão deverão conter um ponto de entrada e como uma decisão está a ser tomada duas setas de saída. A caixa de processo poderá conter várias setas de entrada, mas no máximo uma seta de saída. A informação textual no fluxograma, para além dos nomes dos processos e decisões, poderá referir-se a possíveis resultados das caixas de decisão, tais como, sim ou não ou outros valores que reflitam possíveis resultados.

Os símbolos que fazem parte da notação de fluxogramas suportam explicitamente as três construções básicas da programação estruturada a sequência a seleção e a repetição [5].

3.2.1.2 Obter casos de teste a partir de Fluxogramas

A modelação de fluxogramas em conjunto com uma sólida definição de requisitos ajuda as organizações a melhorar o seu processo de execução de testes, devido à capacidade que os fluxogramas possuem de remover redundância e testes não necessários reduzindo dessa forma tempo de execução de testes e custo de serem encontrados defeitos numa fase posterior do desenvolvimento. Ao serem modelados requisitos com fluxogramas é removida ambiguidade o que permite aos profissionais dos testes adaptar os casos de teste aos requisitos, priorizar casos de teste com base na frequência que determinada ação é executada e analisar que tipo de dados poderão ser esperados por parte do software. Quando uma mudança nos requisitos é efetuada é possível identificar através do fluxograma quais os caminhos que foram afetados por essa alteração. Dessa forma em vez de testar todo um ciclo de testes completo para garantir

que nada foi alterado juntamente com a alteração efetuada é possível identificar e efetuar nova validação apenas aos testes afetados pelo caminho alterado [21]. Outro ponto essencial da utilização de fluxogramas como ferramenta de modelação de requisitos está relacionado com a sua capacidade técnica de representação de fluxos que se repetem. A existência de fluxos que se repetem é um requisito bastante comum então a capacidade de representação dos fluxogramas destes fluxos constitui um benefício significativo [22].

Os fluxogramas são construídos diretamente a partir dos requisitos e a sua construção é constituída por diferentes tipos de caminhos para estruturar o comportamento de qualquer tipo de aplicações. Quando os fluxogramas são representados por caminhos paralelos é possível obter facilmente, a partir dos mesmos, casos de teste abstratos. Os fluxogramas são muito úteis para representar o comportamento de aplicações que envolvem muita lógica ou a realização de muitos cálculos, devido à sua capacidade em demonstrar ações e diferentes caminhos possíveis a seguir. A possibilidade de identificação de casos de teste reais não se coloca pois os fluxogramas refletem as ações que são necessárias de realizar, como por exemplo no caso de uma calculadora que demonstra os diferentes caminhos possíveis para realização dos cálculos de soma, de subtração, de multiplicação e de divisão mas não executa efetivamente os cálculos apenas demonstra que é possível realizar esses cálculos. A partir da observação do fluxograma é possível derivar os casos de teste e geralmente cada caminho possível dá origem a um caso de teste [5].

Tendo em conta a descrição efetuada relativamente à obtenção de casos de teste a partir de fluxogramas, utilizando o fluxograma da figura 2 observando os caminhos representados, em termos práticos, obteríamos os casos de teste já trabalhados descritos na tabela 3.

Tabela 3 - Casos de teste obtidos partir do fluxograma da figura 2

Casos de teste	**Passos do teste**
1. Efetuar compra de artigo(s) e pagamento numa caixa sem fila.	**1.** Entrar na loja; **2.** Escolher artigo(s); **3.** Dirigir-se para uma caixa sem fila; **4.** Efetuar pagamento;
1. Efetuar compra de artigo(s) e pagamento numa caixa com fila.	**1.** Entrar na loja; **2.** Escolher artigo(s); **3.** Dirigir-se para uma caixa com fila; **4.** Aguardar a vez para pagamento;

	5. Efetuar pagamento;

O elemento de decisão o losango que representa a tomada de uma decisão no fluxograma da figura 2, permite a identificação de dois cenários que surgem na coluna casos de teste da tabela 3. Para além da identificação dos dois cenários, é possível ainda, observando os símbolos retangulares identificar as ações, que se encontram representadas na coluna passos do teste na tabela 3, que correspondem aos passos de execução para cada um dos casos de teste. É possível também, identificar onde começa o fluxograma e onde termina, através da indicação efetuada e assinalada com o símbolo oval e pela numeração de cada passo.

3.2.1.3 Limitações da notação

A principal limitação da notação de fluxogramas está relacionada com a representação de aplicações orientadas a eventos. Dado não existirem símbolos específicos para a representação de eventos, a representação de aplicações orientadas a eventos é feita através de símbolos não apropriados, dificultando o seu entendimento e tornando-os apenas reconhecidos através do seu conteúdo semântico. Quando a aplicação orientada a eventos contém eventos que ocorrem de forma independente, a sua representação é também dificultada pela sequência inerente à construção de fluxogramas.

Outra das limitações da notação de fluxogramas está relacionada com a utilização de informação escrita. Existe pouca oportunidade de representação de informação escrita nos fluxogramas o que não permite a sua utilização de forma efetiva em casos que seja necessário expressar uma estrutura de dados e o seu relacionamento [5].

3.2.2 Notação de Tabelas de decisão (*Decision tables*)

A construção de uma tabela de decisão é constituída pela sua divisão em duas regiões principais. Uma das regiões corresponde ao conjunto de condições e a outra região corresponde ao conjunto de ações. Quando as condições são satisfeitas as ações correspondentes tem de ser executadas. Algumas representações das tabelas de decisão correspondem à utilização da região das condições acima da região das ações tal como é possível visualizar pela tabela 4,

onde a secção das condições surge do lado esquerdo superior, acima da secção das ações posicionada do lado esquerdo inferior [8].

Tabela 4 - Exemplo da representação de tabela de decisão utilizando condições e ações em colunas separadas

Condições	**Regras**				
	1	**2**	**3**	**4**	**5**
Condição 1	V	F	V	V	V
Condição 2	_	F	V	V	F
Condição 3	_	V	F	V	V
Ações					
Ação 1			X		X
Ação 2	X	X		X	

Ou a sua representação de forma horizontal com as duas regiões juntas, como é possível visualizar na tabela 5, onde o seu título é utilizado conjuntamente e as suas secções apesar de divididas se unem na tabela.

Tabela 5 - Exemplo da representação de tabela de decisão utilizando condições e ações na mesma coluna

Condições e Ações	**1**	**2**	**3**	**4**	**5**
Condição 1	V	F	V	V	V
Condição 2	_	F	V	V	F
Condição 3	_	V	F	V	V
Ação 1			X		X
Ação 2	X	X		X	

Também representados na tabela 5, estão diferentes símbolos que podem ser denominados por entradas para especificar os valores que vão existir em cada uma das regiões da tabela. É utilizado para cada região o mesmo número de colunas alinhadas através do limite comum. Cada coluna individualmente contém valores como S (Sim), N (Não) ou V (Verdadeiro), F (Falso) que correspondem a uma relação com as condições e ações chamada de regras. As regras, como exemplo, corresponderiam aos valores que se encontram na tabela 5, numerados de 1 a 5. A

entradas deixada em branco na região das condições, significa que é relevante e na região das ações que é executada. Se as condições numa tabela de decisão forem todas satisfeitas então todas as ações devem ser executadas [8].

3.2.2.1 Técnicas de utilização da notação

Existem diferentes técnicas que devem serem consideradas durante a construção de tabelas de decisão. Quando existem regras numa tabela de decisão, que contém valores de entrada idênticos, surgem condições que não tem efeito nas ações que são realizadas por essas regras. O que acontece com as regras cujos valores de entrada são idênticos é o que se chama de condensação. O seu valor é substituído por um símbolo "- "que indica que a condição não é importante ou não se aplica dando origem a uma tabela de decisão condensada.

Quando são construídas tabelas de decisão que contém condições que se referem a classes de equivalência, que não podem ocorrer ao mesmo tempo, as regras também são representadas através do símbolo "- ". Como as condições são mutuamente exclusivas não podem existir regras no qual dois valores de entrada sejam verdadeiros. Na tabela 6, é possível visualizar o exemplo referido. As condições C1 e C2 e C3 não podem ocorrer em simultâneo logo as regras 1, 2 e 3 não podem ser verdadeiras para mais do que uma entrada.

Tabela 6 - Exemplo de tabela de decisão com classes de equivalência mutuamente exclusivas

Condições	**Regras**		
	1	**2**	**3**
C1. Dia 1 do mês	**V**	**_**	**_**
C2. Dia 2 do mês	**_**	**V**	**_**
C3. Mês de Agosto	**_**	**_**	**V**
Ações			
A1. Ação 1			

Uma manutenção que não seja bem realizada pode dar origem a tabelas de decisão redundantes e inconsistentes. Para além da construção com a devida atenção da tabela de decisão, sempre que existir a necessidade de uma manutenção após a sua construção deverá ser efetuada também de forma cautelosa. Sendo este último passo essencial para evitar a existência de tabelas de decisão redundantes e/ou inconsistentes [5].

3.2.2.2 Obter casos de teste a partir de tabelas de decisão

As tabelas de decisão devem ser declarativas. Por esse motivo, tentar obter uma ordem para definir casos de teste a partir de tabelas de decisão não é considerada a abordagem mais correta. As tabelas de decisão quando são bem construídas podem permitir que todos os cenários de um determinado contexto sejam contemplados. No entanto apenas é possível perceber no imediato se alguma ação ou condição está em falta se o interveniente responsável por criar ou analisar a tabela de decisão possuir experiência e conhecimento suficiente sobre o contexto. As tabelas de decisão são mais adequadas para identificar casos de teste em aplicações que envolvem lógica e muitas decisões a partir das regras que lhe são definidas. Sendo menos apropriadas para aplicações orientadas a eventos onde a fonte de identificação de casos de teste para este tipo de aplicações, a partir da tabela de decisão, poderá ser a definição de uma sequência de regras [5].

Tabela 7 - Levantamento de saldo e atribuição de crédito

Condições	**Regras**		
	1	**2**	**3**
C1. Quantidade levantamento <= Saldo existente na conta	V	F	F
C2. Crédito concedido	_	V	F
Ações			
A1. Efetuado levantamento	V	V	F

Observando o exemplo na tabela 7, que corresponde a requisitos de negócio relacionados com o levantamento de saldo e atribuição de crédito, é possível pelo menos obter um caso de teste por coluna que cobre todas as regras de negócio:

- **Caso de teste para a regra 1**: saldo=10, quantidade levantamento=10 o levantamento é efetuado;
- **Caso de teste para a regra 2**: saldo=5, quantidade levantamento=10 é concedido um crédito e é possibilitado o levantamento;
- **Caso de teste para a regra 3**: saldo=5, quantidade levantamento=10 não é concedido crédito e não é possibilitado o levantamento;

As tabelas de decisão são um bom método para descrever requisitos quando existem várias regras de negócio que interagem juntas. As tabelas de decisão possibilitam a escrita de

requisitos que cobrem todas as condições. Para os profissionais de testes de software a partir da descrição dos requisitos na tabela de decisão, torna-se mais fácil conseguir obter os casos de teste [22].

3.2.2.3 Limitações da notação

A falta de ordem inerente à representação da informação nas tabelas de decisão é a sua principal limitação. Por esse motivo são melhor aplicadas em sistemas cujos requisitos de negócio requerem muita lógica e decisões que podem ser representadas mais facilmente através de regras e ações na tabela de decisão. Devido à falta de uma sequência que permita definir como acontecem os eventos, através da informação descrita na tabela de decisão, a representação de requisitos de negócio relacionados com aplicações orientadas a eventos é limitada. No entanto, é possível na mesma a sua representação, os eventos de entrada podem ser representados como condições e os eventos de saída como ações [5].

3.2.3 Notação de Máquina de estados finitos (*Finite state machine*)

Uma máquina de estados finitos corresponde à parte lógica de um sistema responsável pelo seu comportamento. A sua notação é utilizada para criar modelos que representem o comportamento de aplicações em todas as situações possíveis. É utilizado o conceito de estado para manter histórico sobre todas as situações, em que a máquina de estados finitos já se encontrou e para preservar o percurso que levou a máquina de estados finitos ao estado em que se encontra atualmente. Devido às inúmeras situações na qual se pode encontrar uma máquina de estados finitos, o seu número de estados possíveis é finito.

Um sistema sequencial necessita de ter conhecimento sobre o comportamento do sistema, que corresponde à sequência de alterações nos valores de entrada, para determinar os valores de saída. O histórico dos valores de entrada é armazenado numa variável interna denominada de estado.

Uma máquina de estados finitos para ser entendida não deverá ser muito complexa, por esse motivo, o número de entradas, saídas e estados que a constituem deverá ter limites. Do ponto de vista documental um diagrama de uma máquina de estados finitos deverá caber numa folha A4 ou A5 [9].

3.2.3.1 Técnicas de utilização da notação

As técnicas de utilização da notação de máquina de estados finitos estão diretamente ligadas à sua forma de representação. A escolha da forma de representação de uma máquina de estados finitos depende sempre do comportamento a ser descrito.

O modelo mais simples para representação de uma máquina de estados finitos é através de um *parser*, que não produz valores de saída apenas reflete as mudanças nos valores de entrada. O seu funcionamento pode ser representado através de o uso de uma matriz de transições ou de um diagrama de transições de estados.

As aplicações que requerem ações para serem executadas também podem ser representadas através da notação de máquinas de estados finitos. Dependendo das condições e do momento que são realizadas, diferentes tipos de ações podem ser definidos.

A matriz de transições tem duas formas de representação. Uma das formas é através de uma tabela onde são definidos valores de entrada que causam transições tal como exemplificado na tabela 8.

Tabela 8 - Exemplo de matriz de transições de estados de um contador da máquina de vendas

Para / De	**Início**	**Cinco**	**Dez**	**Quinze**	**Vinte**	**Parar**
Início	-	5	10	-	-	-
Cinco	-	-	5	10	-	-
Dez	-	-	-	5	10	-
Quinze	-	-	-	-	5	10
Vinte	-	-	-	-	-	5
Parar	-	-	-	-	-	-

A linha (De) representa os estados presentes e a coluna (Para) os estados seguintes. As condições da tabela devem ser cumpridas para que uma transição ocorra de um estado presente para um estado verdadeiro seguinte. Os valores de entrada são considerados uma condição única, no caso de existirem vários valores de entrada que causam a mesma transição podem surgir listados na tabela repetidas vezes. As transições mais complexas podem requerer uma explicação verbal adicional. Algumas abordagens de criação de matriz de transições utilizam o símbolo “- “para representar condições que não se aplicam.

A outra forma de representação da matriz de transições reflete os estados seguintes como uma função do estado atual e dos valores de entrada tal como se visualiza na tabela 9 [9].

Tabela 9 - Forma alternativa da matriz de transições para o contador da máquina de vendas

De / Para	5	10
Início	Cinco	Dez
Cinco	Dez	Quinze
Dez	Quinze	Vinte
Quinze	Vinte	Parar
Vinte	Parar	Vinte
Parar	Parar	Parar

A representação de um diagrama de transições de estados é muito semelhante à representação de uma matriz de transições. O diagrama é constituído por um grafo que utiliza dois elementos, um círculo para indicar o estado e uma seta para indicar uma transição. Em termos matemáticos traduzem-se os círculos em vértices e as linhas em bordas de um grafo orientado. A figura 3 representa um diagrama de transições de estados para um exemplo de uma máquina de estados de um *parser* que guarda os valores de entrada a cada transição de estado. É efetuada uma transição do estado "iniciado" para o estado "terminado" ao receber dois eventos [10].
Um diagrama de transição de estados dependendo da aplicação a representar pode ser mais complicado de desenhar, dado que o desenho do diagrama se traduz num grafo e o da matriz de transições numa tabela. No entanto a sua compreensão é mais fácil através do grafo.

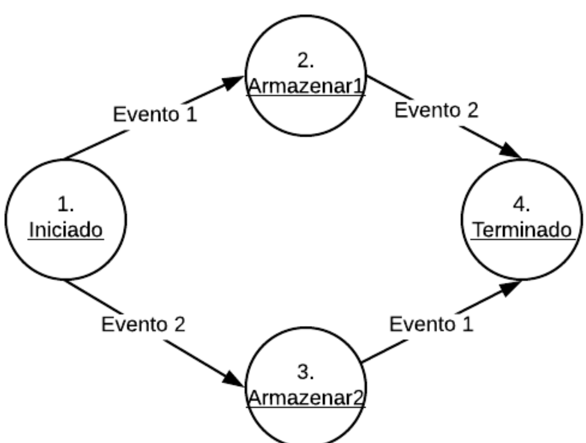


Figura 3 - Diagrama de transições de estados de um *parser*

Em relação a aplicações que requerem ações controladas pelo sistema para serem executadas, a sua representação é feita através de máquinas de estados finitos dependendo das condições e do seu momento de execução.

Existem modelos definidos para identificar os diferentes tipos de comportamentos existentes que podem ocorrer com as máquinas de estados finitos e que definem a sua construção, os modelos *Moore* e *Mealy*. O modelo *Moore* representa máquinas de estados finitos que geram apenas ações de entrada e o modelo *Mealy* que geram apenas ações de saída.

Existem diferentes tipos de momentos que definem a execução de diferentes tipos ações. Se forem ações de entrada e de saída ocorrem quando uma máquina de estados entra num estado e quando saí de outro, respetivamente. Quando uma condição de entrada é verdadeira ocorre uma ação de entrada de valores. Quando um estado é alterado ocorre uma ação de transição [9].

Existem regras que são aplicadas dependendo dos diferentes tipos de ações que são executadas. A utilização de ações ajuda a tornar a máquina de estados finitos mais fácil de entender. Quando um estado de uma máquina de estados finitos é alterado as ações de entrada de valores, saída, transição e entrada são executadas seguindo a sequência mencionada, no mesmo momento. Quando não ocorre nenhuma mudança de estados apenas as ações de valores de entrada são executadas.

Devido à complexidade existente associada à representação de ações utilizando uma matriz de transições ou um diagrama de transições de estados foram introduzidas as tabelas de transições de estados. Cada estado existente poderá conter a sua própria tabela de transições. A tabela de transições de estados, analisando o exemplo da tabela 10 é constituída por diferentes ações de entrada, saída, valores de entrada que geram ações e pelas transições. Cada campo poderá conter várias ações listadas.

Tabela 10 - Tabela de transição de estados

Estado	**Ações_entrada**	**Ação_entrada_1** **Ação_entrada_2**
	Ações_ saída	Ação_saída_1 Ação_saída_2
	Ação_valores_entrada_1	Ação_valores_entrada_1 Ação_valores_entrada_2
	Ação_valores_entrada_2	Ação_valores_entrada_3

Estado_seguinte1	Condições_transição_1	
Estado_seguinte2	Condições_transição_2	

Uma forma de manter a simplicidade da tabela é a mesma não possuir a opção para introduzir ações de transição. Os valores de entrada que geram ações com uma condição igual a uma condição de transição podem ser tratados como uma ação de transição.

As regras de prioridade para as transições e para os valores de entrada que geram ações, caso a tabela de transição seja apenas para expressar comportamento, deveram ser bem definidas. Uma regra de prioridade explícita para transições diz-nos que é impossível realizar mais do que uma transição ao mesmo tempo e que a sequência que surge na tabela a define. As regras de prioridade para valores de entrada que geram ações dependem do ambiente de execução e por esse motivo a sequência em que se encontram na tabela não afeta a prioridade [9].

3.2.3.2 Obter casos de teste a partir de máquina de estados finitos

Uma máquina de estados finitos pode suportar várias métricas de testes nomeadamente todos os estados, eventos de entrada e de saída, transições e todos os caminhos. O caminho percorrido numa máquina de estados finitos é idêntico a um longo caso de uso e a casos de teste num SUT.

Os estados iniciais e os estados finais que podem ser definidos em algumas máquinas de estados finitos e na identificação de casos de teste podem corresponder a pré-condições e pós-condições de um caso de uso. A construção dos casos de teste pressupõe a definição de uma sequência que é possível obter facilmente a partir das causas e das ações das transições da máquina de estados finitos.

Obter casos de teste a partir de uma máquina de estados finitos, que representa um sistema orientado a eventos, funciona de forma diferente. Apesar de não existir uma descrição narrativa nem um nome descritivo para cada ação existem várias fontes de informação para identificação de casos de teste nestes sistemas. Em aplicações que exigem muitos cálculos as máquinas de estados finitos como no exemplo da figura 4 não são tão úteis para obter casos de teste. É possível observar que pouca ou nenhuma informação representada na figura é útil para a definição de casos de teste [9].

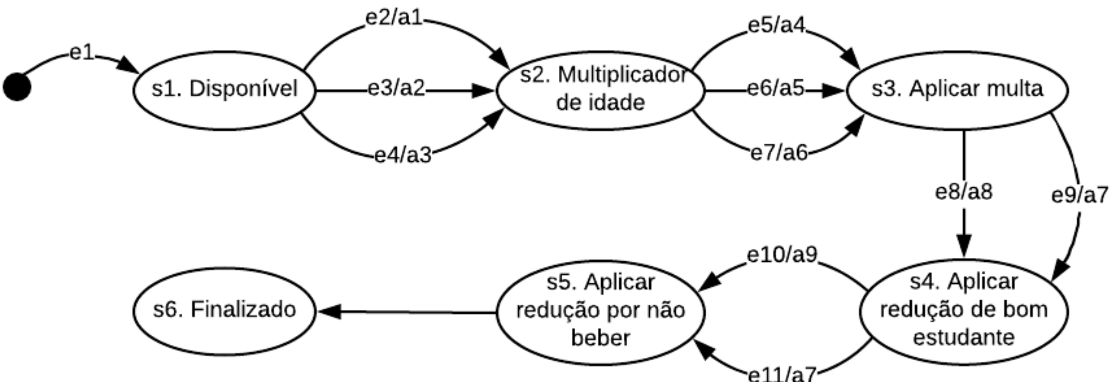


Figura 4 - Máquina de estados finitos de um sistema de cálculos

3.2.3.3 Limitações da notação

Existem duas principais restrições na utilização de máquinas de estados finitos. Uma delas está relacionada com o facto de não usarem memória e a outra com o facto de os estados que fazem parte da máquina de estados finitos serem independentes. Devido a não usarem memória as condições na máquina de estados que dizem que um caminho foi percorrido não podem afetar transições de estados seguintes. Para que isso aconteça os estados que fazem parte da máquina de estados finitos devem também ser independentes. No entanto uma das consequências originadas a partir da independência de estados, denominada por "explosão de estados", acontece quando os estados surgem em demasia nas máquinas de estados finitos como resposta ao número crescente de variáveis de estados no sistema e no espaço do tamanho do sistema. A vulnerabilidade das máquinas de estados finitos para serem alvo da "explosão de estados" é elevada e por esse motivo esta vulnerabilidade é também considerada uma limitação da notação [9].

3.2.4 Notação de Redes de Petri (Petri nets)

As redes de Petri são utilizadas para modelação de diversos tipos de sistemas e a sua estrutura é composta por quatro elementos: P um conjunto de lugares, T um conjunto de transições e por duas funções uma entrada I e uma de saída O, que relacionam transições e lugares.

Uma definição formal de uma rede de Petri pode corresponder a uma quatro-tupla C=(P,T,I,O) onde P={p1, p2,…,pn} é um conjunto de lugares, n>0, T={t1,t2,…,tm} é o conjunto de transições, m>0, I é a função de entrada e O a função de saída ambas responsáveis por efetuar o mapeamento das transições para lugares. Uma rede de Petri é considerada também uma forma especial de máquina de estados finitos.

Através de uma forma gráfica uma rede de Petri traduz-se num grafo orientado biparti-te com dois tipos de nós, um círculo O que representa um lugar e uma barra que representa uma transição. Para efetuar a ligação entre os lugares e as transições são utilizadas setas com a direção pretendida de lugares para transições e vice-versa de um nó de um grafo para outro [11].

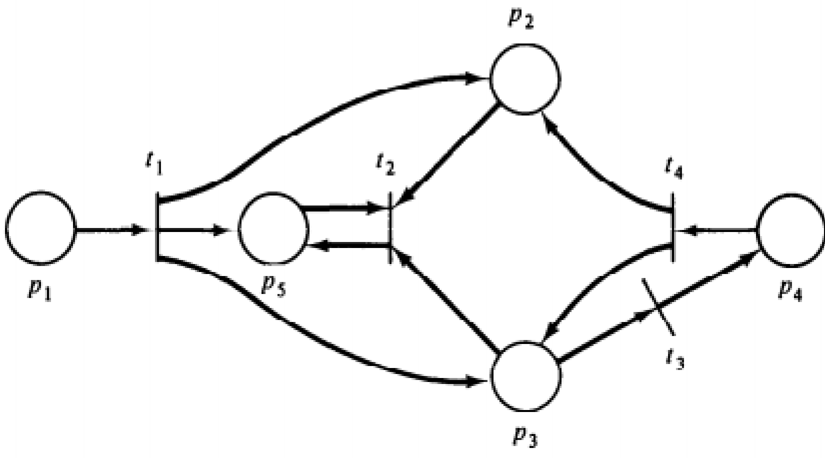


Figura 5 - Exemplo de uma rede de Petri [11]

3.2.2.1 Técnicas de utilização da notação

Um procedimento considerado geral para a construção de uma rede de Petri pode ser dividido em três passos. O primeiro passo deverá passar pela determinação de todos os estados e eventos possíveis do sistema a ser desenhado. O segundo passo deverá passar pelo desenho de um círculo e pela identificação de cada um dos estados. O terceiro passo deverá ser para cada estado identificado a definição de todos os eventos possíveis e para eventos permutáveis desenhar e identificar através de uma seta o evento do estado atual para o evento do estado seguinte até que todos os estados sejam contemplados [12].

Outras regras para modelar uma rede de Petri referem que deverão ser utilizadas transições para representar ações que contém valores de entrada e valores de saída, que os lugares deverão representar dados, pré e pós condições, estados, mensagens e eventos. A relação de entrada deverá ser utilizada para representar pré-requisitos, entradas e a relação de saída consequências e saídas. As marcas numa rede de Petri deverão ser utilizadas para representar estados da rede, memória ou contadores e uma sequência de marcas deverá ser utilizada para gerar casos de teste. As tarefas podem ser consideradas transições individuais ou *subnets* que se forem de lugares de entrada para uma transição podem ser utilizadas para definir o contexto de um evento de entrada [5].

3.2.2.2 Obter casos de teste a partir de redes de Petri

Os caminhos que constituem uma rede de Petri permitem a identificação de casos de teste. A flexibilidade existente em relação à marcação e às setas de entrada e de saída que é possível utilizar nas redes de Petri atribuem-lhe uma forma de memória que não é possível conseguir através da representação de uma máquina de estados finitos. As redes de Petri permitem expressar várias situações que são úteis em contexto de testes tais como conflitos, prioridade, condições mútuas e exclusivas. A sequência de marcas pode representar situações menos óbvias, tais como, situações que possam não permitir mais nenhum progresso que se designam por *deadlock* e *livelock*. Existem redes de Petri que são fáceis de construir e por esse motivo permitem a representação de comportamentos ao nível de classes e dessa forma permitem o suporte de testes de alto nível [5].

3.2.2.3 Limitações da notação

A principal limitação da notação de redes de Petri está relacionada com a sua incapacidade para representar eventos. A forma mais aproximada de representação de eventos numa rede de Petri é através da marcação de um lugar para representar que um evento ocorreu. Para ultrapassar esta limitação da notação foi criada a notação de redes de Petri orientadas a eventos (*event-driven petri nets)* dedicada exclusivamente à representação de eventos através de redes de Petri.

A marcação de lugares de saída e a representação da repetição de eventos de entrada apresenta alguns problemas. A marcação de um lugar, visto que, o lugar permanece marcado torna a sua interpretação enganosa e incorreta, da mesma forma que acontece quando um evento de entrada ocorre mais que uma vez e a sua representação tem de ser repetida na rede de Petri.

Um outro constrangimento durante a utilização da notação de redes de Petri está relacionado com a utilização de lugares comuns numa rede de Petri e com os lugares utilizados para representar eventos, sendo que, estes últimos têm de ser considerados para reconhecerem eventos de entrada sensíveis ao contexto [5].

3.2.3 Notação de Redes de Petri orientadas a eventos (*Event-Driven Petri nets)*

Uma rede de Petri orientada a eventos pode ser descrita como uma tripartite (P,D,T,*In, Out*) um grafo orientado constituído por três conjuntos de nós P,D e T e duas funções de mapeamentos

In e *Out*. O valor P corresponde ao conjunto de eventos de porta o D corresponde ao conjunto de lugares de dados o T corresponde ao conjunto de transições, *In*, *Out* correspondem a funções de mapeamentos que possibilitam a ordenação de conjuntos de pares de transições e de eventos ordenados.

O conjunto T de transições corresponde a transições de redes de Petri mais comuns que são interpretadas como ações. Existem dois tipos de lugares, eventos de porta, lugares de dados que são valores de entrada ou valores de saída de transições em T tal como definido pelas funções de entrada e saída *In* e *Out* [5].

3.2.3.1 Técnicas de utilização da notação

Existem várias regras que devem ser seguidas para uma correta utilização da notação. Durante a representação de transições as mesmas deverão ser utilizadas para representar ações, bem como, os lugares deverão ser utilizados para representar dados, pré, pós-condições, estados e mensagens. Os eventos por sua vez deverão ser utilizados para representar eventos de valores de entrada de porta e eventos de valores de saída de porta. As relações de entrada deverão ser utilizadas para representar pré-requisitos e valores de entrada para as ações. As relações de saída deverão ser utilizadas para representar consequências e valores de saída das ações. A utilização de marcações na rede de Petri orientada a eventos deverão ser utilizadas para representar estados da rede, memória ou contadores.

As tarefas deverão ser consideradas transições individuais e os conjuntos de lugares de valores de entrada para uma transição por norma deverão ser utilizados para definir um contexto. As redes de Petri orientadas a eventos cobrem várias situações que são úteis em contexto de testes, tais como, sequência, seleção, repetição, exclusão mútua e sincronização [5].

3.2.3.2 Obter casos de teste a partir de redes de Petri orientadas a eventos

As informações que se aplicam às redes de Petri no que diz respeito à obtenção de casos de teste também se aplicam para as redes de Petri que são orientadas a eventos. Por serem especificamente orientadas a eventos possuem algumas vantagens que as redes de Petri não orientadas a eventos não contêm. Os eventos são expressamente modelados, os valores de entrada que são sensíveis ao contexto são facilmente identificados e os caminhos que constituem as redes de Petri orientadas a eventos são boas fontes de informação para construção de casos de teste. Quando os caminhos de redes de Petri orientadas a eventos são convertidos em casos de teste, várias métricas de testes podem ser consideradas. Exemplos de

variáveis que podem ser consideradas para métricas de teste são todas as transições, eventos de entrada e eventos de entrada sensíveis ao contexto, todos os eventos de entrada em cada contexto e de saída e todos os eventos de saída com múltipla causa e de saída para cada causa. Todas as pré-condições, pós-condições e todos os caminhos válidos de onde seja possível obter um resultado [5].

3.2.3.3 Limitações da notação

Para além das limitações já mencionadas para as redes de Petri que também são consideradas para as redes de Petri orientadas a eventos, existem mais algumas que surgem devido à utilização de representação de eventos. A forma utilizada para desmarcar um evento de saído contínuo é pouco comum, mas pode ser gerida. Não existe nenhuma diferença gráfica entre a representação de eventos discretos e eventos contínuos. Outra limitação está relacionada com o facto da forma representação não ser escalável [5].

3.2.4 Notação de *Statecharts*

Um *statechart* é um grafo hierárquico orientado (S, T, R, In, Out) onde S é um conjunto de estados que pode conter subestados e regiões ortogonais, T é um conjunto de transições que podem ser definidas visualmente através dos seus pontos iniciais e finais nos limites dos estados, serem causadas por valores de entrada dentro do conjunto de valores de entrada em *In* e causarem valores de saída em *Out*. R é um conjunto de regiões ortogonais e *In* e *Out* são um conjunto de valores de entrada e saída respetivamente [5].

3.2.4.1 Técnicas de utilização da notação

As boas práticas de utilização da notação de *statecharts* referem que a utilização de transições deve ser efetuada para representar mudanças nos estados e subestados. Para a realização de anotações nas transições deverão ser utilizados os símbolos representados da tabela 11 à tabela 13 [5].

Tabela 11 - Tabela com elementos da linguagem para eventos

Evento	Ocorre na seguinte situação:
en(S)	Entra no estado S.
ex(S)	Sai do estado S.
entering(S)	Entra no estado S.
exiting(S)	Sai do estado S.

st(A)	Atividade A é iniciada.
sp(A)	Atividade A é parada.
ch(V)	O valor do item de dados V é alterado.
tr(C)	O valor da condição C é alterada para Verdadeiro (a partir do Falso).
fs(C)	O valor da condição C é alterada para Falso (a partir do Verdadeiro).
rd(V)	O item V de dados é lido.
wr(V)	O item V de dados é escrito.
tm(E, N)	O valor de tempo N que passou desde que o último evento E ocorreu.
E(C)	E ocorreu e a condição C é Verdadeira.
not E	E não ocorreu.
E1 and E2	E1 e E2 ocorreram em simultâneo.
E1 or E2	E1 e E2 ou os dois ocorreram.

Tabela 12 - Tabela com elementos da linguagem para condições

Condição	Verdadeiro quando:
in(S)	O sistema está no estado S.
ac(S)	Atividade A está ativa.
hg(A)	Atividade A está suspensa.
EXP1 R EXP2	O valor da expressão EXP1 e EXP2 satisfazem a relação R. Em expressões numéricas R poderá ser: =, /=, < ou >. Em expressões não apenas numéricas os valores podem ser: = ou /=.
not C	O valor C não é verdadeiro.
C1 e C2	C1 e C2 são verdadeiros.
C1 or C2	C1 e C2 ou os dois são verdadeiros.

Tabela 13 - Tabela com elementos da linguagem para ações

Ação	Efetua:
E	Gera o evento E.
tr!(C)	Assigna o valor verdadeiro à condição C.
fs!(C)	Assigna o valor falso à condição C.
V:=EXP	Assigna o valor de EXP ao valor de dados V.

st!(A)	Ativa a atividade A.
sp!(A)	Termina a atividade A.
sd!(A)	Suspende a atividade A.
rs!(A)	Resume a atividade A.
rd!(V)	Lê o valor de dados V.
wr!(V)	Escreve o valor de dados V.

Deverão ser utilizados estados para representados dados, pré e pós-condições. Utilizados valores de entrada e valores de saída para representar eventos de entrada e de saída de porta, condições de dados e utilizar regiões ortogonais para representar dispositivos separados ou uma situação de processamento concorrente [5].

3.2.4.2 Obter casos de teste a partir de *Statecharts*

A técnica de obtenção de casos de teste a partir de *statecharts* é a mesma que é realizada a partir da notação de redes de Petri orientadas a eventos. No entanto existem algumas limitações. As regiões concorrentes de um *statechart* revelam a existência de uma possível concorrência, mas apenas pode ocorrer um evento de cada vez. A representação de eventos não é tão evidente como é na notação de redes de Petri orientadas a eventos e não existe uma forma fácil de reconhecer eventos de entrada sensíveis ao contexto [5].

3.2.4.3 Limitações da notação

A principal desvantagem da utilização da notação é a complexidade existente nos elementos que dela fazem parte, sobretudo na linguagem utilizada nas transições. Esta complexidade dificulta a realização de uma inspeção técnica num *statechart* complexo [5].

3.2.5 Notação UML

A linguagem UML (*Unified Modeling Language*) é uma linguagem de modelação utilizada para especificar, construir, visualizar e manter histórico das partes constituintes de um sistema. Através da modelação de um sistema é possível recolher decisões e informações sobre como deverá ser construído. Comunicar a estrutura e comportamento pretendido e ideal para o sistema. Possibilitar a visualização, ter um controlo da arquitetura do sistema e expor as suas oportunidades de simplificação e reutilização.

A linguagem UML permite aplicar três diferentes técnicas de construção de modelos. Uma das técnicas é o *forward engineering* que permite a geração de código com base no modelo, outra das técnicas é o *reverse engineering* geração do modelo com base no código. A outra técnica é o *round-trip engineering* que se traduz num ciclo iterativo de desenvolvimento com geração de código a partir de um modelo e a atualização do modelo com base no código cujo objetivo da linguagem UML neste processo é o de suportar os processos de desenvolvimento orientados a objetos já existentes.

Os modelos construídos facilitam o entendimento do funcionamento, construção do sistema e a comunicação incluindo o que pode ser representado numa linguagem de programação. Os símbolos utilizados em UML possuem uma semântica bem definida, permitem especificar decisões importantes ao nível da análise, desenho e implementação.

Através da utilização da notação UML é possível criar histórico utilizando documentação relacionada com as partes constituintes de um sistema. A documentação pode conter detalhes relacionados com os conceitos de um problema ou cenários de instalação e até mesmo adicionar *links* para informação externa, tais como, documentos de requisitos e planos de testes [13].

As técnicas de utilização da notação são colocadas em prática através da construção de modelos que possuem vários propósitos, nomeadamente [14]:

- Recolher e definir requisitos de negócio de forma precisa e conhecimento de domínio para permitir que todos os envolvidos possam entender e concordar sobre os mesmos;
- Incentivar o pensamento sobre a forma de construção de um sistema antes de o mesmo ser colocado em prática;
- Recolher decisões de construção do sistema de forma mutável separada dos requisitos;
- Gerar vários tipos de produtos de trabalho, desde produtos do género de demonstrações à geração de classes, procedimentos técnicos, interfaces entre outros;
- Organizar, filtrar, permitir encontrar e examinar informação em sistemas de larga escala;
- Explorar múltiplas soluções economicamente através da análise e comparação entre vários tipos de modelos;
- Gerir a dificuldade existente nos modelos através da abstração que é possível obter na modelação para um nível que permita ser compreendido por humanos;

3.2.5.1 Técnicas de utilização da notação

Os tipos de diagramas UML existentes podem ser comportamentais e considerar aspetos dinâmicos do sistema ou serem estruturais e considerar apenas aspetos estáticos do sistema.
Dentro dos modelos comportamentais fazem parte os diagramas de casos de uso, sequência, colaboração, estado e atividades. Dentro dos modelos estruturais os diagramas que fazem parte são os diagramas de classes, objetos, componentes e instalação [13]. Alguns dos diagramas foram analisados com maior detalhe no que respeita à sua técnica de utilização.

- **Diagrama de casos de uso**

O objetivo principal de um diagrama de casos de uso é o de facilitar o trabalho de equipas de desenvolvimento durante a análise de requisitos funcionais de um sistema. A representação de um diagrama de casos de uso inclui a relação entre atores, que personificam humanos que interagem com o sistema e com os processos, tal como, os relacionamentos entre os diferentes casos de uso.
Os casos de uso podem surgir agrupados no diagrama para um sistema completo ou para um grupo em particular relacionado apenas com uma funcionalidade do sistema. Para desenhar um caso de uso deve desenhar-se um círculo no centro do diagrama e atribuir-lhe um nome também no centro ou abaixo do caso de uso.
Para desenhar um ator poderá ser utilizado um símbolo que representa uma pessoa do lado direito ou esquerdo do diagrama. As linhas devem ser utilizadas para representar o relacionamento entre os atores e os casos de uso. Um exemplo de um diagrama de casos de uso que contempla todas as técnicas e pontos mencionados pode visualizar-se pela figura 6 [16].

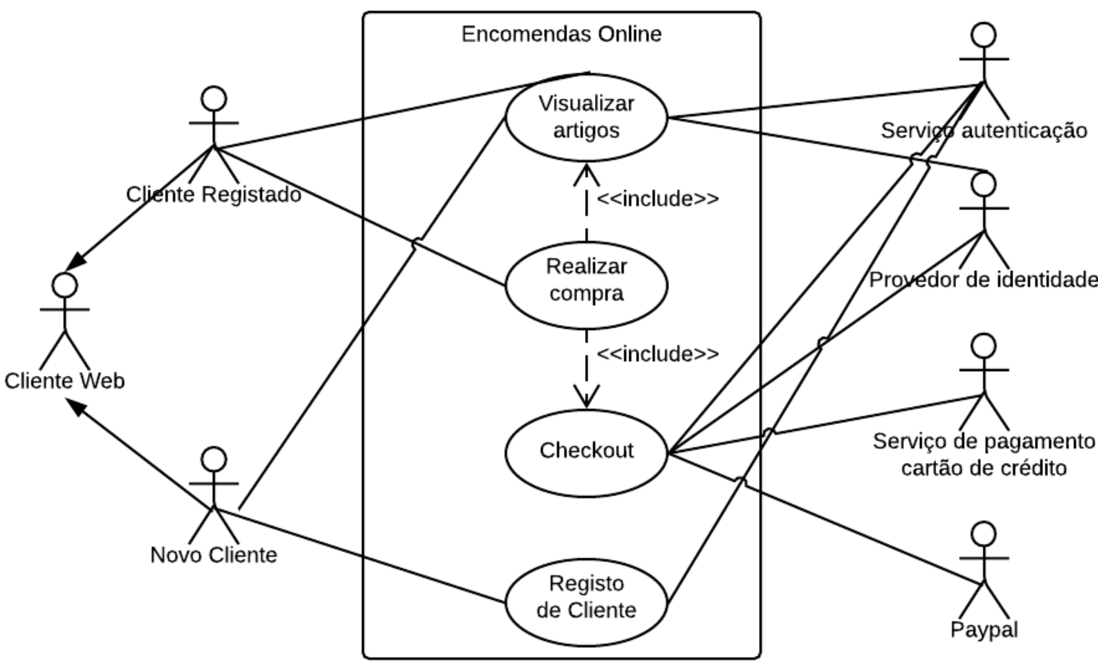


Figura 6 - Exemplo diagrama casos de uso [15]

- **Diagrama de sequência**

Os diagramas de sequência são utilizados para detalhar tal como a sua designação indica uma sequência de acontecimentos para um caso de uso específico ou até mesmo apenas para uma parte. Permitem demonstrar pedidos que existem entre os diferentes objetos e a sua sequência. Podem também em alguns casos a um nível detalhado demonstrar diferentes pedidos para diferentes objetos.

Os diagramas de sequência possuem duas dimensões, uma dimensão vertical que mostra a sequência de pedidos ordenados que ocorrem ao longo do tempo e uma dimensão horizontal que mostra as instâncias dos objetos na qual as mensagens são enviadas.

Para desenhar um diagrama de sequência é necessário identificar as instâncias das classes e colocá-las dentro de uma caixa com o seu nome separado por um espaço e dois pontos. Quando uma classe enviar uma mensagem para outra instância de outra classe deverá ser desenhada uma seta a apontar para a instância da classe que a vai receber e colocado o nome da mensagem por cima da linha da seta. O valor retornado deverá também ser representado por uma linha tracejada [16].

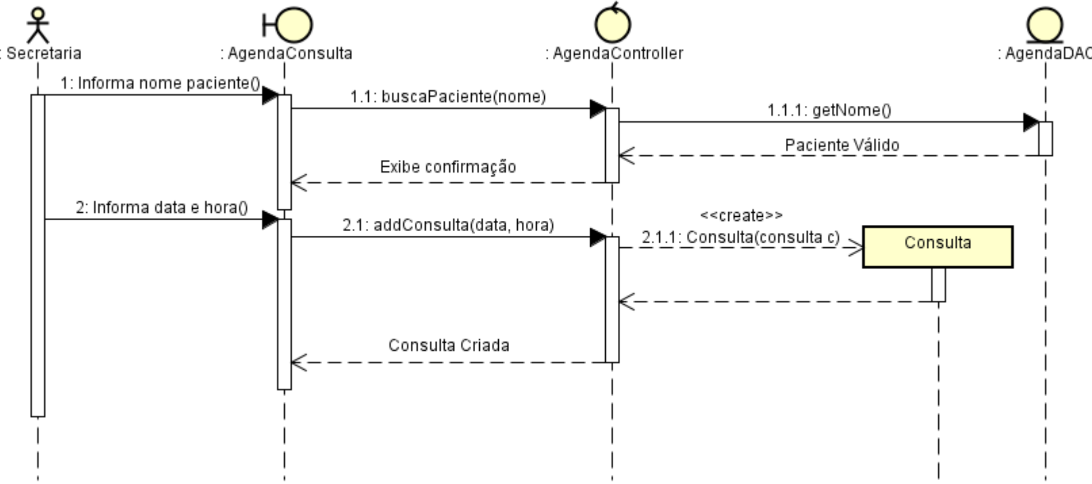


Figura 7 - Exemplo de diagrama de sequência [17]

- **Diagrama de estado**

O diagrama de estado é utilizado para modelar os diferentes estados que uma classe pode conter e como as transições dessa classe ocorrem de um estado para o outro. Apenas as classes com três ou mais potenciais estados durante a atividade de um sistema devem ser modeladas. A notação de um diagrama de estado é constituída por cinco elementos básicos um ponto inicial representado por um círculo sólido. Transições entre estados representadas por setas. Estados representados por retângulos com os rebordos arredondados. Pontos de decisão representados através de círculos abertos e pontos de fim representados através de círculos com um círculo sólido no seu interior.

O desenho do diagrama de estado deverá começar por um ponto inicial e uma seta de transição que representa o estado inicial da classe. Os estados deverão ser desenhados em qualquer posição no diagrama e depois conectados utilizando as linhas de transição [16].

- **Diagrama de atividades**

Os diagramas de atividades são utilizados para demonstrar o funcionamento entre dois ou mais objetos de uma classe durante o decorrer de uma atividade. Sendo considerados diagramas menos técnicos o conteúdo expressado por eles facilita a compreensão dos sistemas modelados.

A notação utilizada para construir um diagrama de atividades é constituída por um círculo sólido que se liga à atividade inicial. A atividade é modelada desenhando um retângulo com rebordos arredondados juntamente com o nome da atividade. As atividades são ligadas a outras atividades através de setas de transição ou a pontos de decisão que se ligam a diferentes

atividades guardadas por condições do ponto de decisão. As atividades que terminam o processo modelado são ligadas ao ponto de fim [16].

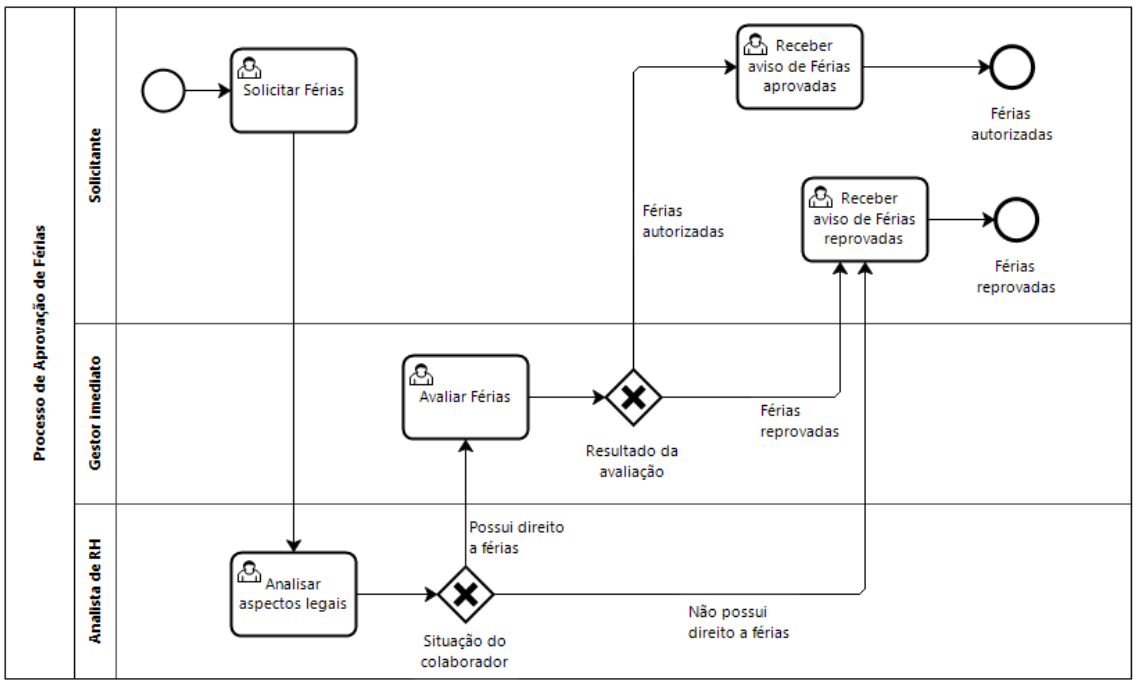


Figura 8 - Exemplo de diagrama de atividades [18]

3.2.5.2 Obter casos de teste a partir da notação UML

A obtenção de casos de teste utilizando a notação UML é também realizada individualmente de acordo com cada diagrama utilizado. Foram analisadas, para os diagramas de casos de uso, sequência e atividades o seu valor na obtenção de casos de teste.

- **Obter casos de teste a partir de diagramas de casos de uso**

O diagrama de casos de uso não oferece suporte para a geração de casos de teste. No máximo, pode ajudar a definir uma ordem na qual os testes podem ser agrupados, presumivelmente através dos atores definidos [5].

- **Obter casos de teste a partir de diagramas de sequência**

Os diagramas de sequência são redundantes relativamente à informação que representam. A sua estrutura pode ser interpretada como um caso de teste de integração devido à clara identificação das fontes e destinos de comunicações internas [5].

Observando o diagrama da figura 7, um exemplo de um caso de teste que pode ser obtido a partir da informação descrita no diagrama poderá ser:

Tabela 14 – Casos de teste obtidos a partir da figura 7

Caso de teste	**Passos do teste**
Agendamento de uma consulta médica	1. Secretária pesquisa nome de paciente; Obtém paciente válido; Informa data e hora da consulta; Agenda a consulta;

- **Obter casos de teste a partir de diagramas de atividades**

As técnicas já mencionadas de obtenção de casos de teste que se aplicam aos fluxogramas também se aplicam aos diagramas de atividades. A sequência existente nos diagramas de atividades possibilita na maioria dos casos uma obtenção direta de casos de teste abstratos [5]. Observando a figura 8, por exemplo, poderão ser obtidos dois casos de teste:

Tabela 15 – Casos de teste obtidos a partir da figura 8

Casos de teste	Passos do teste
1. **Aprovar férias do colaborador**	**1.** Receber solicitação de férias do colaborador; **2.** Analisar aspetos legais do colaborador; **2.** Analisar viabilidade de atribuição de férias; **3.** Aprovar férias do colaborador;
2. **Reprovar férias do colaborador**	**1.** Receber solicitação de férias do colaborador; **2.** Analisar aspetos legais do colaborador; **3.** Analisar viabilidade de atribuição de férias; **4.** Reprovar férias do colaborador;

3.2.5.3 Limitações da notação

O facto da notação UML conter uma grande variedade de diagramas torna-a complexa de aprender e de utilizar.

É necessário muito tempo para gerir e manter os diagramas UML dado que para serem corretamente utilizados e interpretados tem de estar completamente alinhados com o funcionamento do software. Por este motivo o tempo para gerir e manter os diagramas UML adiciona mais tempo de trabalho a um projeto.

Não é claro no âmbito do desenvolvimento de software quem são os principais intervenientes que beneficiam com a construção dos diagramas UML. De acordo com um artigo publicado no site Eiffel software a linguagem UML não apresenta vantagem para os profissionais do desenvolvimento de software dado que estes trabalham com código e não com imagens ou diagramas. Mesmo para gestores de projeto ou executivos a demonstração sobre como funciona um software poderá tornar-se mais fácil através do uso de um quadro ou da escrita num papel em vez de ser investido tempo a aprender a linguagem UML [19].

3.2.6 Notação BPMN (Business Process Modeling and Notation)

A notação BPMN *"Business Process Modeling and Notation"* é uma notação cujo propósito é o de modelar os passos que devem ser seguidos de um processo de negócio, de forma a melhorar a eficiência ou obter vantagem competitiva, desde o seu início até ao seu fim. Durante este processo de modelação são descritos visualmente atividades e fluxos de informação necessários à execução do processo. Através da modelação dos processos é possível obter um entendimento fácil de cada etapa. É utilizada uma linguagem padrão e comum para que todas as partes envolvidas tenham o mesmo entendimento sobre os processos fornecendo detalhes suficientes para permitir uma implementação completa. A notação BPMN utiliza quatro tipos de elementos para a construção dos diagramas de processo de negócios, nomeadamente [20]:

- Objetos de fluxo (*flow objects*): eventos, atividades e decisores de fluxos (*gateways*);
- Objetos de conexão (*connecting objects*): fluxo sequencial, fluxo de mensagens, associação;
- Divisórias (*swim lanes*): *"pool"* ou *"lane"*;
- Artefactos (*artifacts*): objetos de dados, grupos, anotações;

Na tabela 16, é possível observar cada símbolo que pertence à notação e a descrição sobre o seu papel na construção dos processos de negócio.

Tabela 16 - Símbolos da notação BPMN

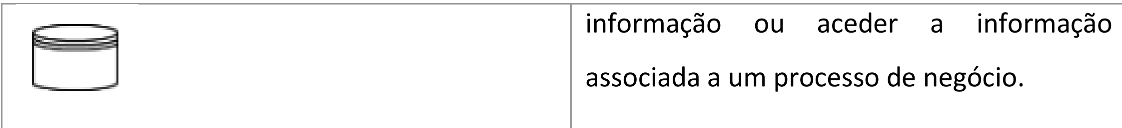
BPMN tipos de eventos	Descrição
○	Símbolo de começo de evento, assinala o primeiro passo do processo.

	Símbolo intermédio de evento, que representa qualquer evento que ocorra entre o evento de começo e o evento de fim.
	Símbolo de fim de evento, assinala o último passo do processo.
BPMN símbolos de eventos	
	Símbolo de mensagem que aciona o processo, facilita os processos intermediários ou termina o processo.
	Símbolo de temporizador que corresponde a uma hora, data, tempo ou datas recorrentes que desencadeiam o processo, auxiliam processos intermediários ou concluem o processo.
	Símbolo de escalada é utilizado num subprocesso de eventos e ocorre quando alguém num nível mais alto de responsabilidade dentro da organização é envolvido num processo.
	Símbolo condicional representa um processo que começa ou continua quando uma condição de negócio ou uma regra de uma condição é conhecida.
	Símbolo de link que representa um subprocesso que pertence a um processo maior.
	Símbolo de erro que representa um erro encontrado no início, meio ou fim de um processo.
	Símbolo de cancelamento pode representar uma transação que foi cancelada dentro de

	um subprocesso. Num evento de fim o símbolo de cancelamento representa que o cancelamento foi desencadeado.
	Símbolo de compensação representa que uma devolução foi desencadeada.
	Símbolo de sinal comunica entre diferentes processos. O símbolo de sinal pode começar um processo, facilitá-lo ou completá-lo.
	Símbolo de multiplicação representa que múltiplos desencadeamentos iniciaram um processo.
	Símbolo de múltiplo paralelismo representa uma instância de um processo não começa, ou continua ou termina sem que todos os eventos possíveis tenham ocorrido.
	Símbolo de término que desencadeia o término imediato de um step de um processo. E todas as instâncias relacionadas são terminadas ao mesmo tempo.
BPMN símbolos de atividades	
	Símbolo que representa uma tarefa.
	Símbolo que representa um subprocesso.
	Símbolo que representa uma transação.
	Símbolo que representa uma chamada.

BPMN símbolos de decisores de fluxos	
	Símbolo de exclusividade que avalia o estado do processo de negócio baseado na condição que quebra o fluxo em um ou mais caminhos mutuamente exclusivos.
	Símbolo baseado em eventos que avalia qual foi o evento que ocorreu.
	Símbolo de paralelismo que representa duas tarefas concorrentes no fluxo de negócio.
	Símbolo de inclusão que parte o fluxo de processo em um ou mais fluxos.
	Símbolo de evento baseado em exclusividade que começa uma nova instância de um processo com a ocorrência de um evento subsequente.
	Símbolo de complexidade que é mais utilizado em fluxos complexos num processo de negócio.
	Símbolo de paralelismo baseado em eventos que permite que múltiplos processos ocorram ao mesmo tempo.
BPMN ligação entre objetos	
	Símbolo de fluxo de sequência que efetua a ligação entra objetos de fluxo numa determinada sequência.
	Símbolo de mensagem que representa mensagens de um processo de um participante para outro.

....	Símbolo de associação que permite mostrar uma relação entre artefactos e fluxos de objetos.
BPMN Swimlanes	
Pool Lane Lane	As *swimlanes* são utilizadas para organizar os aspetos envolventes de um processo num diagrama BPMN. Os elementos podem ser organizados de forma horizontal ou vertical. Para além de organizarem também revelam atrasos, ineficiências e os trabalhadores responsáveis de cada etapa de um processo.
BPMN artfactos	
Group Annotation	Anotações: permitem modelar para descrever partes adicionais de um fluxo de um modelo ou notação. Grupos: organizam tarefas ou processos que tem significância em todo o processo. Objetos de dados: representam informação colocada dentro de um processo, resultante do processo, informação que precisa de ser recolhida ou guardada.
	Símbolo de entrada de informação que representa requisitos de informação que as tarefas no processo de negócio necessitam.
	Símbolo de saída de informação que representa informação produzida como resultado de um processo de negócio.
	Símbolo de recolha de informação que significa informação recolhida dentro de um processo de negócio.
	Símbolo de armazenamento de informação que representa a habilidade de guardar

	informação ou aceder a informação associada a um processo de negócio.

3.2.6.1 Técnicas de utilização

Os símbolos que constituem a notação são bastante claros e apenas pela sua observação é possível entender qual será o seu funcionamento e o seu objetivo. Os diagramas ajudam a descrever processos de negócio de forma sólida através de uma forma escrita ou apenas através do seu entendimento. Um diagrama BPMN que seja bem construído deve conter apenas um evento de início e apenas um evento de fim, no entanto podem existir atividades internas dentro de *swiwlanes*. As atividades cuja representação seja maior podem ser decompostas em diagramas BPMN de forma separada à semelhança do que acontece com os processos dos fluxogramas [5].

3.2.6.2 Obter casos de teste a partir da notação BPMN

A geração de casos de teste a partir de processos de negócio permite obter uma cobertura de casos de uso de negócio bastante completa, levando à deteção antecipada de possíveis erros no *software* e nos serviços utilizados [6].

A identificação de casos de teste utilizando processos de negócio é dividida em cinco fases [6]:

1. Descrição de negócio e definição de requisitos;
2. Arquitetura e descrição de componentes;
3. Implementação;
4. Testes de componentes, integração e de sistema;
5. Testes de aceitação;

Para acontecerem as fases 1, 2 e 5 é necessário o conhecimento do responsável pelo processo, bem como, dos responsáveis de execução do processo do departamento de operações em questão. É necessário também o conselheiro do processo da organização de negócio interna. Nas fases 2, 3 e 4 é necessário um analista das tecnologias da informação, que pode representar a pessoa de contacto que tem o conhecimento de suporte aos processos de negócio e que gere a execução dos casos de teste.

A primeira etapa passa pela estruturação de casos de teste utilizando um mapa de processos e/ou requisitos. A segunda etapa passa pela utilização de diagramas de processos de negócio já

existentes e pela análise de cada caminho nos diagramas para identificação dos casos de teste [6].

A geração de casos de teste com a notação BPMN é muito semelhante à geração de casos de teste a partir de fluxogramas. Os caminhos representados, à semelhança dos caminhos representados nos fluxogramas, permitem a identificação direta de casos de teste abstratos [5].

3.2.6.3 Limitações da notação

O facto de a notação BPMN possuir símbolos com uma expressão exata obriga à necessidade de domínio total do seu vocabulário para uma utilização eficaz, fazendo com que seja necessário muito tempo para aprendizagem da notação. A principal limitação da notação é a sua vasta coletânea de símbolos e por consequência o tempo exigido para a sua correta compreensão e utilização [5].

CAPÍTULO 4

FERRAMENTAS MODEL BASED TESTING

Dada a relevância, durante a implementação da metodologia MBT, da utilização de ferramentas que a suportam, o capítulo 4 é dedicado à investigação das ferramentas existentes. Neste capítulo é também selecionada uma ferramenta para utilização no caso de estudo prático, com base nos resultados obtidos a partir de uma comparação entre as ferramentas de uso livre.

4.1 Ferramentas existentes

Na tabela 17, encontram-se listadas todas as ferramentas atualmente existentes no mercado que suportam a metodologia MBT.

Tabela 17 - Listagem de ferramentas MBT [27]

Ferramenta	Data última atualização	Formato de escrita	Tipo
4Test	2017	Customizado (Baseado em *Gherkin*)	Comercial + Uso livre
BPM-Xchange	2017	BPMN, UML	Comercial
Conformiq Creator	2017	Diagrama de atividades, DSL	Comercial
Conformiq Designer	2016	Máquina de estados UML, QML	Comercial
DTM	2016	Modelo de atividades customizado	Comercial
fMBT	2017	Customizado (AAL)	Uso livre
GraphWalker	2017	FSM	Uso livre
GSL	2017	Petri net	Comercial
JSXM	2016	EFSM (*Stream X-machines*)	Uso académico
JTorX	2014	LTS	Uso livre
MaTeLo	2017	*Markov chains*	Comercial

MBTsuite	2016	UML ou BPMN	Comercial
MISTA	2015	PrT net	Uso académico
Modbat	2016	EFSM (DSL baseado em *Scala*)	Uso livre
ModelJUnit	2016	EFSM	Uso livre
MoMuT:: UML	2017	Diagramas de estados UML, OOAS	Uso académico
OSMO	2016	Programa modelo em java	Uso livre
RT-Tester	2017	UML/SysML, *Matlab*	Comercial
Smartesting CertifyIt	2017	UML/BPMN + OCL	Comercial
Smartesting Yest	2017	*Workflow* de modelos utilizando tabelas de decisão	Comercial
Tcases	2016	Customizado	Uso livre
TEMPPO	2014	Modelo de fluxo de tarefas	Comercial
TestCast	2017	Máquina de estados UML	Comercial
TestOptimal	2017	(E)FSM	Comercial
Tricentis Tosca	2017	Modelo de dados	Comercial
T-VEC	2013	*Simulink*	Comercial

4.2 Comparação entre ferramentas

Da listagem, as ferramentas de uso livre, foram selecionadas para uma comparação mais detalhada. Esta comparação foi realizada tendo em conta vários critérios importantes a serem considerados, durante a utilização de qualquer ferramenta, a eficiência, usabilidade, multiplataforma, escalabilidade, integração contínua e o suporte. O objetivo desta comparação foi selecionar a melhor ferramenta MBT a ser utilizada no caso prático. Posteriormente à realização desta comparação detalhada, foi utilizado o método AHP para auxiliar na seleção da ferramenta. A ferramenta selecionada para o caso prático foi o *GraphWalker*.

Tabela 18 - Comparação de características entre ferramentas

	Eficiência	Usabilidade	Multiplataforma	Escalabilidade	Integração Contínua	Suporte
fMBT	1. O desenvolvimento dos casos de teste é realizado de forma muito rápida [5].	1. A ferramenta permite a realização de testes em classes individuais C++, interfaces gráficas, dispositivos móveis, sistemas distribuídos e em diferentes plataformas [5]. 2. Disponibiliza um editor de modelos, um gerador de sequências de testes e vários adaptadores para realização dos testes [5]. 3. É uma ferramenta complicada de utilizar, exige conhecimento da linguagem de programação *Python* [5]. 4. Fornece poucas funcionalidades. E a forma de a utilizar é considerada pobre em comparação com outras ferramentas *Model-Based Testing* comerciais. Existindo ainda, consideradas melhores, outras ferramentas gratuitas [5].	1. Pode ser utilizada em Linux [5].	1. Não foi encontrada referência.	1. Não foi encontrada referência.	1. A última atualização da versão da ferramenta corresponde a dia 21 de outubro de 2017 [36]. 2. Contém uma página de suporte, uma lista de emails. Um repositório no GitHub cuja última atualização é referente ao ano de 2017. [36] 3. Possui algumas descrições e alguns exemplos sobre como utilizar a ferramenta [36].
Graph Walker	1. É considerada uma ferramenta que permite a automação de testes de forma rápida [5]. 2. A ferramenta de modelação para a criação dos modelos, *yEd*, é considerada de utilização fácil e rápida [5].	1. Permite a geração de testes em modo *online* e *offline* o a partir de modelos. Utiliza como notação de modelação a das máquinas de estados finitos [5]. 2. Permite a geração dos testes a partir do modelo com base em critérios de geração de sequências de teste internos à ferramenta. E que atuam como condição de geração, mas também de paragem de execução [5].	1. Funciona em qualquer plataforma que suporte Java [5].	1. A utilização de uma componente gráfica para representação dos modelos, é considerada uma limitação na escalabilidade da ferramenta [5].	1. Permite integração com ferramentas de entrega contínua, como por exemplo, o *Jenkins*.	1. Existe alguma documentação e alguns projetos exemplo. 2. Existe um grupo do google atualmente ativo, cujo tempo de resposta, é quase imediato.
JSXM	1. Permite a geração de casos de teste unitários através da transformação de casos de teste abstratos em casos de teste unitários	1. Utiliza as linguagens Java e XML cuja utilização é bastante comum; 2. Requer bastantes conhecimentos técnicos e experiência	1. Pode ser utilizada em Windows, OSX/Unix; [28]	1. Existe referência relacionada com a intenção de acrescentar novas funcionalidades	1. Não foi encontrada referência sobre a possibilidade, no entanto, sendo testes	1. Apenas existe uma conta no GitHub para reportar problemas, mas já não é

	utilizando o *JUnit*; [28] 2. Não existem muitos casos de uso, nem artigos sobre a sua utilização. De acordo com a referência [28] a ferramenta foi utilizada em várias aplicações cujo foco é a utilização de serviços web e em todas elas foi capaz de encontrar erros. [29] 3. A geração de casos de teste é garantida que funcione no caso de uma especificação uniforme dos parâmetros de entrada. No entanto através de demonstração, por exemplo, também funciona com uma especificação não uniforme dos parâmetros de entrada [29].	prévia com linguagens de programação e com a utilização de linha de comandos; 3. Não utiliza nenhuma linguagem de modelação muito utilizada ou conhecida. Utiliza a linguagem (*Stream X-Machine*) das máquinas de estados finitos estendidas [29].		que valida ser possível alterar o que já existe [29].	unitários desenvolvidos através de um projeto *Maven*, deverá ser possível integrar com ferramentas de integração contínua;	possível aceder; 2. Última atualização de informações relativas à ferramenta é referente ao ano de 2014; 3. Contém vários manuais de utilização bastante completos [28].
MISTA	1. Permite a realização e verificação do modelo criado para assegurar que todos os estados e transições podem ser encontrados. Realiza esta validação tendo em conta um estado inicial e um estado final e dessa forma assegura que o modelo utilizado se encontra bem construído [5]. 2. A ferramenta de realização de modelos pode por vezes tornar-se pesada e causar alguns problemas durante a sua utilização [5]. 3. Permite a simulação de redes de Petri para assegurar que o modelo funciona conforme esperado [5]. 4. A ferramenta proporciona um conjunto vasto de opções e critérios de	1. Suporta Java, C, C++, C#, *Python*, HTML e VB. E suporta as *frameworks* de testes *xUnit*, *Selenium* IDE e *Robot framework* [31]. 2. O editor gráfico para desenvolvimento do modelo é baseado numa plataforma denominada PIPE3 (*Platform Independent Petri Net Editor*). A forma de construção dos modelos é simples consiste num editor semelhante a uma folha de cálculo para criar textualmente os modelos e com opções para inserir código auxiliar para geração de código de testes [5]. 3. A utilização da ferramenta, para além do conhecimento sobre o seu funcionamento, exige também conhecimento sobre as redes de Petri e/ou sobre as máquinas de estados finitos [31]. 4. A geração de casos de teste apenas exige a escolha de um critério	1. Pode ser utilizada em Mac, Windows e Linux [5].	1. Não foi encontrada referência.	1. Não foi encontrada referência.	1. Existem dois documentos com instruções sobre a utilização da ferramenta que estão divididos de acordo com o tipo de rede de Petri utilizada [31]. 2. O manual de utilizador contém apenas 56 páginas que constitui um bom indicador da simplicidade da ferramenta [8]. 3. Existem vários projetos que são considerados como sendo bastante completos que podem ser assistidos

	cobertura de testes [5]. 5. As redes de petri são consideradas uma ferramenta para modelação de sistemas poderosa, que possibilitam que a ferramenta MISTA tenha capacidade de modelar uma grande variedade de sistemas [5].	de cobertura de testes e o pressionar de um botão. A forma de geração dos casos de teste faz com que a ferramenta seja adequada para utilização em conjunto com uma outra metodologia de desenvolvimento de testes o *test-driven development* [5]. 5. Comparada com outras ferramentas comerciais é considerada uma ferramenta de utilização simples [5]. 6. Como utiliza redes de petri e máquinas de estado finitos para a realização dos modelos, em vez de uma linguagem de modelação customizada, é considerada uma ferramenta mais acessível [5].				e que explicam o âmbito das redes de petri em relação à ferramenta [31].
ModBat	1. Permite expressar facilmente uma vasta variedade de ações do sistema, incluindo operações não determinísticas [34]. 2. Foi possível encontrar defeitos que antes não tinha sido possível em sistemas complexos [34]. 3. A capacidade de expressão das ferramentas é um fator importante para eficiência da técnica do *Model-Based Testing* [34].	1. Utiliza a linguagem de programação *Scala* cuja utilização é considerada complexa [32]. 2. Utiliza a notação de máquinas de estados finitos estendidas [32]. 3. É uma ferramenta dedicada aos testes de *APIs* [32]. 4. O modelo contém a estrutura de máquinas de estados finitos não determinísticas [32].	1. A aplicação é constituída por dois ficheiros .jar e, portanto, qualquer sistema com suporte na execução de programas em Java permite o seu funcionamento [33].	1. O software não pode ser modificado, submetido a engenharia reversa ou divulgado a terceiros [32].	1. Não foi encontrada referência.	1. Existe um ficheiro README com informações sobre como proceder para realizar a instalação, com algumas indicações sobre como utilizar a ferramenta [34]. 2. Existe um email de contacto pouco percetível [34]. 3. Última atualização referente ao ano de 2003 [32].
ModelJunit	1. Durante a utilização da ferramenta foi reportado que a mesma continha alguns bugs e deixava completamente de funcionar de tempos a tempos [5]. 2. O facto de existir pouca documentação	1. O *ModelJUnit* é uma libraria Java que estende o *JUnit* para suportar a metodologia do *Model-Based Testing* [5]. 2. Os modelos são criados, utilizando a notação de máquinas de estados finitos e a linguagem de programação Java. E	1. Funciona em qualquer plataforma que suporte Java.	1. Não foi encontrada referência.	1. Não foi encontrada referência sobre a possibilidade, no entanto, sendo testes realizados utilizando *JUnit* deverá ser possível integrar com ferramentas	1. Existem três projetos localizados em páginas distintas com diferentes versões disponíveis [5]. 2. Parece não existir praticamente nenhuma

	sobre a ferramenta de uma forma geral também foi considerado como um impedimento na eficiência da ferramenta tornando a sua utilização complicada. [5].	de seguida convertidos numa forma de representação gráfica [5]. 3. É possível utilizar diferentes métricas de cobertura de testes para a geração dos casos de teste [5].			de integração contínua.	informação disponível e/ou oficial sobre a ferramenta [5]. 3. A ferramenta parece ter sido abandonada a última atualização é referente ao ano de 2014 [5].
MoMuT:UML	1. A ferramenta foi aplicada com sucesso em diferentes casos de uso industriais na área dos sistemas embebidos [35]. 2. A geração de casos de teste é eficiente a encontrar falhas durante a sua execução [37]. 3. Foram encontrados erros através da utilização da ferramenta que não teriam sido encontrados realizando apenas testes manuais [38].	1. Permite gerar casos de teste a partir de modelos UML, aplicando operadores de mutação no modelo para obter vários modelos "defeituosos". Utiliza o modelo original e um modelo "defeituoso" por cada vez e procura por sequências de entradas e saídas para revelar uma diferença de comportamento entre o modelo original e o modelo defeituoso [37].	1. Existem pacotes de instalação da ferramenta para Windows e Linux [35].	1. Não foi encontrada referência.	1. Não foi encontrada referência.	1. Existe um email de contacto [35]. 2. Última modificação da ferramenta realizada em 2018 [31].
OSMO	1. Os modelos visuais bem desenhados perdem expressividade [40]. 2. Existiram vários problemas com modelos visuais incompreensíveis [40]. 3. A abordagem proporcionada pela ferramenta é considerada muito efetiva, no design de testes. Especialmente para profissionais dos testes de software com *skills* em Java [40]. 4. Permite a execução online e *offline*, mas a sua utilização é restrita apenas para testar uma aplicação [40]. 5. Os eventos externos ao sistema a ser testado, são suportados por elementos do	1. Utiliza modelos que são programas escritos e anotados em Java. Cria testes explorando esses modelos através da utilização de diferentes estratégias [39]. 2. É considerada uma ferramenta simples [40]. 2. Para se aproveitar toda a potencialidade da ferramenta, é necessário utilizar, para criação dos modelos, a mesma linguagem de programação utilizada no desenvolvimento das aplicações. Dessa forma, poderá permitir total integração da ferramenta com os sistemas [40].	1. Funciona em qualquer plataforma que suporte Java.	1. Não foi encontrada referência.	1. Não foi encontrada referência.	1. Contém uma página no GitHub com bastante informação, um manual de utilização, documentos e exemplos [39].

	ambiente de testes [40].					
TCases	1. A precisão dos casos de teste gerados é totalmente dependente do que é descrito nos ficheiros XML. As variáveis, condições ou impedimentos relacionados tem de se encontrar bem detalhados e de forma precisa [43].	1. É uma ferramenta de testes combinatória para testar funções do sistema e gerar dados de entrada. [42] 2. Para utilização da ferramenta é necessário existir um documento XML que define o SUT [42]. 3. A geração dos testes acontece por nível de cobertura de dados permitindo o controlo do número de casos de teste gerados [42].	1. Funciona nas plataformas Windows e Linux [44].	1. Não foi encontrada referência.	1. Não foi encontrada referência sobre a possibilidade, no entanto, sendo testes que podem ser convertidos em testes *JUnit* deverá ser possível integrar com ferramentas de entrega contínua [42].	1. Contém uma página no GitHub com bastante informação, um manual de utilização, documentos e exemplos [41]. 2. Última atualização da ferramenta foi em setembro de 2018 [44].
JTorX	1. Foi comprovado que através da utilização do *JTorX* foram encontrados defeitos num SUT que não teriam sido encontrados sem a utilização da ferramenta [45].	1. O design do *JTorX* é baseado na teoria refinada de *ioco* na qual, os casos de teste recebem dados de entrada. [45] 2. A interface necessita de melhorias [45]. 3. O programa ocasionalmente durante a sua utilização, perdeu dados de entrada [45].	1. Funciona nas plataformas Windows, Linux e Mac [46].	1. Não foi encontrada referência.	1. Não foi encontrada referência.	1.Existe a página da ferramenta que contém bastante detalhe. No entanto, contém poucos exemplos de utilização da ferramenta. [46]

4.3 Seleção de ferramenta a utilizar no caso prático

Para auxiliar na escolha da melhor ferramenta MBT para utilização no caso de estudo apresentado na componente prática foi utilizado o método *Analytic Hierarchy Process* (AHP).

O primeiro passo, seguindo o método AHP foi construir uma árvore de decisão para refletir os objetivos, critérios e as alternativas disponíveis para a tomada de decisão, distribuídos por níveis. Foi então desenhada a árvore de decisão com o objetivo, selecionar a melhor ferramenta MBT para o caso de estudo. Com os critérios de comparação entre ferramentas, eficiência, usabilidade, multiplataforma, escalabilidade, integração contínua e suporte. As ferramentas a ser alvo de comparação são: *JSXM, JTORX, MISTA, Modbat, ModelJunit, MoMuT:UM, fMBT, OSMO, TCase* e o *GraphWalker*.

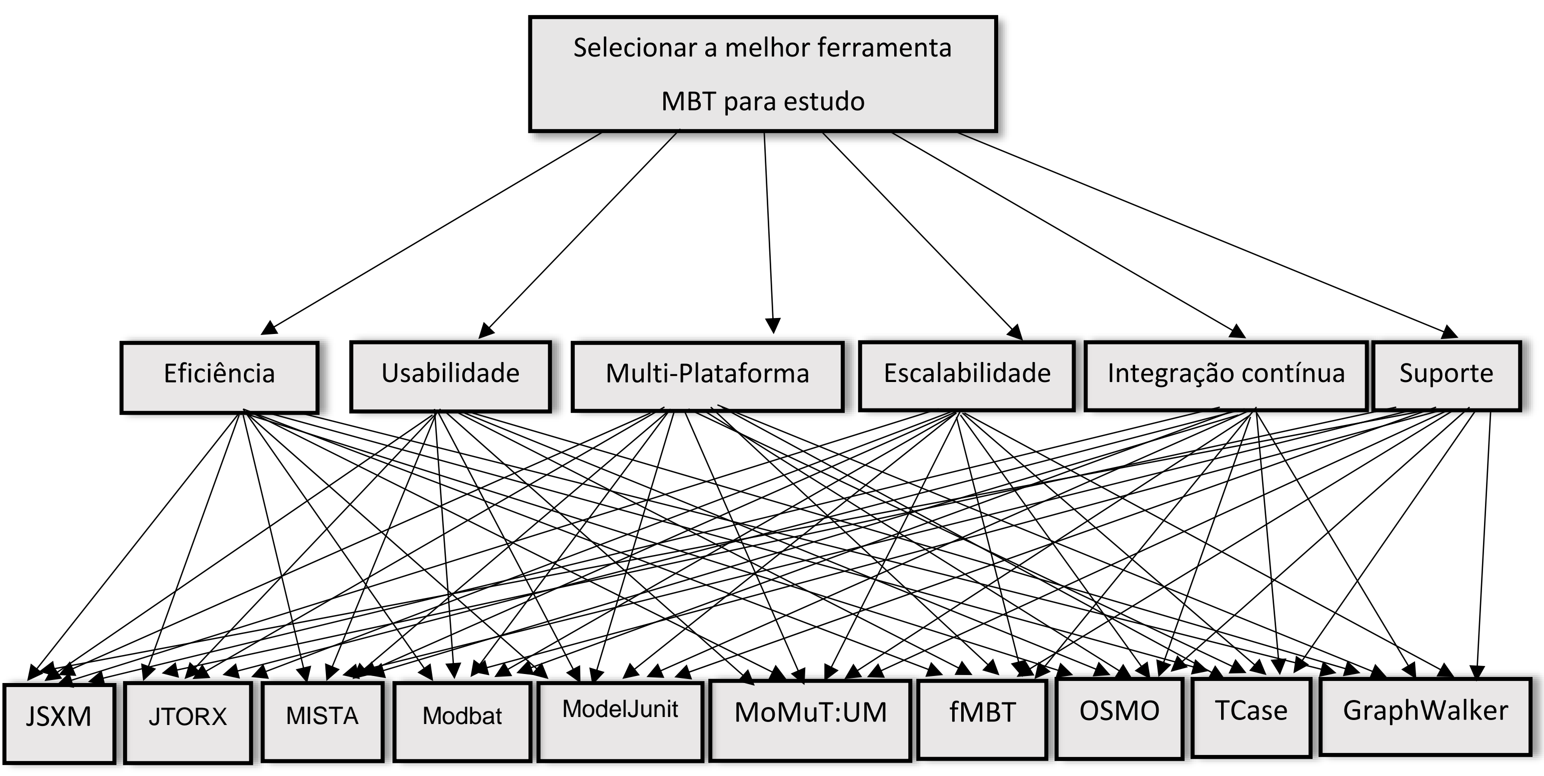


Figura 9 - Árvore de decisão a utilizar seguindo método AHP

O segundo passo foi efetuar uma comparação dos critérios com as alternativas. Para a realização desta comparação foi utilizada uma escala que se designa, escala de *Saaty* ou escala fundamental do método AHP. Esta escala indica quantas vezes mais um elemento é importante sobre o outro elemento em relação ao critério da qual é comparado.

Tabela 19 - Escala de *Saaty* ou escala fundamental do método AHP

Escala de importância	Definição	Explicação
1	Importância igual.	As duas atividades contribuem da mesma forma para o objetivo.
3	Moderada importância de uma sobre a outra (um pouco mais importante).	A experiência e o juízo favorecem uma atividade em relação à outra.
5	Importância grande ou essencial (muito mais importante).	A experiência e o juízo favorecem fortemente uma atividade em relação à outra.
7	Importância muito grande.	Uma atividade é muito fortemente favorecida em relação à outra. Pode ser demonstrado na prática.

9	Importância extrema ou absoluta.	A evidência favorece uma atividade em relação à outra, como o mais alto grau de segurança.
2,4,6,8	Valores intermédios.	Quando se procura uma condição de compromisso entre duas definições.

A partir da comparação é gerada uma matriz de comparação para os critérios disponíveis em relação às alternativas e em relação ao objetivo global.

Tabela 20 - Matriz de comparação de critérios

	Eficiência	**Usabilidade**	**Multiplataforma**	**Escalabilidade**	**Integração contínua**	**Suporte**
Eficiência	9/9	9/7	9/7	9/7	9/7	9/7
Usabilidade	7/9	7/7	7/5	9/7	7/7	7/7
Multiplataforma	7/9	7/7	7/5	5/7	7/7	5/7
Escalabilidade	7/9	7/9	7/5	7/7	7/7	9/7
Integração contínua	7/9	7/7	7/7	7/7	7/7	8/7
Suporte	7/9	7/7	7/5	7/9	7/8	7/7

Analisando a tabela 20 o critério da eficiência é bem mais importante do que todos os restantes critérios apresentados na tabela. O critério da usabilidade é menos importante do que a eficiência, do que a escalabilidade, é mais importante do que o critério multiplataforma, sendo de igual importância em relação aos restantes. O critério multiplataforma é menos importante do que a eficiência, do que a escalabilidade e do que o suporte, sendo de igual importância, em relação aos restantes critérios. O critério escalabilidade é mais importante do que a usabilidade, do que o critério multiplataforma e do que o critério suporte, sendo de igual importância em relação aos restantes. O critério integração contínua é menos importante do que a eficiência, é mais importante do que o suporte sendo de igual importância em relação aos restantes. O critério suporte é menos importante do que a eficiência, do que a escalabilidade e do que a integração contínua. É mais importante do que o critério multiplataforma e é de igual importância em relação aos restantes. A matriz de comparação é posteriormente normalizada dando origem à priorização de critérios.

Tabela 21 - Matriz de comparação de critérios com soma por colunas

	Eficiência	Usabilidade	Multiplataforma	Escalabilidade	Integração contínua	Suporte
Eficiência	1.0	1.2	1.2	1.2	1.2	1.2
Usabilidade	0.7	1.0	1.4	1.2	1.0	1.0
Multiplataforma	0.7	1.0	1.4	1.4	1.0	0.7
Escalabilidade	0.7	0.7	1.4	1.0	1.0	1.2
Integração contínua	0.7	1.0	1.0	1.0	1.0	1.1
Suporte	0.7	1.0	1.4	0.7	0.8	1.0
Soma por colunas	4.5	6.1	7.8	6.0	6.0	6.2

Tabela 22 - Matriz de comparação de critérios normalizada com média

	Eficiência	Usabilidade	Multiplataforma	Escalabilidade	Integração contínua	Suporte	Média=Prioridade
Eficiência	0.2	0.19	0.15	0.2	0.2	0.19	0.18
Usabilidade	0.16	0.15	0.17	0.2	0.16	0.16	0.16
Multiplataforma	0.16	0.15	0.17	0.23	0.16	0.11	0.16
Escalabilidade	0.16	0.11	0.17	0.16	0.16	0.19	0.15
Integração contínua	0.16	0.15	0.14	0.16	0.16	0.17	0.15
Suporte	0.16	0.15	0.17	0.11	0.13	0.16	0.14

A partir dos resultados obtidos, as prioridades definidas relativamente aos critérios são o critério eficiência que surge em primeiro lugar seguido pela usabilidade e pelo critério multiplataforma. De seguida o critério escalabilidade, o critério integração contínua e por fim o critério suporte. De seguida é realizada uma comparação entre todas as ferramentas em relação a cada critério.

Tabela 23 - Matriz de comparação de alternativas com critério eficiência

	fMBT	GraphWalker	JSXM	MISTA	ModBat	Model Junit	MoMuT:UML	OSMO	TCases	JTorX
fMBT	3/3	3/9	3/4	3/7	3/5	3/1	3/7	3/3	3/1	1/3
GraphWalker	3/9	9/9	9/4	9/7	9/5	9/1	9/7	9/3	9/1	9/3
JSXM	4/3	4/9	4/4	4/7	1/5	4/1	4/7	4/1	4/2	4/4
MISTA	7/3	7/9	7/4	7/7	7/5	7/1	7/4	7/3	7/4	7/3
ModBat	5/3	5/9	5/1	5/7	5/5	5/1	5/5	5/3	5/1	5/3
ModelJunit	1/3	1/9	1/4	1/7	1/5	1/1	1/3	1/4	1/2	¼
MoMuT:UML	7/3	7/9	7/4	4/7	5/5	3/1	4/4	4/3	4/2	5/3

OSMO	3/3	3/9	4/1	3/7	3/5	4/1	3/4	3/3	3/2	3/1
TCases	1/3	1/9	2/4	4/7	5/1	2/1	2/4	2/3	3/3	3/5
JTorX	3/1	3/9	4/4	3/7	3/5	4/1	3/5	3/1	5/3	3/3

Tabela 24 - Matriz de comparação de alternativas com critério eficiência com soma por colunas

	fMBT	GraphWalker	JSXM	MISTA	ModBat	Model Junit	MoMuT:UML	OSMO	TCases	JTorX
fMBT	3.0	0.3	0.7	0.4	0.6	3.0	0.4	1.0	3.0	0.3
GraphWalker	0.3	1.0	2.2	1.2	1.8	9.0	1.2	3.0	9.0	3.0
JSXM	1.3	0.4	1.0	0.5	0.2	4.0	0.5	4.0	2.0	1.0
MISTA	2.3	0.7	1.7	1.0	0.4	7.0	1.7	2.3	1.7	2.3
ModBat	1.6	0.5	5.0	0.7	1.0	5.0	1.0	1.6	5.0	1.6
ModelJunit	0.3	0.1	0.2	0.1	0.2	1.0	0.3	0.2	0.5	0.2
MoMuT:UML	0.4	0.7	0.2	0.5	1.0	3.0	1.0	1.3	2.0	1.6
OSMO	1.0	0.3	4.0	0.4	0.6	4.0	0.7	1.0	1.5	3.0
TCases	0.3	0.1	0.5	0.5	5.0	2.0	0.5	0.6	1.0	0.6
JTorX	3.0	0.3	1.0	0.4	0.6	4.0	0.6	3.0	1.6	1.0
Soma por colunas	13.5	4.4	16.5	5.7	11.4	42	7.9	18	27.3	14.6

Tabela 25 - Matriz de comparação de alternativas com critério eficiência normalizada com média

	fMBT	GraphWalker	JSXM	MISTA	ModBat	ModelJunit	MoMuT:UML	OSMO	TCases	JTorX	Média= Prioridade
fMBT	0.22	0.06	0.04	0.07	0.05	0.07	0.05	0.05	0.10	0.02	0.085
GraphWalker	0.02	0.22	0.13	0.2	0.15	0.21	0.15	0.16	0.10	0.2	0.15
JSXM	0.09	0.09	0.06	0.08	0.17	0.09	0.06	0.22	0.07	0.06	0.099
MISTA	0.17	0.15	0.10	0.17	0.03	0.16	0.2	0.12	0.06	0.15	0.13
ModBat	0.11	0.11	0.30	0.12	0.08	0.11	0.12	0.08	0.18	0.10	0.13
ModelJunit	0.02	0.02	0.01	0.01	0.01	0.02	0.03	0.01	0.01	0.01	0.016
MoMuT:UML	0.02	0.02	0.01	0.08	0.08	0.07	0.12	0.07	0.07	0.10	0.064
OSMO	0.07	0.06	0.24	0.07	0.05	0.09	0.08	0.05	0.05	0.2	0.096
TCases	0.02	0.02	0.03	0.08	0.43	0.04	0.06	0.06	0.03	0.04	0.081
JTorX	0.2	0.06	0.06	0.07	0.05	0.09	0.4	0.16	0.05	0.06	0.12

Para o critério eficiência a prioridade em relação às ferramentas é do *GraphWalker* com pontuação de 0.15 superior a todas as outras ferramentas.

Tabela 26 - Matriz de comparação de alternativas com critério usabilidade

	fMBT	GraphWalker	JSXM	MISTA	ModBat	ModelJunit	MoMuT:UML	OSMO	TCases	JTorX
fMBT	1/1	9/1	1/1	1/7	1/1	1/3	1/1	1/3	1/2	1/1

GraphWalker	9/1	9/9	9/1	9/7	9/1	9/2	9/1	9/2	9/2	9/1
JSXM	1/1	2/9	2/2	2/5	2/1	1/2	1/1	2/3	1/2	1/3
MISTA	7/1	7/9	5/2	7/7	1/1	7/7	7/3	3/7	3/5	3/3
ModBat	1/1	1/9	1/2	1/3	1/1	1/3	1/3	1/5	1/2	1/1
ModelJunit	3/1	2/9	2/1	7/7	3/1	3/3	3/3	3/5	3/2	3/1
MoMuT:UML	1/1	9/1	1/1	3/7	3/1	3/3	3/3	3/3	3/3	3/1
OSMO	3/1	2/9	3/2	7/3	5/1	5/3	3/3	3/3	3/1	3/1
TCases	2/1	2/9	2/1	5/3	2/1	2/3	3/3	1/3	3/3	1/1
JTorX	1/1	1/9	3/1	3/3	1/1	1/3	1/3	1/3	1/1	1/1

Tabela 27 - Matriz de comparação de alternativas com critério usabilidade com soma por colunas

	fMBT	**GraphWalker**	**JSXM**	**MISTA**	**ModBat**	**ModelJunit**	**MoMuT:UML**	**OSMO**	**TCases**	**JTorX**
fMBT	1.0	9.0	1.0	0.1	1.0	0.3	1.0	0.3	0.5	1.0
GraphWalker	9.0	1.0	9.0	1.2	9.0	4.2	9.0	4.2	4.5	9.0
JSXM	1.0	0.2	1.0	0.4	2.0	0.5	1.0	0.6	0.5	0.3
MISTA	7.0	0.7	2.5	1.0	1.0	1.0	2.3	0.4	0.6	1.0
ModBat	1.0	0.1	0.5	0.3	1.0	0.3	0.3	0.2	0.5	1.0
ModelJunit	3.0	0.2	2.0	1.0	3.0	1.0	1.0	0.6	1.2	3.0
MoMuT:UML	1.0	9.0	1.0	0.4	3.0	1.0	1.0	1.0	1.0	3.0
OSMO	3.0	0.2	1.5	2.3	5.0	1.6	1.0	1.0	3.0	3.0
TCases	1.0	0.2	2.0	1.6	2.0	0.6	1.0	0.3	1.0	1.0
JTorX	2.0	0.1	3.0	1.0	1.0	0.3	0.3	0.3	1.0	1.0
Soma por colunas	29	20.7	23.5	9.3	28	10.8	17.9	8.9	13.8	23.3

Tabela 28 - Matriz de comparação de alternativas com critério usabilidade normalizada com média

	fMBT	**GraphWalker**	**JSXM**	**MISTA**	**ModBat**	**ModelJunit**	**MoMuT:UML**	**OSMO**	**TCases**	**JTorX**	**Média=Prioridade**
fMBT	0.03	0.43	0.04	0.01	0.03	0.02	0.05	0.03	0.03	0.04	0.071
GraphWalker	0.3	0.04	0.38	0.12	0.32	0.38	0.50	0.47	0.32	0.38	0.321
JSXM	0.03	0.009	0.04	0.04	0.07	0.04	0.05	0.06	0.03	0.01	0.037
MISTA	0.24	0.003	0.10	0.10	0.03	0.09	0.12	0.04	0.04	0.04	0.080
ModBat	0.03	0.004	0.02	0.03	0.03	0.02	0.016	0.02	0.03	0.04	0.024
ModelJunit	0.10	0.009	0.08	0.10	0.10	0.09	0.05	0.06	0.08	0.12	0.078
MoMuT:UML	0.03	0.43	0.04	0.04	0.10	0.09	0.05	0.11	0.11	0.12	0.112
OSMO	0.10	0.009	0.06	0.24	0.17	0.14	0.05	0.11	0.21	0.12	0.120

TCases	0.03	0.009	0.08	0.17	0.07	0.05	0.05	0.03	0.07	0.04	0.059
JTorX	0.06	0.004	0.12	0.10	0.03	0.02	0.01	0.03	0.07	0.04	0.048

Para o critério usabilidade a prioridade em relação às ferramentas é do *GraphWalker* com pontuação de 0.321 superior a todas as outras ferramentas.

Tabela 29 - Matriz de comparação de alternativas com critério Multiplataforma

	fMBT	GraphWalker	JSXM	MISTA	ModBat	ModelJunit	MoMuT:UML	OSMO	TCases	JTorX
fMBT	1/1	1/5	1/1	1/3	1/5	1/5	1/1	1/5	1/1	1/3
GraphWalker	5/1	5/5	5/1	5/3	5/5	5/5	5/1	5/5	5/1	5/1
JSXM	1/1	1/5	1/1	1/3	1/5	1/5	1/1	1/5	1/1	1/3
MISTA	5/1	5/5	5/1	5/3	5/5	5/5	5/1	5/5	5/1	5/1
ModBat	5/1	5/5	5/1	5/3	5/5	5/5	5/1	5/5	5/1	5/1
ModelJunit	5/1	5/5	5/1	5/3	5/5	5/5	5/1	5/5	5/1	5/1
MoMuT:UML	3/1	3/5	3/5	1/5	1/5	1/5	3/3	1/5	1/5	1/5
OSMO	5/1	5/5	5/1	5/5	5/5	5/5	5/1	5/5	5/5	5/5
TCases	1/1	1/5	1/1	1/5	1/5	1/5	5/1	5/5	3/3	1/5
JTorX	3/1	1/5	3/1	1/5	1/5	1/5	5/1	5/5	5/1	5/5

Tabela 30 - Matriz de comparação de alternativas com critério multiplataforma com soma por colunas

	fMBT	GraphWalker	JSXM	MISTA	ModBat	ModelJunit	MoMuT:UML	OSMO	TCases	JTorX
fMBT	1.0	0.2	1.0	0.3	0.5	0.5	1.0	0.5	1.0	0.33
GraphWalker	5.0	1.0	5.0	1.6	1.0	1.0	5.0	1.0	5.0	5.0
JSXM	1.0	0.2	1.0	0.3	0.5	0.5	1.0	0.5	1.0	0.33
MISTA	5.0	1.0	5.0	1.6	1.0	1.0	5.0	1.0	5.0	5.0
ModBat	5.0	1.0	5.0	1.6	1.0	1.0	5.0	1.0	5.0	5.0
ModelJunit	5.0	1.0	5.0	1.6	1.0	1.0	5.0	1.0	5.0	5.0
MoMuT:UML	3.0	0.6	0.6	0.2	0.5	0.5	1.0	0.5	0.5	0.2
OSMO	5.0	1.0	5.0	1.0	1.0	1.0	5.0	1.0	1.0	1.0
TCases	1.0	0.2	1.0	0.2	0.5	0.5	5.0	1.0	1.0	0.2
JTorX	3.0	0.2	3.0	0.2	0.5	0.5	5.0	1.0	5.0	1.0
Soma por colunas	34	6.4	31.6	8.6	7.5	7.5	38	8.5	29.5	23.06

Tabela 31 - Matriz de comparação de alternativas com critério multiplataforma normalizada com média

	fMBT	GraphWalker	JSXM	MISTA	ModBat	ModelJunit	MoMuT:UML	OSMO	TCases	JTorX	Média= Prioridade
fMBT	0.02	0.03	0.03	0.03	0.06	0.06	0.02	0.05	0.03	0.014	0.034
GraphWalker	0.14	0.15	0.15	0.18	0.13	0.13	0.13	0.11	0.16	0.21	0.149

JSXM	0.02	0.03	0.03	0.03	0.06	0.06	0.02	0.05	0.03	0.014	0.034
MISTA	0.14	0.15	0.15	0.18	0.13	0.13	0.13	0.11	0.16	0.21	0.149
ModBat	0.14	0.15	0.15	0.18	0.13	0.13	0.13	0.11	0.16	0.21	0.149
ModelJunit	0.14	0.15	0.15	0.18	0.13	0.13	0.13	0.11	0.16	0.21	0.149
MoMuT:UML	0.08	0.09	0.01	0.02	0.06	0.06	0.02	0.05	0.01	0.008	0.040
OSMO	0.14	0.15	0.15	0.11	0.13	0.13	0.13	0.11	0.03	0.04	0.112
TCases	0.02	0.03	0.03	0.02	0.06	0.06	0.13	0.11	0.03	0.008	0.049
JTorX	0.08	0.03	0.09	0.02	0.06	0.06	0.13	0.11	0.58	0.04	0.12

Para o critério multiplataforma a prioridade em relação às ferramentas é do *GraphWalker*, *MISTA*, *ModBat* e do *ModelJunit* todas com pontuação de 0.149, superior a todas as outras ferramentas.

Tabela 32 - Matriz de comparação de alternativas com critério Escalabilidade

	fMBT	**GraphWalker**	**JSXM**	**MISTA**	**ModBat**	**ModelJunit**	**MoMuT:UML**	**OSMO**	**TCases**	**JTorX**
fMBT	1/1	1/5	1/3	1/1	1/3	1/1	1/1	1/1	1/1	1/1
GraphWalker	5/1	5/5	5/3	5/1	5/3	5/1	5/1	5/1	5/1	5/1
JSXM	3/1	3/5	3/3	3/1	3/3	3/1	3/1	3/1	3/1	3/1
MISTA	1/1	1/5	1/3	1/1	1/3	1/1	1/1	1/1	1/1	1/1
ModBat	3/1	3/5	3/3	3/1	3/3	3/1	3/1	3/1	3/1	3/1
ModelJunit	1/1	1/5	1/3	1/1	1/3	1/1	1/1	1/1	1/1	1/1
MoMuT:UML	1/1	1/5	1/3	1/1	1/3	1/1	1/1	1/1	1/1	1/1
OSMO	1/1	1/5	1/3	1/1	1/3	1/1	1/1	1/1	1/1	1/1
TCases	1/1	1/5	1/3	1/1	1/3	1/1	1/1	1/1	1/1	1/1
JTorX	1/1	1/5	1/3	1/1	1/3	1/1	1/1	1/1	1/1	1/1

Tabela 33 - Matriz de comparação de alternativas com critério Escalabilidade com soma por colunas

	fMBT	**GraphWalker**	**JSXM**	**MISTA**	**ModBat**	**ModelJunit**	**MoMuT:UML**	**OSMO**	**TCases**	**JTorX**
fMBT	1.0	0.2	0.3	1.0	0.3	1.0	1.0	1.0	1.0	1.0
GraphWalker	5.0	1.0	1.6	5.0	1.6	5.0	5.0	5.0	5.0	5.0
JSXM	3.0	0.6	1.0	3.0	1.0	3.0	3.0	3.0	3.0	3.0
MISTA	1.0	0.2	0.3	1.0	0.3	1.0	1.0	1.0	1.0	1.0
ModBat	3.0	0.6	1.0	3.0	1.0	3.0	3.0	3.0	3.0	3.0
ModelJunit	1.0	0.2	0.3	1.0	0.3	1.0	1.0	1.0	1.0	1.0

MoMuT:UML	1.0	0.2	0.3	1.0	0.3	1.0	1.0	1.0	1.0	1.0
OSMO	1.0	0.2	0.3	1.0	0.3	1.0	1.0	1.0	1.0	1.0
TCases	1.0	0.2	0.3	1.0	0.3	1.0	1.0	1.0	1.0	1.0
JTorX	1.0	0.2	0.3	1.0	0.3	1.0	1.0	1.0	1.0	1.0
Soma por colunas	18	3.6	5.7	18	5.7	18	18	18	18	18

Tabela 34 - Matriz de comparação de alternativas com critério Escalabilidade normalizada com média

	fMBT	**GraphWalker**	**JSXM**	**MISTA**	**ModBat**	**Model Junit**	**MoMuT:UML**	**OSMO**	**TCases**	**JTorX**	**Média= Prioridade**
fMBT	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
GraphWalker	0.27	0.27	0.28	0.27	0.28	0.27	0.27	0.27	0.27	0.27	0.27
JSXM	0.16	0.16	0.17	0.16	0.17	0.16	0.16	0.16	0.16	0.16	0.162
MISTA	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
ModBat	0.16	0.16	0.17	0.16	0.17	0.16	0.16	0.16	0.16	0.16	0.162
ModelJunit	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
MoMuT:UML	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
OSMO	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
TCases	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
JTorX	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05

Para o critério escalabilidade a prioridade em relação às ferramentas é do *GraphWalker* com pontuação igual 0.27 superior a todas as outras ferramentas.

Tabela 35 - Matriz de comparação de alternativas com critério Integração contínua

	fMBT	**GraphWalker**	**JSXM**	**MISTA**	**ModBat**	**ModelJunit**	**MoMuT:UML**	**OSMO**	**TCases**	**JTorX**
fMBT	1/1	1/9	1/5	1/1	1/1	1/5	1/1	1/1	1/5	1/1
GraphWalker	9/1	9/9	9/5	9/1	9/1	9/5	9/1	9/1	9/5	9/1
JSXM	5/1	9/5	5/5	5/1	5/1	5/5	5/1	5/1	5/5	5/1
MISTA	1/1	1/9	1/5	1/1	1/1	1/5	1/1	1/1	1/5	1/1
ModBat	1/1	1/9	1/5	1/1	1/1	1/5	1/1	1/1	1/5	1/1
ModelJunit	5/1	5/9	5/5	5/1	5/1	5/5	5/1	5/1	5/5	5/1
MoMuT:UML	1/1	1/9	1/5	1/1	1/1	1/5	1/1	1/1	1/5	1/1
OSMO	1/1	1/9	1/5	1/1	1/1	1/5	1/1	1/1	1/5	1/1
TCases	5/1	5/9	5/5	5/1	5/1	5/5	5/5	5/1	5/5	5/1
JTorX	1/1	1/9	1/5	1/1	1/1	1/5	1/1	1/1	1/5	1/1

Tabela 36 - Matriz de comparação de alternativas com critério Escalabilidade com soma por colunas

	fMBT	GraphWalker	JSXM	MISTA	ModBat	ModelJunit	MoMuT:UML	OSMO	TCases	JTorX
fMBT	1.0	0.11	0.2	1.0	1.0	0.2	1.0	1.0	0.2	1.0
GraphWalker	9.0	1.0	1.8	9.0	9.0	1.8	9.0	9.0	1.8	9.0
JSXM	5.0	1.8	1.0	5.0	5.0	1.0	5.0	5.0	1.0	5.0
MISTA	1.0	0.11	0.2	1.0	1.0	0.2	1.0	1.0	0.2	1.0
ModBat	1.0	0.11	0.2	1.0	1.0	0.2	1.0	1.0	0.2	1.0
ModelJunit	5.0	0.5	1.0	5.0	5.0	1.0	5.0	5.0	1.0	5.0
MoMuT:UML	1.0	0.11	0.2	1.0	1.0	0.2	1.0	1.0	0.2	1.0
OSMO	1.0	0.11	0.2	1.0	1.0	0.2	1.0	1.0	0.2	1.0
TCases	5.0	0.5	1.0	5.0	5.0	1.0	1.0	5.0	1.0	5.0
JTorX	1.0	0.11	0.2	1.0	1.0	0.2	1.0	1.0	0.2	1.0
Soma por colunas	30	4.46	6	30	30	6	26	30	6	30

Tabela 37 - Matriz de comparação de alternativas com critério integração contínua normalizada com média

	fMBT	GraphWalker	JSXM	MISTA	ModBat	ModelJunit	MoMuT:UML	OSMO	TCases	JTorX	Média= Prioridade
fMBT	0.03	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.029
GraphWalker	0.3	0.22	0.3	0.3	0.3	0.3	0.34	0.3	0.3	0.3	0.296
JSXM	0.16	0.40	0.16	0.16	0.16	0.16	0.19	0.16	0.16	0.16	0.187
MISTA	0.03	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.029
ModBat	0.03	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.029
ModelJunit	0.16	0.11	0.16	0.16	0.16	0.16	0.19	0.16	0.16	0.16	0.158
MoMuT:UML	0.03	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.029
OSMO	0.03	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.029
TCases	0.16	0.11	0.16	0.16	0.16	0.16	0.03	0.16	0.16	0.16	0.142
JTorX	0.03	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.029

Para o critério integração contínua a prioridade em relação às ferramentas é do *GraphWalker* com pontuação igual 0.296 superior a todas as outras ferramentas.

Tabela 38 - Matriz de comparação de alternativas com critério Suporte

	fMBT	GraphWalker	JSXM	MISTA	ModBat	ModelJunit	MoMuT:UML	OSMO	TCases	JTorX
fMBT	3/3	3/9	3/1	3/5	3/1	3/1	3/2	3/5	3/3	3/2
GraphWalker	9/3	9/9	9/1	9/5	9/1	9/1	9/2	9/5	9/3	9/2
JSXM	1/3	1/9	1/1	1/5	1/1	1/1	1/2	1/5	1/3	1/2
MISTA	5/3	5/9	5/1	5/5	5/1	5/1	5/2	5/5	5/3	5/2
ModBat	1/3	1/9	1/1	1/5	1/1	1/1	1/2	1/5	1/3	1/2
ModelJunit	1/3	1/9	1/1	1/5	1/1	1/1	1/2	1/5	1/3	1/2
MoMuT:UML	2/3	2/9	2/1	2/5	2/1	2/1	2/2	2/5	2/3	2/2
OSMO	5/3	5/9	5/1	5/5	5/1	5/1	5/2	5/5	5/3	5/2
TCases	3/3	3/9	3/1	3/5	3/1	3/1	3/2	3/5	3/3	3/2
JTorX	2/3	2/9	2/2	2/5	2/1	2/1	2/2	2/5	2/3	2/2

Tabela 39 - Matriz de comparação de alternativas com critério Suporte com soma por colunas

	fMBT	GraphWalker	JSXM	MISTA	ModBat	ModelJunit	MoMuT:UML	OSMO	TCases	JTorX
fMBT	1.0	0.3	3.0	0.6	3.0	3.0	1.5	0.6	1.0	1.5
GraphWalker	3.0	1.0	9.0	1.8	9.0	9.0	4.5	1.8	3.0	4.5
JSXM	0.3	0.11	1.0	0.2	1.0	1.0	0.5	0.2	0.3	0.5
MISTA	1.6	0.5	5.0	1.0	5.0	5.0	2.5	1.0	1.6	2.5
ModBat	0.3	0.11	1.0	0.2	1.0	1.0	0.5	0.2	0.3	0.5
ModelJunit	0.3	0.11	1.0	0.2	1.0	1.0	0.5	0.2	0.3	0.5
MoMuT:UML	0.6	0.22	2.0	0.4	2.0	2.0	1.0	0.4	0.6	1.0
OSMO	1.6	0.5	5.0	1.0	5.0	5.0	2.5	1.0	1.6	0.5
TCases	1.0	0.3	3.0	0.6	3.0	3.0	1.5	0.6	1.0	1.0
JTorX	0.6	0.5	1.0	0.4	2.0	2.0	1.0	0.4	0.6	1.5
Soma por colunas	10.3	3.65	31	6.4	32	32	16	6.4	10.3	14

Tabela 40 - Matriz de comparação de alternativas com critério Suporte normalizada com média

	fMBT	GraphWalker	JSXM	MISTA	ModBat	ModelJunit	MoMuT:UML	OSMO	TCases	JTorX	Média= Prioridade
fMBT	0.09	0.08	0.09	0.093	0.09	0.09	0.09	0.093	0.09	0.10	0.090
GraphWalker	0.29	0.27	0.29	0.28	0.28	0.28	0.28	0.28	0.29	0.32	0.286
JSXM	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.02	0.03	0.028
MISTA	0.15	0.13	0.16	0.15	0.15	0.15	0.15	0.15	0.15	0.17	0.151
ModBat	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.02	0.03	0.028
ModelJunit	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.02	0.03	0.028
MoMuT:UML	0.05	0.06	0.06	0.062	0.06	0.06	0.06	0.062	0.05	0.07	0.059

OSMO	0.15	0.13	0.16	0.15	0.15	0.15	0.15	0.15	0.15	0.03	0.137
TCases	0.09	0.08	0.09	0.093	0.09	0.09	0.09	0.093	0.09	0.07	0.087
JTorX	0.05	0.13	0.03	0.062	0.06	0.06	0.06	0.062	0.05	0.10	0.066

Para o critério suporte a prioridade em relação às ferramentas é do *GraphWalker* com pontuação igual a 0.286 superior a todas as outras ferramentas.

A última fase de utilização do método AHP, consiste no estabelecimento das prioridades compostas para as alternativas. Cada valor relativo à prioridade das ferramentas em relação aos critérios, é multiplicado pela prioridade dos critérios. Após o somatório de cada linha são obtidas as prioridades compostas.

Tabela 41 - Matriz de obtenção de prioridades compostas

	Eficiência	**Usabilidade**	**Multiplataforma**	**Escalabilidade**	**Integração contínua**	**Suporte**	**Prioridade dos critérios**	**Prioridades compostas**
fMBT	0.085	0.071	0.034	0.05	0.029	0.090	0.18 0.16 0.16 0.15 0.15 0.14	0.10551
GraphWalker	0.15	0.321	0.149	0.27	0.296	0.286		0.22714
JSXM	0.099	0.037	0.034	0.162	0.187	0.028		0.08545
MISTA	0.13	0.080	0.149	0.05	0.029	0.151		0.09303
ModBat	0.13	0.024	0.149	0.162	0.029	0.028		0.08365
ModelJunit	0.016	0.078	0.149	0.05	0.158	0.028		0.07432
MoMuT:UML	0.064	0.112	0.040	0.05	0.029	0.059		0.05595
OSMO	0.096	0.120	0.112	0.05	0.029	0.137		0.08543
TCases	0.081	0.059	0.049	0.05	0.142	0.087		0.07284
JTorX	0.12	0.048	0.12	0.05	0.029	0.066		0.06957

O GraphWalker surge como a ferramenta mais indicada para utilização no caso prático com pontuação de 0.22714 bastante superior em relação às restantes ferramentas. Tendo por base, os critérios definidos e as respetivas comparações e prioridades o GraphWalker foi a ferramenta selecionada para alvo de estudo no caso prático.

4.4 Ferramenta selecionada em detalhe

Nesta secção serão abordados de forma detalhada todos os pontos mais importantes relacionados com a utilização da ferramenta selecionada. Esta secção foi subdividida em cinco pontos que serão utilizados para explicar em detalhe como funciona o *GraphWalker*, quais as suas funcionalidades, como é possível desenhar o modelo, que algoritmos de geração de sequências de testes e condições de paragem permite utilizar, que anotações e parametrizações são permitidas, bem como, os plugins que possui e que facilitam a sua utilização. Toda a informação relacionada com os detalhes da ferramenta foi obtida a partir da referência [25] que corresponde à página oficial do *GraphWalker*.

4.4.1 Funcionalidades do *GraphWalker*

O *GraphWalker* é uma ferramenta que suporta o *Model-Based Testing*. Através da leitura de modelos construídos na forma de grafos orientados, gera sequências de testes. Na representação dos modelos as setas representam uma ação e as caixas uma verificação, tal como apresentado no exemplo do modelo abaixo.

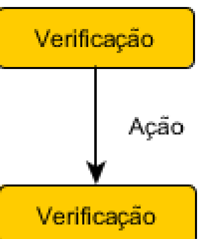


Figura 10 - Legenda elementos do modelo

Utilizando um modelo e uma regra de geração, através de algoritmos matemáticos o *GraphWalker*, possui a capacidade de percorrer e gerar caminhos aleatórios pelo modelo que correspondem a sequências de testes. A figura seguinte, ilustra um exemplo de um modelo e as setas sublinhadas a vermelho um caminho gerado. Posteriormente através da implementação de código, o caminho sublinhado, poderá ser executado, até cumprir uma condição de paragem, de forma automática.

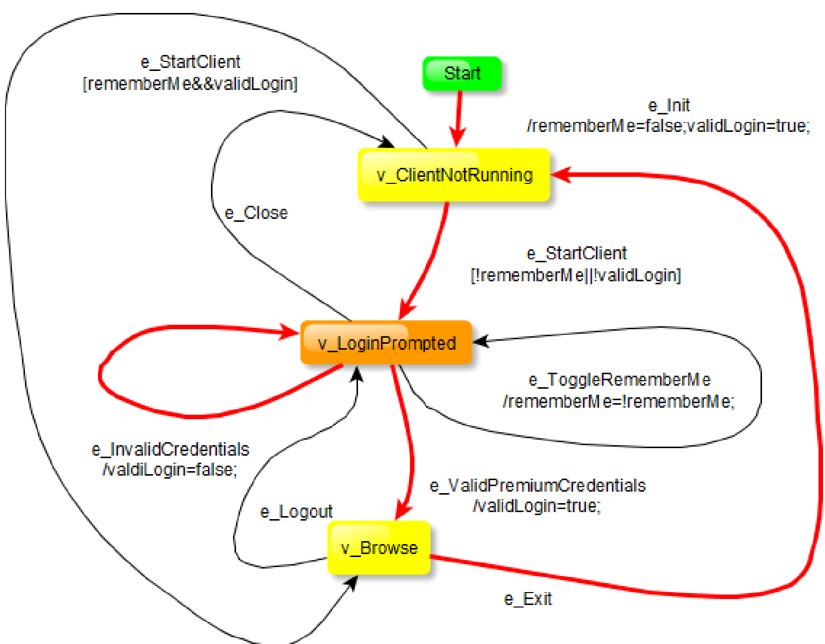


Figura 11 - Exemplo de um caminho gerado num modelo

A especificação de casos de teste utilizando o *Model-Based Testing* em conjunto com o *GraphWalker,* pode ser realizada utilizando um ou mais modelos. Em cada modelo deverá existir um gerador de sequências de testes e uma condição de paragem. A execução termina apenas quando for cumprida a condição de paragem para todos os modelos. Cada caminho percorrido pelo modelo representa um caso de teste.

Uma seta no modelo representa qualquer situação que faça o sistema alterar-se para um novo estado que é necessário verificar. As caixas representam uma verificação e apenas nestes elementos poderá ser realizada uma verificação e nunca nas setas do modelo.

As gerações de sequências de testes podem acontecer de forma *offline* e *online*. De forma *offline* é realizada apenas uma vez sem estar diretamente ligada a nenhum código de testes automatizados. O caminho é gerado na linha de comandos e os resultados são guardados num ficheiro que posteriormente é utilizado para a realização dos testes automatizados. A geração *online* é realizada durante a execução dos testes onde o *GraphWalker* já se encontra integrado com o código dos testes automatizados não sendo necessário lidar com as sequências dos testes em ficheiros. O código dos testes automatizados possui acesso direto ao contexto de execução do modelo, que é onde se encontra guardada toda a informação da geração da sequência de testes.

Existe sempre a necessidade de serem implementados testes automatizados com outra ferramenta que interaja com o SUT, pois o *GraphWalker* não interage com o SUT. A construção dos modelos é realizada utilizando uma ferramenta externa.

A seleção dos testes é realizada utilizando regras especiais. Essas regras podem ser exemplificadas e divididas em duas partes. Como executar o algoritmo de geração de sequências de testes e o que cobrir durante essa execução. A forma como é descrita a utilização do gerador de sequências de testes é realizada da seguinte forma: "**gerador (condição_de_paragem (condição))**". No contexto do trabalho prático, o gerador das sequências de testes, é utilizado na classe de execução dos testes. Um exemplo prático da sua utilização, de acordo com a seguinte descrição "`setPathGenerator (new RandomPath (new EdgeCoverage (100)))`, corresponde à geração de sequências de teste de forma aleatória até percorrer 100% das setas que se encontram descritas no modelo.

O *GraphWalker* pode trabalhar com vários modelos dentro de um mesmo projeto. Durante a geração de um caminho, pode escolher saltar para outro. Esta funcionalidade pode ser bastante útil para separar várias funcionalidades em diferentes modelos. Pode ser utilizada através de palavras-chave durante a criação dos modelos.

O *GraphWalker* pode ser utilizado como um serviço *Restful* e como um serviço de *WebSocket*. Contém para cada um deles uma API para permitir a interação.

4.4.2 Regras de modelação

O objetivo da construção do modelo é o de descrever o comportamento de um SUT. O resultado traduz-se num modelo constituído por setas e por caixas e a forma como interagem. O modelo relembra um diagrama de estados ou de uma máquina de estados finitos, onde a seta corresponde a uma ação e a caixa a um estado. Abaixo, na figura 12 segue um exemplo bastante simples de como poderá ser construído um modelo.

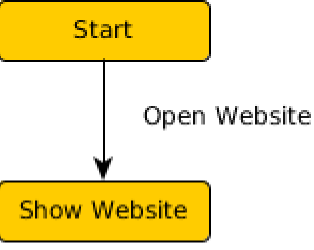


Figura 12 - Exemplo de um modelo simplificado

As caixas representam um estado a verificar. Normalmente na implementação dos testes automatizados podem corresponder a validações, utilizando "*assertions*". O *GraphWalker* não utiliza as cores, nem o formato da caixa no modelo para efetuar validações. Normalmente é utilizado o formato de uma caixa para representação dos estados. Caso fossem utilizados noutro formato seriam aceites pelo *GraphWalker*.

As setas representam uma transição de uma caixa para outra e podem ser, uma qualquer verificação que é necessário realizar para chegar ao próximo estado no modelo. Normalmente na implementação dos testes automatizados podem corresponder por exemplo, à abertura de um site, seleção de um menu ou o clique de um botão. O *Graphwalker* apenas aceita que as setas no modelo sigam uma direção e também não faz nenhuma validação no que diz respeito às cores ou à espessura que as setas poderão ter.

Existem algumas regras mais específicas que dizem respeito à criação dos modelos. A primeira caixa de início não é de uso obrigatório, a ser utilizada deverá ser única e ser renomeada por "*Start*". Apenas pode existir uma seta a sair da caixa de "*Start*" e esta não será incluída em nenhum caminho gerado.

O nome existente nas caixas e nas setas é oriundo de uma etiqueta que lhes permite associar texto. Corresponde à primeira palavra, ou à primeira linha que a etiqueta contém.

As setas podem ainda conter dois mecanismos que permitem adicionar validações extra, os "*Guards*" e as "*Action*". Os "*Guards*" funcionam como uma declaração programática "*if*" que determina se a seta no modelo pode ser percorrida ou não. Pode ser descrita da seguinte forma: [*loggerdIn* == *true*]. As "*Actions*" correspondem a código que queremos executar e pode ser descrito da seguinte forma: /*loggedIn*=false; *rememberMe*=*true*, o seu objetivo é servir de informação aos "*Guards*". Os "*Guards*" e as "*Actions*" são também utilizadas para permitir a partilha de atributos entre modelos e podem ser descritos da seguinte forma para as "*Actions*" e para as "*Guards*" respetivamente: [*global.loggedIn=false*; *global.rememberMe=true]*, [*global.loggedIn* == *true*].

Algumas palavras-chave podem ser utilizadas no modelo para aumentar as suas funcionalidades e a sua usabilidade. As palavras-chave são:

- **BLOCKED** - Uma seta ou uma caixa que contenha esta palavra-chave, será excluída quando um caminho for gerado. Se for uma caixa, será removida do modelo, mas se for uma seta, será removida conjuntamente com as suas caixas de verificação de entrada e de saída.

- **SHARED** - Esta palavra-chave é apenas utilizada em caixas no modelo. Ao ser utilizado o *GraphWalker* pode saltar do modelo atual, para qualquer outro modelo, desde que a caixa para onde vai saltar, contenha o mesmo nome SHARED, sendo esta decisão realizada de forma aleatória. É descrita da seguinte forma: SHARED: SOME_NAME.
- **INIT** - Apenas uma caixa pode ter esta palavra-chave. Ao utilizar dados num modelo, são inicializados através da utilização desta palavra-chave. Pode ser utilizada em mais do que uma caixa e é descrita da seguinte forma: INIT: *loggedIn* = *false*; *rememberMe* = *true*;
- **REQTAG** - Apenas uma caixa pode conter esta palavra-chave. É utilizada para criar rastreabilidade com requisitos externos e os modelos. É descrita da seguinte forma: REQTAG: String1, String2.
- **weight -** Apenas uma seta pode conter esta palavra-chave e é utilizada unicamente em conjunto com o algoritmo de geração de sequências de teste de forma aleatória. Permite guardar um valor real que determina a probabilidade de uma seta no modelo ser percorrida ou não. Pode ser descrita da seguinte forma: *weight=float value between 0.0 and 1.0*.

4.4.3 Geradores e condições de paragem

Os geradores são algoritmos de geração de sequências de teste que em conjunto com as condições de paragem decidem como será percorrido o modelo. Os geradores podem ser utilizados de forma encadeada uns após os outros, bem como, as várias condições de paragem que podem ser utilizadas através dos operadores lógicos "OR","AND", "||" e "&&".
Os geradores suportados pelo *GraphWalker* e as suas funções são:

- **random(condições_de_paragem)**

A sua execução funciona através da seleção de uma seta ligada a uma caixa de forma aleatória, repetindo o processo na próxima caixa.

- **weighted_random(condições_de_paragem)**

Funciona da mesma forma que o algoritmo **random** mas, utiliza a palavra-chave **weight** para representar a probabilidade de uma caixa ser percorrida ou não pelo modelo.

- **quick_random(condições_de_paragem)**

Tenta percorrer o caminho, mas curto pelo modelo. Escolhe uma caixa não visitada de forma aleatória, seleciona o caminho mais curto para essa caixa utilizando o algoritmo "Dijkstra".

Percorre o caminho e marca as caixas percorridas como visitadas. Quando chega à caixa inicialmente selecionada, recomeça e repetindo todos os passos anteriores.

- **a_star(condição_de_paragem cujo nome é uma seta ou uma caixa)**

Gera o caminho mais curto até uma caixa ou seta do modelo.

As condições de paragem suportadas pelo *GraphWalker* e as suas funções são:

- **edge_coverage (inteiro que representa a percentagem de cobertura das setas no modelo)**

A condição de paragem é um inteiro que representa a percentagem de cobertura das setas no modelo durante a execução do gerador.

- **vertex_coverage (inteiro que representa a percentagem de cobertura das caixas no modelo)**

A condição de paragem é um inteiro que representa a percentagem de cobertura das caixas no modelo durante a execução do gerador.

- **requirement_coverage (inteiro que representa a percentagem de cobertura de requisitos)**

A condição de paragem é um inteiro que representa a percentagem de cobertura dos requisitos no modelo durante a execução do gerador.

- **dependency_edge_coverage (inteiro que representa o limite da dependência)**

A condição de paragem é um inteiro que representa o limite da dependência no modelo durante a execução do gerador.

- **reached_vertex (nome da caixa para atingir)**

A condição de paragem é uma caixa que se pretende atingir no modelo durante a execução do gerador.

- **reached_edge (nome da seta para atingir)**

A condição de paragem é uma seta que se pretende atingir no modelo durante a execução do gerador.

- **time_duration (inteiro que representa os segundos para percorrer o modelo)**

A condição de paragem é um inteiro que representa os segundos que o gerador deverá demorar a percorrer o modelo.

- **length(inteiro)**

A condição de paragem é um inteiro que representa o número total de pares de setas e caixas gerados na execução do gerador.

- **never**

A condição de paragem nunca interromperá a execução do gerador.

4.4.4 Anotações

Normalmente é necessário realizar uma preparação do ambiente onde vão ser executados os testes. Tal como em ferramentas como o *JUnit* e o *TestNG*, também o *GraphWalker* possui as mesmas funcionalidades na seguinte lista de anotações:

- **@BeforeExecution** - o método anotado poderá ser utilizado antes da execução do *GraphWalker*. Por exemplo, poderia ser utilizado para abrir uma página de um browser ou iniciar um servidor de testes.
- **@AfterExecution** - o método anotado poderá ser utilizado depois da execução do *GraphWalker*. Por exemplo, para recolher alguns resultados das execuções dos testes, como logs, relatórios entre outros.
- **@BeforeElement** - o método anotado poderá ser utilizado antes de qualquer invocação de elemento (caixa ou seta) na execução do *GraphWalker*.
- **@AfterElement** - o método anotado poderá ser utilizador após qualquer chamada de elemento (caixa ou seta) na execução do *GraphWalker*.

4.4.5 Plugins

O *GraphWalker* contém vários plugins que executam algumas funções necessárias ao seu correto funcionamento. Na lista seguinte encontram-se os plugins e as suas funcionalidades:

- **graphwalker:generate-sources e graphwalker:generate-test-sources**

Os dois plugins geram interfaces a partir dos modelos e colocam-nas numa pasta dedicada no projeto **src/main/resources**.

- **graphwalker:test**

Executa o plugin **generate-sources** olha para as classes que contenham a anotação @GraphWalker e executa-as como testes utilizando o *GraphWalker*.

- **graphwalker:validate-models e graphwalker:validate-test-models**

Os dois plugins validam os modelos que existam na pasta **src/main/resources.**

- **graphwalker:watch**

O plugin olha aos modelos colocados na pasta **src/main/resources** e se existirem alterações executa o plugin **graphwalker:generate-sources** para gerar as interfaces com as novas alterações a partir dos modelos.

CAPÍTULO 5

COMPONENTE PRÁTICA

Neste capítulo é apresentado um caso de estudo sobre a utilização da metodologia MBT. A ferramenta selecionada para apresentar este caso prático a suportar a metodologia MBT foi o *GraphWalker*. Foram apresentados todos os passos da sua execução em detalhe seguindo como metodologia de investigação foi a metodologia de investigação em ação.

5.1 Metodologia utilizada

Devido à elevada exigência pretendida na realização do caso prático apresentado nesta tese, foi utilizada uma metodologia de estudo, a metodologia da investigação em ação.

A metodologia da investigação em ação, é caracterizada como uma investigação em ação em vez de uma investigação sobre a ação. Utiliza uma abordagem científica para estudar a resolução de problemas importantes, sociais ou organizacionais, em conjunto com os intervenientes que experienciam estes problemas de forma direta. É considerada uma parceria colaborativa na qual, os membros do sistema a ser estudado participam ativamente no processo cíclico que faz parte desta metodologia. É uma pesquisa concorrente com a ação, pois, o seu objetivo é tornar a ação mais eficaz enquanto é gerado conhecimento científico. É uma abordagem para a resolução de problemas e uma sequência de eventos cujos resultados não são apenas soluções para os problemas. Mas sim a aprendizagem que surge a partir de resultados intencionais e não intencionais, bem como, uma contribuição para a teoria e para o conhecimento científico. A metodologia da investigação em ação é então designada como uma abordagem que tem como principal objetivo agir e criar conhecimento ou uma teoria sobre a ação realizada. Esta metodologia difere das abordagens tradicionais de pesquisa cujo objetivo é apenas o de criar conhecimento [23].

A investigação em ação é constituída por quatro principais etapas que acontecem seguindo a ordem [23]:

1. **Diagnóstico**: Na fase de diagnóstico são identificadas as questões que irão definir o tema do trabalho com base nas ações que serão planeadas e executadas.
2. **Planeamento da ação**: A fase de planeamento é criada a partir da análise do contexto e propósito do projeto, do problema principal e das questões identificadas na fase do diagnóstico.

3. **Execução da ação**: Na fase de execução da ação o plano é implementado.
4. **Avaliação da ação**: Na fase de avaliação todas as ações intencionais e não intencionais são examinadas com o objetivo de verificar se o diagnóstico realizado estava correto, se a ação efetuada estava correta e se foi executada corretamente. Na fase de avaliação são também identificados os dados a serem utilizados no próximo ciclo de diagnóstico, planeamento e ação. É também definido um novo ciclo de aprendizagem experimental em cada uma das etapas que acontecem na metodologia da investigação em ação.

Este ciclo de aprendizagem também ele é constituído por quatro fases, experimentar, refletir, interpretar e agir.

Durante a realização do caso prático, foram tomadas decisões tendo em conta as etapas que ocorrem e pelas quais um projeto, utilizando a metodologia da investigação em ação. Contextualizando o caso prático tendo em conta as diferentes fases da metodologia em ação, na fase do diagnóstico no âmbito do caso prático, foram identificadas as seguintes questões que direcionaram a investigação:

Questão 1: Como pode ser aplicada a metodologia *Model-Based Testing* utilizando cenários de trabalho real?

Questão 2: Quais as vantagens e as desvantagens da metodologia MBT em relação à metodologia BDD?

Na fase do planeamento, com base nas questões identificadas, foi tomada a decisão de que os cenários e a metodologia utilizada para comparação no caso prático seriam cenários já conhecidos e experienciados em âmbito de trabalho real. Bem como, as tecnologias e linguagens de programação utilizadas para a implementação dos testes automatizados.

O caso prático foi então dividido em três partes cada uma delas com um cenário de utilização diferente. Uma das partes apresentada com a utilização de um website, a outra foi com a de uma aplicação móvel e a outra com serviços *web*.

Na fase de execução da ação foi então seguido e colocado o plano em prática. Durante esta fase surgiram várias situações, mais à frente descritas na análise dos resultados práticos, que passaram pelas diferentes fases da aprendizagem experimental a experimentação, reflexão e a interpretação. E que deram origem a conhecimento sobre o funcionamento da metodologia e da ferramenta que a suporta, bem como, à tomada de decisões no processo de execução da ação.

Na fase de avaliação foi verificada a viabilidade dos resultados e também nesta fase foi gerado conhecimento importante, descrito também mais à frente na análise dos resultados práticos.

5.1.1 Cenário prático utilizando website

Para apresentar o cenário foi utilizado o website www.sapo.pt e a sua funcionalidade e provedor de email disponibilizado. Para facilitar a explicação e o detalhe relativamente à implementação prática do cenário utilizando um website, a descrição foi dividida em cinco pontos. Criação do projeto com o *GraphWalker*, verificação da correta criação do modelo, criação da classe de execução de testes, criação dos testes automatizados e configuração do projeto no *Jenkins*.

5.1.1.1 Criação de projeto com *GraphWalker*

O projeto criado foi do tipo *Maven*, através da execução do seguinte comando:

➔ C:\Users\manue>mvn archetype:generate -B -DarchetypeGroupId=org.graphwalker -DarchetypeArtifactId=graphwalker-maven-archetype -DgroupId=com.company -DartifactId=testAutomation_GraphWalker_MasterThesis

```
C:\Users\manue>mvn archetype:generate -B -DarchetypeGroupId=org.graphwalker -DarchetypeArtifactId=graphwalker-maven-archetype -DgroupId=com.company -DartifactId=testAut
omation_GraphWalker_MasterThesis
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:3.0.1:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:3.0.1:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:3.0.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] Archetype [org.graphwalker:graphwalker-maven-archetype:3.4.2] found in catalog remote
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: graphwalker-maven-archetype:3.4.2
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.company
[INFO] Parameter: artifactId, Value: testAutomation_GraphWalker_MasterThesis
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.company
[INFO] Parameter: packageInPathFormat, Value: com/company
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.company
[INFO] Parameter: groupId, Value: com.company
[INFO] Parameter: artifactId, Value: testAutomation_GraphWalker_MasterThesis
[INFO] Project created from Archetype in dir: C:\Users\manue\testAutomation_GraphWalker_MasterThesis
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 24.423 s
[INFO] Finished at: 2018-05-28T22:25:33+01:00
[INFO] Final Memory: 12M/136M
[INFO] ------------------------------------------------------------------------
C:\Users\manue>
```

Figura 13 - Resultado da criação do projeto com sucesso

Após criação do projeto com sucesso, como é possível visualizar pela figura 13, devido a um defeito conhecido, a informação do ficheiro POM.xml teve de ser editada. Foi necessário adicionar a informação do plugin mencionada no trecho de código 1. Então nesse momento após edição, o projeto assumiu a versão correta e reconheceu a utilização do *GraphWalker*.

```
Trecho de código 1 – Dependência a ser adicionada para projeto assumir utiliz
ação do GraphWalker

  <build>
         <plugins>
             <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
            <plugin>
                 <groupId>org.graphwalker</groupId>
                 <artifactId>graphwalker-maven-plugin</artifactId>
                 <version>${graphwalker.version}</version>
```

O projeto foi criado por defeito com a classe exemplo **SomeSmallTest.java** e com o respetivo modelo SmallTest.graphml. A classe e o modelo foram eliminadas. O modelo correto foi importado, tendo sido previamente criado utilizando a ferramenta *yEd*, para a pasta *resources*. De seguida foi criada a classe LoginTest.java para conter os métodos gerados. Sendo os métodos gerados a partir de uma classe interface, também ela gerada automaticamente a partir do modelo. Para a geração da classe interface, foi executado o seguinte comando:

➔ mvn graphwalker:generate-sources

O *GraphWalker* reconhece então a classe interface **LoginSearch.java** e permite a implementação de forma automática dos métodos. No trecho de código 2 é possível visualizar um exemplo com a criação da classe **LoginTest.java** e com a interface gerada **LoginSearch.java**.

```
Trecho de código 2 - Estrutura da classe de teste após geração das
dependências necessárias

@GraphWalker
public class LoginTest extends ExecutionContext implements LoginSearch
{

}
```

O projeto deverá ficar com a estrutura abaixo apresentada na figura 14. E deverá permitir implementar os métodos conforme ilustrado pela figura 15.

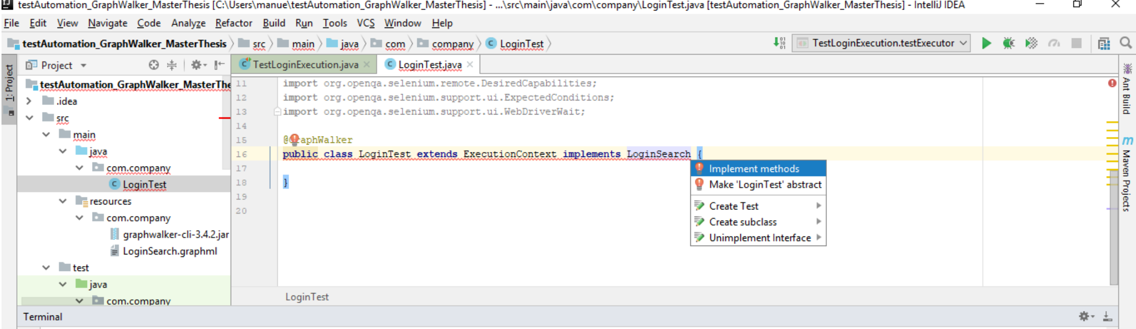


Figura 14 - Implementação de métodos de forma automática a partir da interface gerada a partir do modelo

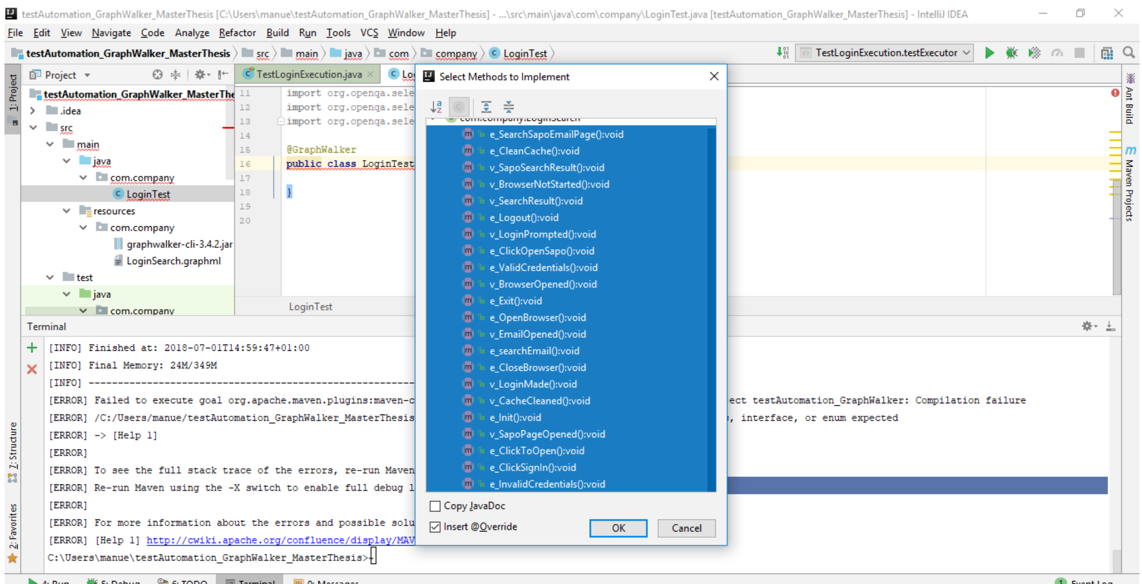


Figura 15 - Listagem dos métodos contidos pela classe da interface

Em cada método gerado foi posteriormente implementado o código para execução dos testes de forma automática, sempre que invocados os métodos, utilizando os diferentes algoritmos de geração de sequências de testes, suportados pelo *GraphWalker*.

No trecho de código 3, é possível visualizar o exemplo completo. A classe de testes LoginTest.java a implementar a classe interface LoginSearch.java, gerada automaticamente a partir do modelo, e já com a implementação dos métodos gerados a partir da classe interface.

Trecho de código 3 – Classe LoginTest.java após implementação dos métodos

```
package com.company;
import org.graphwalker.core.machine.ExecutionContext;
import org.graphwalker.java.annotation.GraphWalker;
@GraphWalker
public class LoginTest extends ExecutionContext implements LoginSearch {
    @Override
    public void e_SearchSapoEmailPage() {
    }
    @Override
    public void e_CleanCache() {
    }
    @Override
    public void v_SapoSearchResult() {
    }
    @Override
    public void v_BrowserNotStarted() {
    }
    @Override
    public void v_SearchResult() {
    }
    @Override
    public void e_Logout() {
    }
    @Override
    public void v_LoginPrompted() {
    }
    @Override
    public void e_ClickOpenSapo() {
    }
    @Override
    public void e_ValidCredentials() {
    }
    @Override
    public void v_BrowserOpened() {
    }
    @Override
    public void e_Exit() {
    }
    @Override
    public void e_OpenBrowser() {
    }
    @Override
    public void v_EmailOpened() {
    }
    @Override
    public void e_searchEmail() {
    }
    @Override
    public void e_CloseBrowser() {
    }
    @Override
    public void v_LoginMade() {
    }
    @Override
    public void v_CacheCleaned() {
    }
    @Override
    public void e_Init() {
    }
    @Override
    public void v_SapoPageOpened() {
    }
    @Override
    public void e_ClickToOpen() {
    }
    @Override
    public void e_ClickSignIn() {
    }
    @Override
    public void e_InvalidCredentials() {
    }
}
```

5.1.1.2 Verificação da criação correta do modelo

O modelo foi criado utilizando a ferramenta *yEd*. A ferramenta permite criar os modelos de forma simples e rápida. Permite ainda exportar diretamente para o formato que é lido pelo *GraphWalker* .graphml. A modelação aceite pelo *GraphWalker* é feita com base em grafos orientados. Esta forma de modelação corresponde à notação de máquinas de estados finitas e das máquinas de estados finitas estendidas. O modelo LoginSearch.graphml representado na figura 16 corresponde ao fluxo de execução de três funcionalidades do website:

1. Efetuar login no sapo mail;
2. Após login -> Efetuar uma pesquisa por um email;
3. Abrir o email pesquisado;

Surgiram algumas situações, durante a execução do caso prático, em que foi necessário alterar o modelo. Para realizar essa alteração foram seguidos os seguintes passos:

1. As funcionalidades descritas no modelo foram alteradas utilizando a ferramenta *yEd*;
2. Após alteração foi novamente importado para a pasta do projeto, mencionada nos passos anteriores;
3. Foi executado o comando de geração das dependências da classe interface, mencionado nos passos anteriores;
4. Foram implementados os novos métodos gerados a partir da classe interface;
5. Foi implementado o código dos testes automatizados para os novos métodos gerados;

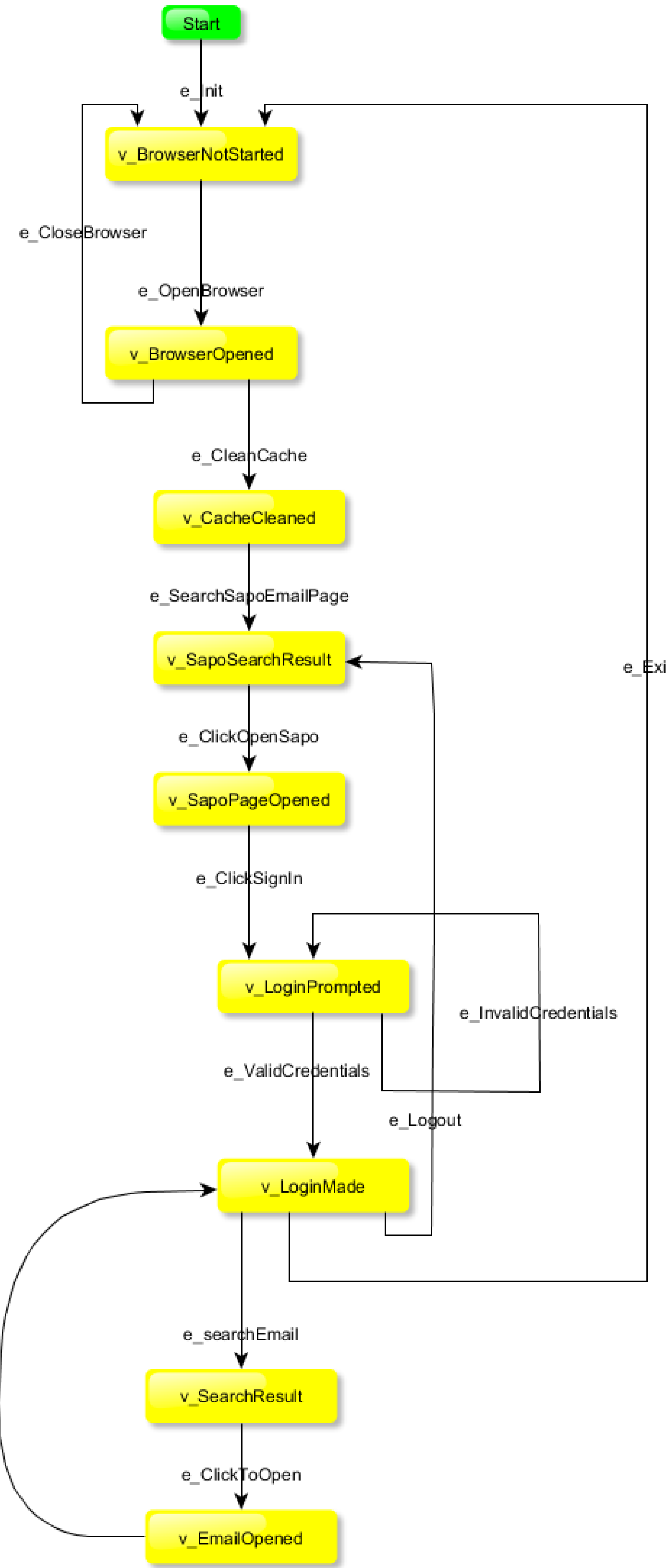


Figura 16 - Modelo representativo das funcionalidades de login e pesquisa no website

O ficheiro .jar foi importado para a localização no projeto onde se encontra o modelo, para permitir a verificação da viabilidade do modelo.

- Em modo *offline*, foi necessário executar o comando apresentado no trecho de código 4.

```
Trecho de código  4 – Comando para validação do modelo em modo offline

java -jar graphwalker-cli-3.4.2.jar offline --start-element e_Init --model LoginSearch.graphml "random(edge_coverage(100))";
```

Juntamente com a indicação pela qual é pretendido que sejam geradas as sequências de testes. Sendo que o gerador de sequências de testes utilizado no trecho de código 4 poderá ser outro qualquer de acordo com o pretendido.

```
Trecho de código 5 – Classe LoginTest.java com gerador GraphWalker embutido

"@GraphWalker(value="random(vertex_coverage(100))",start= "e_Init")
public class LoginTest extends ExecutionContext implements Login {"
```

.

- Em modo *online*, dentro da pasta do projeto onde se encontra a classe com o gerador de sequências de testes embutido, como exemplo, apresentado no trecho de código 5. Deverá ser executado o comando apresentado no trecho de código 6.

```
Trecho de código  6 – Comando para validação do modelo em modo online

mvn graphwalker:test;
```

O resultado da validação do modelo deverá ser algo semelhante ao apresentado no trecho de código 7, de acordo com o gerador de sequências de testes utilizado.

Trecho de código 7 - Resultado da validação do modelo


```
[INFO] Result :
 [INFO] {
  "totalFailedNumberOfModels": 0,
  "totalNotExecutedNumberOfModels": 0,
  "totalNumberOfUnvisitedVertices": 0,
  "verticesNotVisited": [],
  "totalNumberOfModels": 1,
  "totalCompletedNumberOfModels": 1,
  "totalNumberOfVisitedEdges": 12,
  "totalIncompleteNumberOfModels": 0,
  "edgesNotVisited": [
    {
      "modelName": "Login",
      "edgeId": "e10",
      "edgeName": "e_CloseBrowser"
    },
    {
      "modelName": "Login",
      "edgeId": "e13"
    }
  ],
  "vertexCoverage": 100,
  "totalNumberOfEdges": 14,
  "totalNumberOfVisitedVertices": 9,
  "edgeCoverage": 85,
  "totalNumberOfVertices": 9,
  "totalNumberOfUnvisitedEdges": 2
}
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 9.936 s
[INFO] Finished at: 2018-06-17T12:05:46+01:00
[INFO] Final Memory: 23M/310M
[INFO] ------------------------------------------------------------------------
```

5.1.1.3 Criação de classe de execução de testes

Foi criada uma classe para permitir a execução dos testes, utilizando diferentes algoritmos de geração de sequências de testes. Para que esta classe fosse reconhecida, como sendo uma classe de testes, foi criada de acordo com a estrutura abaixo na figura 17. A pasta java, onde se encontrava a classe de execução de testes, foi marcada como "*Test Sources root*".

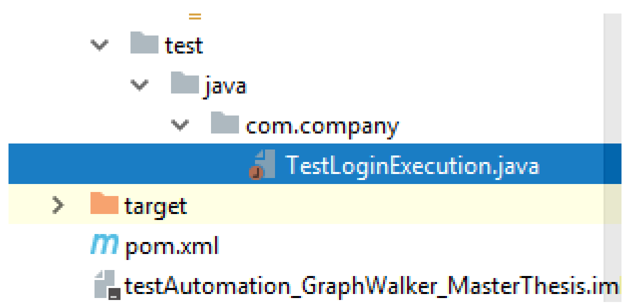


Figura 17 - Classe execução de testes

O método *testExecutor* é utilizado para executar a geração das sequências de testes, que no caso apresentado pelo trecho de código 8, utiliza o algoritmo de geração de sequências de testes "`new RandomPath (new EdgeCoverage (100))`".

```
Trecho de código 8 – Conteúdo da classe de execução de testes

public class TestLoginExecution {
    @Test
    public void testExecutor() throws IOException {
        Executor executor = new TestExecutor(LoginTest.class);

executor.getMachine().getCurrentContext().setPathGenerator(new
RandomPath(new EdgeCoverage(100)));
        Result result = executor.execute(true);
        if (result.hasErrors()) {
            for (String error : result.getErrors()) {
                System.out.println(error);
            }
        }
        System.out.println("Done: [" +
result.getResults().toString(2) + "]");
    }
}
```

Para proceder à execução dos casos de teste, ainda que sem implementação dos testes automatizados, é utilizada uma anotação **@Test**. Esta anotação é oriunda da *framework* de testes o ***JUnit.*** Possibilita a execução do método e a geração das sequências de testes utilizando a estrutura apresentada na figura 18.

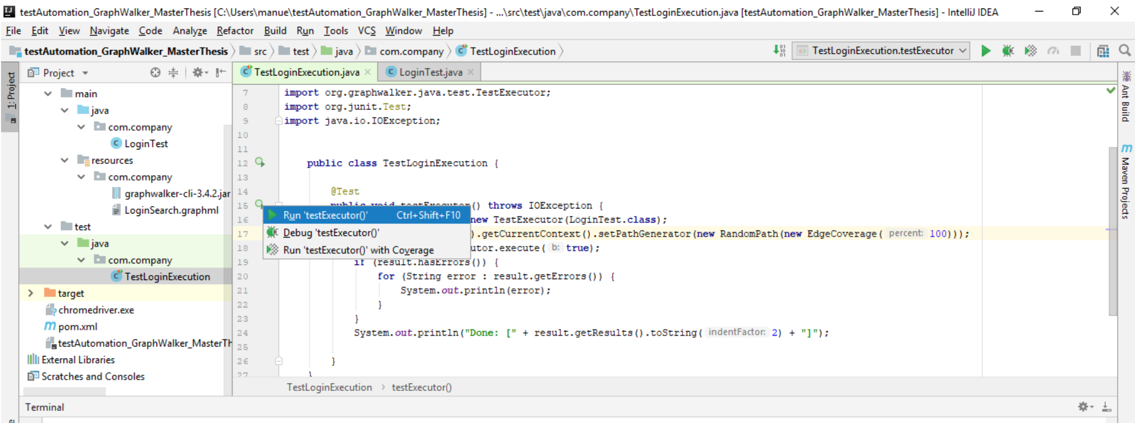


Figura 18 - Execução dos testes utilizando anotação @*Test*

É possível visualizar um exemplo de um resultado de uma execução, ainda sem existir a implementação dos testes automatizados, com sucesso no trecho de código 9.

Trecho de código 9 – Resultado da execução dos testes


```
Done: [{
  "totalFailedNumberOfModels": 0,
  "totalNotExecutedNumberOfModels": 0,
  "totalNumberOfUnvisitedVertices": 0,
  "verticesNotVisited": [],
  "totalNumberOfModels": 1,
  "totalCompletedNumberOfModels": 1,
  "totalNumberOfVisitedEdges": 14,
  "totalIncompleteNumberOfModels": 0,
  "edgesNotVisited": [],
  "vertexCoverage": 100,
  "totalNumberOfEdges": 14,
  "totalNumberOfVisitedVertices": 9,
  "edgeCoverage": 100,
  "totalNumberOfVertices": 9,
  "totalNumberOfUnvisitedEdges": 0
}]
```


5.1.1.4 Implementação dos testes automatizados

O passo 4 corresponde à implementação dos testes automatizados. A sua implementação é importante para permitir a execução automática dos testes durante a geração das sequências de teste. A classe com os testes automatizados implementados pode ser visualizada no anexo A.

5.1.1.5 Configuração do projeto no *Jenkins*

A configuração de um projeto de testes automatizados utilizando o *GraphWalker* no *Jenkins*, foi relativamente simples. Foram seguidos os seguintes passos:

1. Instalação do plugin que habilita a criação de projetos *Maven*;
2. Criação do job do tipo projeto *Maven*;
3. Efetuar o preenchimento das configurações do job de acordo com as configurações apresentadas na figura 19;

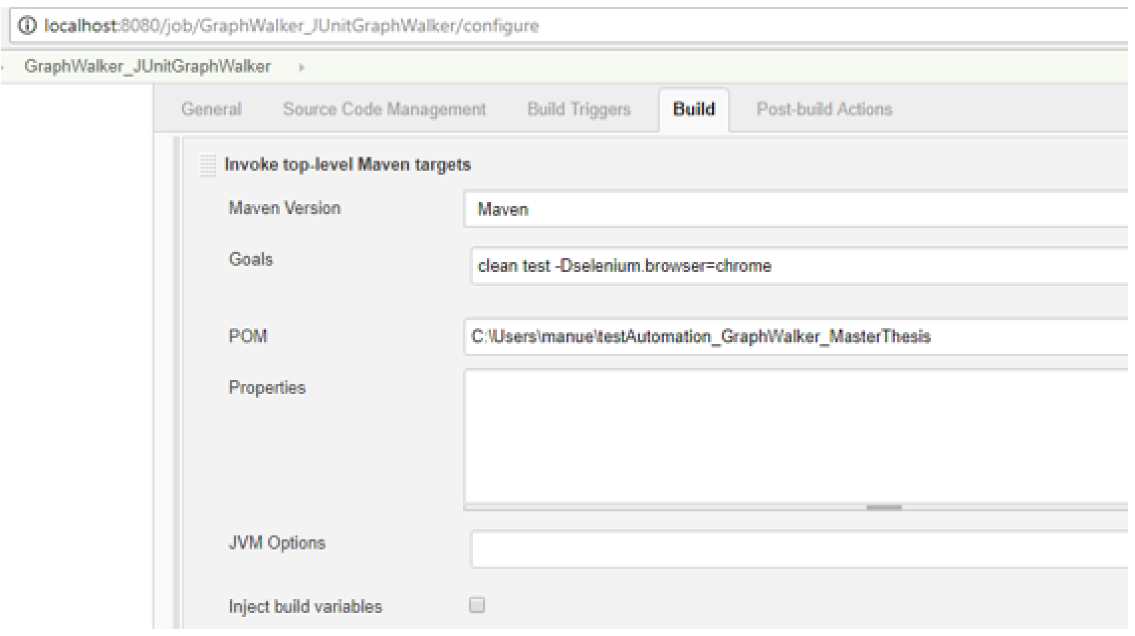


Figura 19 - Configurações do projeto no *Jenkins*

5.1.2 Cenário prático utilizando uma aplicação móvel

Para apresentar este cenário foi utilizada a aplicação móvel de email *Mail.Ru* com sistema operativo *Android*. Para facilitar a explicação e o detalhe, a descrição foi dividida em cinco pontos. Criação do projeto com o *GraphWalker*, verificação da correta criação do modelo, criação da classe de execução de testes, criação dos testes automatizados. Os passos que foram realizados de igual forma aos do caso prático utilizando o *website*, foram apenas mencionados, para consulta dos seus detalhes.

5.1.2.1 Criação do projeto

O projeto criado foi do tipo *Maven* da mesma forma que foi criado para a componente prática utilizando o *website*. É possível consultar os detalhes no ponto 5.1.1.1.

No trecho de código 10 é possível visualizar a classe de testes LoginTest.java a implementar a classe interface LoginSearchApp.java, gerada automaticamente a partir do modelo, e já com a implementação dos métodos gerados a partir da classe interface.

Trecho de código 10 – Classe LoginTest.java após implementação dos métodos

```
@GraphWalker
public class LoginTest extends  ExecutionContext implements LoginSearchApp {
  @Override
  public void e_OpenApp() {
  }

  @Override
  public void e_invalidCredentials() {
  }

  @Override
  public void v_SearchResult() {
  }

  @Override
  public void v_LoginPrompted() {
  }

  @Override
  public void e_Exit() {
  }

  @Override
  public void v_AppOpened() {
  }

  @Override
  public void e_selectEmailProvider() {
  }
  @Override
  public void v_EmailOpened() {
  }

  @Override
  public void v_AppNotStarted() {
  }

  @Override
  public void e_searchEmail() {
  }

  @Override
  public void v_LoginMade() {
  }

  @Override
  public void e_validCredentials() {
}

  @Override
  public void e_CloseApp() {
}

  @Override
  public void e_logout() {

  }

  @Override
  public void e_Init() {
  }

  @Override
  public void e_ClickToOpen() {
}

  @Override
   public void e_giveAccessApp() {
  }

@Override
  public void v_appAccessGiven() {
 } }
```

5.1.2.2 Verificação da criação correta do modelo

A verificação da criação correta do modelo é realizada da mesma forma que é para o caso prático utilizando o website. Os passos em comum podem ser consultados no ponto 5.1.1.2.

O modelo LoginSearchApp.graphml da figura 20 corresponde à representação de três funcionalidades da aplicação móvel *Android* Mail.Ru:

1. Efetuar login no sapo mail utilizando a app Mail.Ru;
2. Após login -> Efetuar uma pesquisa por um email;
3. Abrir o email pesquisado;

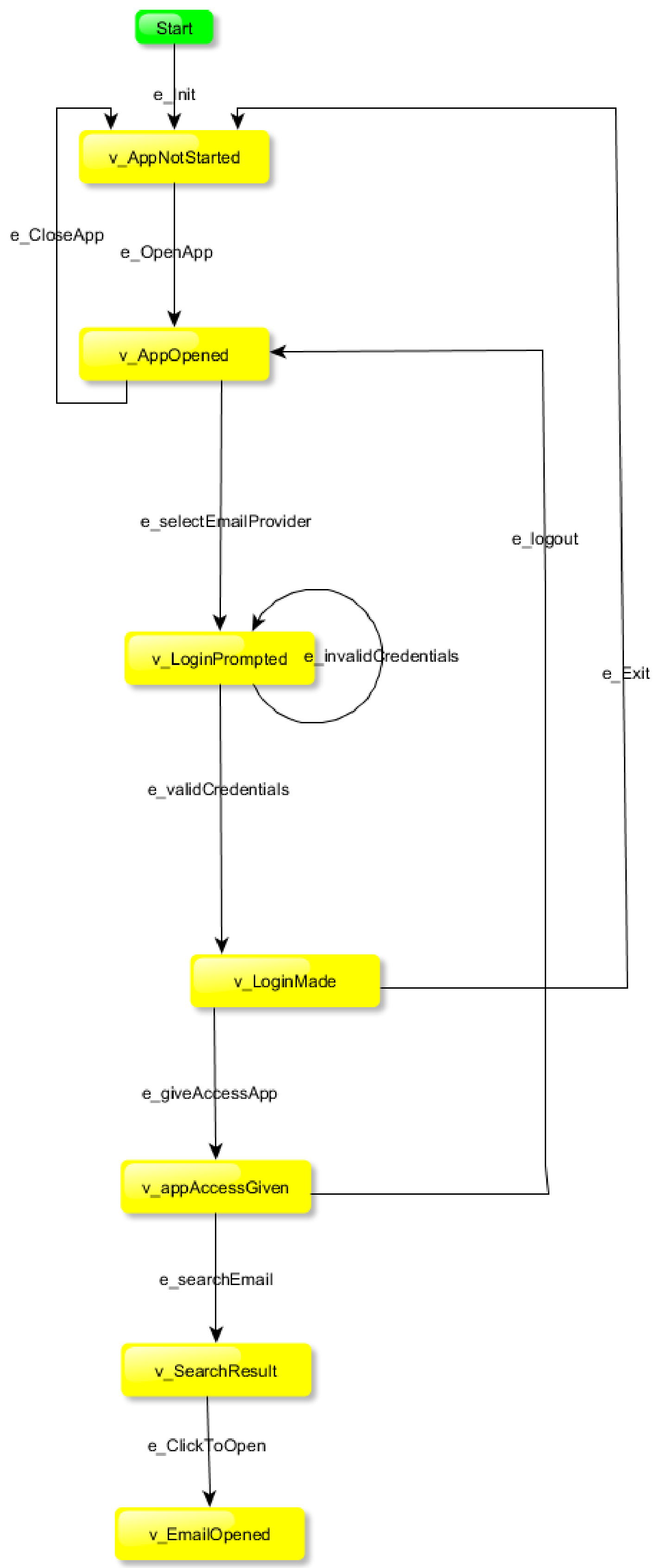


Figura 20 - Modelo representativo das funcionalidades de login e pesquisa na aplicação

5.1.2.3 Criação da classe de execução de testes

O procedimento utilizado é exatamente igual ao que é realizado para o caso prático utilizando um website. Por esse motivo não serão novamente descritos os passos e podem ser consultados no ponto 5.1.1.3.

5.1.2.4 Implementação dos testes automatizados

O passo 4 corresponde à implementação dos testes automatizados para permitir a execução automática dos testes durante a geração das sequências de testes. É possível visualizar a sua implementação no anexo B.

5.1.2.5 Necessidades adicionais para implementação dos testes automatizados utilizando aplicação

Para uma explicação mais simplificada sobre as necessidades adicionais na implementação dos testes automatizados utilizando a aplicação, os detalhes foram divididos em três principais pontos:

1. Criação e execução dos testes automatizados na aplicação de email Mail.ru;
2. Execução dos testes automatizados na aplicação utilizando emulador criado com *Android Studio*;
3. Execução dos testes automatizados na aplicação utilizando dispositivo físico conectado via USB;

1. Criação e execução dos testes automatizados na aplicação Mail.ru *Android*

Para que fossem possíveis a criação e a execução de testes automatizados na aplicação, foram necessárias as instalações do servidor *Appium* para iniciar e terminar as execuções do servidor. E do *Appium Desktop* para iniciar e terminar sessões de inspeção dos elementos da aplicação móvel.

Para Instalar o servidor *Appium,* foi necessário instalar o Node.JS. A instalação do Node.JS foi realizada tendo em conta a instalação da última versão através da execução do comando: ***npm install -g appium***. Sempre que foi iniciada a execução do servidor do *Appium* foram indicados o *host* e a porta.

A utilização do inspetor do *Appium Desktop* foi realizada tendo em conta duas abordagens. Uma abordagem indicando as capacidades e a localização do APK como é possível visualizar a configuração na figura 21.

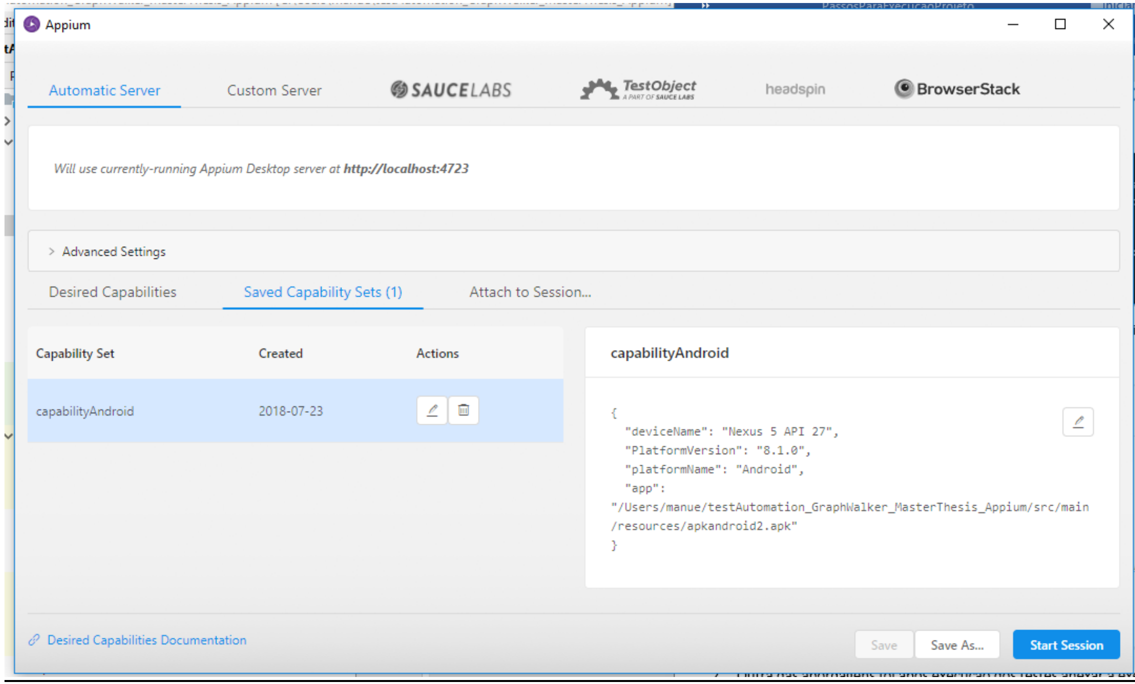


Figura 21 - Configuração para início de uma nova sessão utilizando inspetor do *Appium*

A outra abordagem, após execução dos testes anexando a execução corrente ao Inspetor, como é possível visualizar a configuração na figura 22.

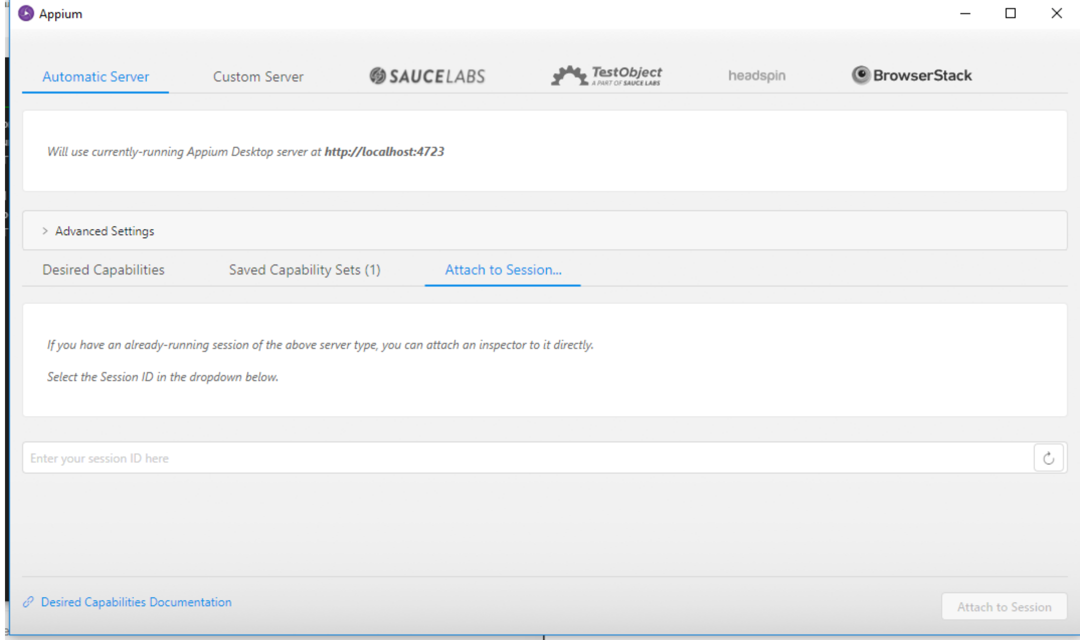


Figura 22 - Configuração para início de uma nova sessão do inspetor do *Appium Desktop* utilizando a sessão corrente em execução

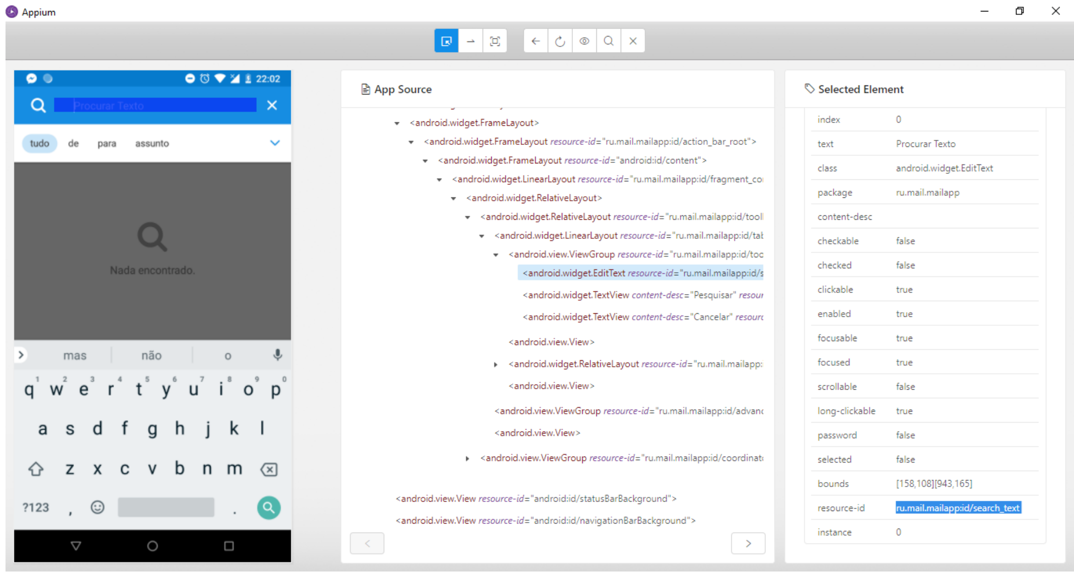


Figura 23 - Inspeção de elementos da aplicação utilizando inspetor do *Appium Desktop*

A utilização de um emulador ou de um dispositivo físico conectado via USB foram também necessárias. Depois da instalação do servidor *Appium* e do *Appium Desktop* foram preparados o emulador através de criação no *Android Studio* e/ou conectado um dispositivo físico via ADB. Foi necessário recorrer a ferramentas específicas que possibilitassem a identificação dos elementos a serem clicados na aplicação durante a execução dos testes automatizados. Para a identificação dos elementos foram utilizadas duas ferramentas o *UiAutomatorViewer* que vem embutido com o *Android Studio* e o inspetor do *Appium* já mencionado anteriormente como fazendo parte do *Appium Deskto*p.

Foi então possível a obtenção dos elementos em cada ecrã da aplicação utilizando diferentes tipos de identificadores, nomeadamente *xpath*, *className*, id, *accessibilityId* entres outros.

2. Execução dos testes automatizados na aplicação Mail.ru utilizando emulador criado com *Android Studio*

Para utilização do emulador foi necessário instalar o *Android Studio* e o *Android SDK* para a criação dos emuladores. O emulador utilizado correspondeu às características de um telemóvel Google *Nexus* 5X com sistema operativo *Android* com a versão *Oreo*.

3. Execução dos testes automatizados na aplicação Mail.ru utilizando dispositivo físico conectado via USB

Foi utilizado um dispositivo físico Google *Nexus* 5X Android versão *Oreo* para auxiliar a criação e execução dos testes automatizados. Após habilitação das ferramentas de programador no telemóvel foi utilizado o IP para realização da conexão ao servidor *Appium* via ABD.

CAPÍTULO 6

ANÁLISE RESULTADOS

No capítulo 6 foram analisados os resultados do caso prático. Os casos utilizando o website e aplicação móvel foram analisados no ponto 6.1 e os casos utilizando os serviços web no ponto 6.2. Foi também realizada uma comparação de funcionamento entre a metodologia MBT e a metodologia BDD.

6.1 Análise de resultados caso prático

Foram necessários vários passos para configurar um projeto Java *Maven* com o *GraphWalker* e uma curva de aprendizagem moderada para a correta utilização da ferramenta. Foi despendido bastante tempo em tentativas até se chegar à melhor abordagem. A documentação no site oficial do *GraphWalker* contém alguns exemplos com os passos de implementação e execução para diferentes tipos de projetos, no entanto, não existe muita informação detalhada sobre a ferramenta. Nem sobre as possíveis formas de a utilizar nem sobre a possibilidade de integração da mesma com o *Jenkins*. A solução pretendida para o projeto alvo do estudo consistiu na execução dos testes utilizando diferentes algoritmos de geração de sequências de testes, condições de paragem. E execução dos diferentes algoritmos a partir do *Jenkins*. Foi necessário alterar a configuração por defeito que é apresentada como solução na página oficial do *GraphWalker*, nos projetos exemplo para contemplar a solução pretendida. Dado não existir documentação suficiente, sobre uma forma adequada de realizar a configuração referida anteriormente, surgiram alguns problemas durante a configuração do projeto. A execução na máquina local funcionava com os vários algoritmos de geração de caminhos e condições de paragem, mas após configuração do job no *Jenkins* a classe era ignorada. Foi necessário então recorrer à comunidade de suporte à utilização do *GraphWalker*. Após a apresentação da questão no grupo de suporte, com todo o detalhe necessário, com a configuração realizada no *Jenkins* e um exemplo da configuração do projeto Java *Maven* com *GraphWalker* disponibilizada no *Github*, a questão foi respondida em poucos dias. Sendo apresentada e disponibilizada também no *Github* uma solução por parte do responsável da ferramenta com as correções necessárias para o projeto funcionar da forma pretendida.

Para que o caso prático fizesse sentido e fosse o mais aproximado possível com um caso em ambiente empresarial real, foi utilizado o mesmo fluxo e abordagem da empresa onde

atualmente a atora desta tese trabalha e onde os testes são uma parte intrínseca do desenvolvimento de software. Apesar dos sistemas utilizados para o caso prático já existirem, tendo em conta que, na metodologia MBT os modelos são desenhados antes de o sistema existir, no âmbito dos casos apresentados também os modelos foram desenhados sem conhecimento do seu comportamento apenas da existência das funcionalidades.

Após criação do projeto e integração com o *GraphWalker*, durante a criação e execução dos testes automatizados para o comportamento descrito nos respetivos modelos, foi possível perceber tanto para o modelo do website como para o modelo da aplicação que cada um deles teria de ser adaptado ao comportamento atualmente existente em cada um dos sistemas. Utilizando diferentes algoritmos de geração de sequências de testes a percorrer pelo modelo e condições de paragem em conjunto com a implementação dos testes automatizados, foi possível verificar que, caso o modelo não esteja completamente de acordo com o comportamento atual dos sistemas a execução dos testes ao passar por caminhos desconhecidos vai falhar.

No exemplo da figura 24 o passo de realização do *logout* foi descrito de forma incorreta. Em vez de após *logout* o utilizador ser redirecionado para a página onde são pedidas as credenciais, deveria ter sido enviado para a página principal após ter efetuado pesquisa pelo website. Durante a execução dos testes automatizados, o sistema ao tentar encontrar o elemento *web* do passo seguinte, que se encontra dentro da página de login falha, pois não se encontra na página correta. É possível visualizar um exemplo da falha dessa execução no anexo C.

Existe vantagem de criar o modelo para representar o comportamento dos sistemas. Mesmo não existindo conhecimento total do sistema, para além de melhorar a comunicação de uma equipa, proporciona um entendimento comum sobre o sistema. É possível também sintetizar o comportamento que é realmente importante ser testado e automatizado em cada funcionalidade. É possível antecipar e efetuar um planeamento mais assertivo relativamente aos testes automatizados que vão necessitar de implementação. Devido ao modelo ser definido numa fase tão inicial do processo é normal que o mesmo sofra alterações durante o desenvolvimento do sistema ou ate mesmo no final. A situação descrita aconteceu com os casos práticos apresentados. Nesse caso, todo o trabalho teve de ser alterado, desde o modelo à implementação dos testes automatizados. Para se manterem de acordo com o novo modelo. A forma de alteração do modelo revelou-se um pouco trabalhosa.

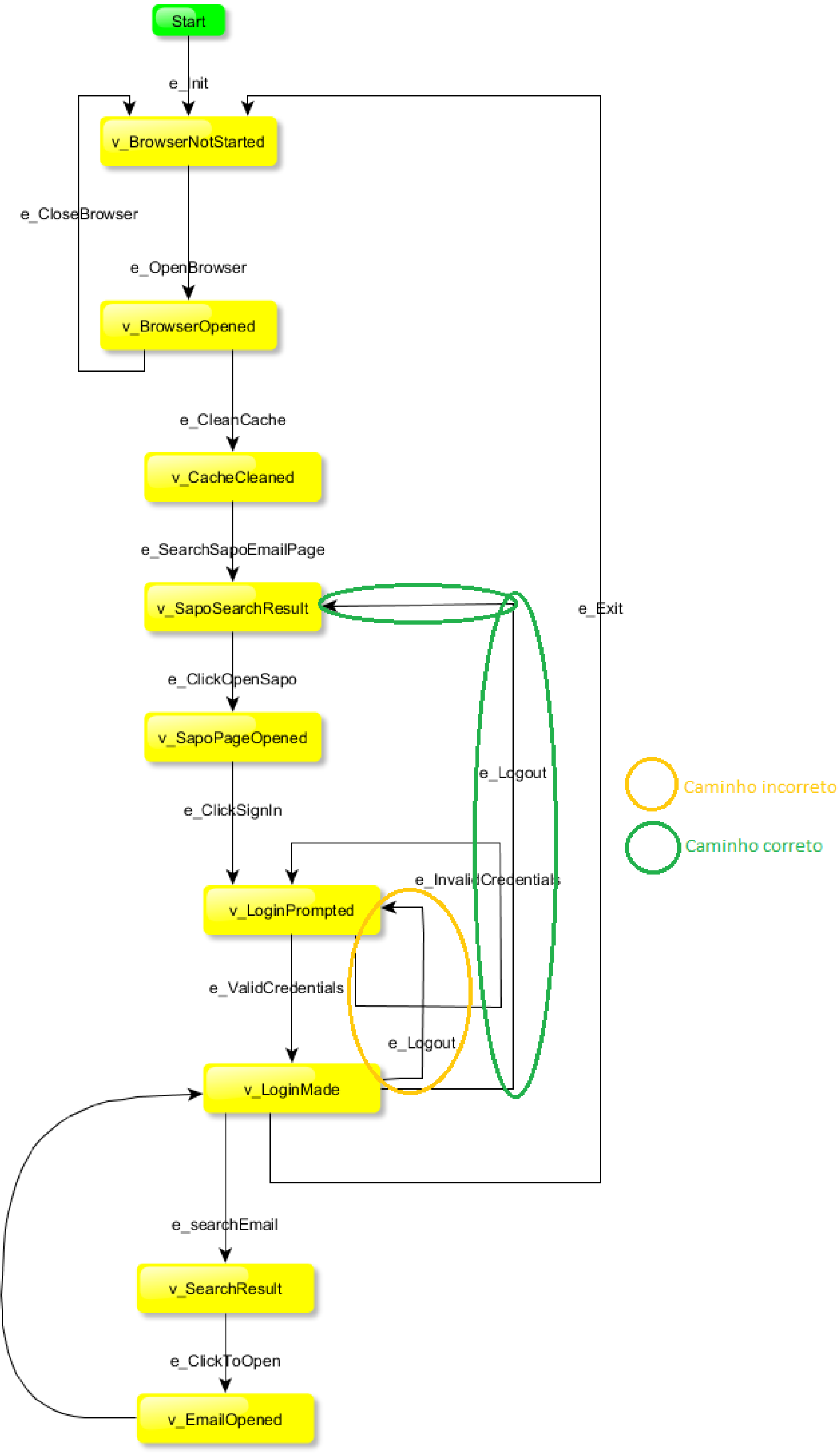


Figura 24 - Exemplo modelo construído de forma incorreta

A utilização dos modelos para representar comportamento funcional de aplicações em conjunto com a utilização do *GraphWalker* e com a criação de testes automatizados, tendo em conta a situação descrita, comprova que, qualquer alteração ao comportamento dos sistemas que esteja a ser representada nos modelos de forma incorreta vai ser detetada. Quer tenha sido introduzida por erros no desenho dos modelos, quer porque o sistema foi alterado e o modelo permaneceu desatualizado ou por defeitos que foram introduzidos no sistema. A mesma situação ocorreu durante a execução dos testes sobre modelos construídos de forma incorreta para a aplicação móvel.

Para cada um dos casos práticos utilizando o website e a aplicação, foram utilizados de forma combinada, diferentes algoritmos de geração de sequências de testes e condições de paragem. Esta combinação permitiu gerar de forma fácil e rápida diferentes sequências de testes através de execução aleatória pelo modelo. E também ao serem combinados proporcionaram uma maior cobertura de casos de teste. O facto de existir uma combinação na utilização dos algoritmos de geração de sequências de testes com condições de paragem, em conjunto com a implementação de testes automatizados, permitiu otimizar o tempo de identificação e de execução de casos. Sendo o tempo gasto com o desenho do modelo reutilizado para a geração das sequências de testes e para a implementação dos testes automatizados em cada método gerado também a partir do modelo. Sendo o tempo gasto bastante inferior à realização de uma especificação de testes e de uma execução de testes de forma manual ou automatizada seguindo a forma convencional.

Para o caso prático foram analisados com maior detalhe os algoritmos de geração de sequências de testes combinados com as seguintes condições de paragem:

- *Random(vertex_coverage(100));*
- *Weighted_random(vertex_coverage(100));*
- *Random(vertex_coverage(50));*
- *Weighted_random(vertex_coverage(50));*
- *Random(vertex_coverage(v_SomeVertex));*

Cada uma das escolhas para combinação de geradores com condições de paragem permitiram que fossem geradas diferentes sequências de teste de forma aleatória com sucesso. O facto de a geração das sequências de teste ser realizada de forma aleatória apresentou algumas desvantagens. Nomeadamente o facto de tornar possível que sequências de testes geradas, com uma maior potencialidade para encontrar defeitos no software, nunca sejam executadas.

Outra desvantagem está relacionada com a utilização de geradores de sequências de testes com condições de paragem com um valor inferior a 100 % de cobertura a percorrer pelos vértices e pelas bordas do modelo. Nos casos em que foi utilizado um modelo com erros durante a execução dos testes, devido à natureza aleatória dos algoritmos, fez com que as bordas ou os vértices com erros de desenho não fossem visitados e por esse motivo não fosse detetada a existência de erros.

No caso mencionado é apresentado um exemplo da situação. É utilizado o exemplo da figura 24, que contém o erro na borda do *logout*. É utilizado o algoritmo de geração de caminhos aleatórios com a condição de paragem "*edge_coverage (50%)*", o resultado da execução é sucesso pois, dada a condição de paragem corresponder ao percorrer de apenas 50% das bordas e serem sempre realizadas de forma aleatória que a borda que contém o erro não é visitada. É possível visualizar o exemplo da execução no anexo D.

6.2 Análise de resultados do caso prático utilizando serviços web

Os serviços web foram implementados seguindo o estilo de arquitetura REST (*Representational State Transfer*) de acordo com regras específicas. Uma das principais regras do estilo de arquitetura REST (*Representational State Transfer*), que é relevante para a apresentação da conclusão deste resultado prático, é o facto de permitir apenas a implementação de serviços cuja comunicação seja sempre realizada sem serem guardados estados, ou seja, cada pedido de cliente deve conter toda a informação necessária para ser entendido pelo serviço.

Os estados são condições que acontecem num determinado momento. Um serviço web que guarde estados, *stateful*, é um serviço que depende desses estados durante um determinado momento no tempo e que utiliza esses estados para alterar as respostas devolvidas de acordo com cada pedido realizado. Por exemplo, um serviço web que guarda estados ao fazer um simples pedido GET depende totalmente do estado registado, que pode ser oriundo de uma validação que esteja a ser feita, como exemplo, num método de outra classe, cuja resposta poderá ser diferente a cada pedido que é realizado. Sem o conhecimento daquele estado o pedido poderá nem ser retornado corretamente. O funcionamento dos serviços *stateful* possui imensos problemas. Um dos principais problemas é o de permitir que sejam geradas sessões e transações incompletas fazendo com que a sua utilização não seja escalável [25].

Principalmente por este motivo e por outros a utilização de serviços REST consequentemente *stateless*, ganhou força no desenvolvimento de software. Sendo esse também um dos motivos para a utilização dos serviços REST na apresentação deste caso prático. No entanto, e tendo em conta as características descritas pela notação utilizada pelo *GraphWalker,* previamente no capítulo 4 no ponto 4.3, os modelos necessitam de estados para ser possibilitarem a descrição das funcionalidades. As funcionalidades desenvolvidas utilizando os serviços web seguiram o estilo de arquitetura REST e consequentemente *stateless*, não guardando estados. Por esse motivo, não foi possível a sua descrição e por consequência nem a sua utilização para a geração, automação e execução de testes.

Foram executados os passos necessários para se chegar a esta conclusão, incluindo o desenvolvimento dos serviços web para as funcionalidades de login e pesquisa e a utilização de um servidor para a realização do seu *deploy.* Após tentativa falhada de descrição do funcionamento dos serviços web no modelo, por opção e dada a sua não relevância foram omitidos os passos de implementação da descrição deste caso prático.

6.3 Comparação *Model-Based Testing* com metodologia *Behavior-Driven Development*

Na tabela 42 é apresentada uma comparação que exemplifica as diferenças entre as metodologias *Model-Based Testing* e *Behavior-Driven Development*. Esta descrição encontra-se divida em três principais fases de implementação de testes. A fase de especificação de casos de teste, a fase de implementação de testes automatizados e a fase de execução dos testes. Os passos relatados na metodologia *Behavior-Driven Development* dizem respeito ao conhecimento e experiência vivenciada em âmbito de trabalho real.

Tabela 42 - Comparação de metodologias

Metodologias / Fases	***Behavior-Driven Development***	***Model-Based Testing***
Especificação de casos de teste	• São identificados os cenários a serem automatizados, a partir da análise do conteúdo de uma ou mais *user stories* e dos seus critérios de aceitação; • São escritos os cenários BDD utilizando a sintaxe *Gherkin*. Por exemplo, utilizando a *framework Cucumber,* fazendo uso das	• É realizada a construção do modelo para descrever as funcionalidades do sistema ainda sem o mesmo existir; • É integrado o modelo com uma ferramenta MBT, como exemplo, no *GraphWalker*; • Após integração do modelo com o *GraphWalker* é gerada a classe interface que implementa os

	keywords: *GIVEN*, *WHEN*, *AND* e *THEN* para descrever os cenários;	métodos descritos no modelo, numa classe de testes; • Após término da implementação do sistema, se necessário deverá ser realizada uma adaptação do modelo ao funcionamento atual do sistema; • O *GraphWalker* utilizando um algoritmo, percorre o modelo e gera sequências de teste de forma aleatória até atingir a condição de paragem definida;
Implementação de testes automatizados	• São automatizados os cenários BDD, por exemplo, utilizando o *Selenium* ou o *Appium*, dependendo das características do produto que contém a funcionalidade alvo de testes. • É automatizado cada passo para permitir a sua execução de forma automática. • A primeira execução irá falhar, visto que, a funcionalidade a ser testada ainda não se encontra desenvolvida; • Após término do desenvolvimento da funcionalidade, são executados os testes, previamente desenvolvidos, para verificar que funcionalidade se encontra completa; • São integrados os testes automatizados numa ferramenta de entrega contínua;	• É realizada a implementação dos testes automatizados em cada método, implementado a partir da classe interface, na classe de testes; • São integrados os testes automatizados numa ferramenta de entrega contínua;
Execução de testes	• É realizada a execução dos testes automatizados de forma integrada após *build* e *deploy* em ferramenta de entrega contínua;	• É realizada a execução dos testes automatizados de forma integrada após *build* e *deploy* em ferramenta de entrega contínua;

6.3.1 Principais vantagens e desvantagens da metodologia MBT em relação à metodologia BDD

Relativamente à fase de especificação de testes existe vantagem em utilizar a metodologia MBT em vez da metodologia BDD. Existe um ganho de tempo com a utilização do modelo para descrever as funcionalidades do sistema. Os algoritmos utilizados para a geração das sequências de testes, utilizam o modelo. Desta forma, o tempo utilizado com o desenho do modelo é reaproveitado pela geração das sequências de testes de forma automática. Também os métodos gerados a partir do modelo são reaproveitados. Na fase de implementação dos

testes automatizados é implementado o código dentro de cada método gerado, para permitir a execução automática das sequências de testes. Contrariamente ao que é realizado utilizando a metodologia BDD, cuja definição inicial de casos de teste não é gerada nem executada de forma automática, mas sim manual. Por outro lado, a forma de escrita simplificada utilizada pela metodologia BDD, para definição dos casos de teste, apresenta-se como uma vantagem sobre a forma utilizada pela metodologia MBT. O desenho do modelo requer *skills* de modelação que são difíceis de adquirir, então é mais fácil escrever casos de teste utilizando o BDD do que desenhar um modelo utilizando o MBT.

Outra vantagem da metodologia BDD sobre a metodologia MBT que diz respeito a uma maior rapidez na adição de novos casos de teste. Com a metodologia BDD é possível adicionar de forma bastante mais rápidos novos casos de teste a um projeto sem que exista impacto nos testes previamente implementados. No caso da MBT, caso exista necessidade de adicionar novos casos de teste, pode ser implementado um novo modelo, que necessita mais tempo para o seu desenho e é mais complexo. Ou alterar um modelo já existente, que também requer mais tempo e que faz com que, as novas alterações afetem toda a estrutura de testes já implementada para a funcionalidade a ser alterada.

A geração da sequência de testes de forma aleatória representa uma vantagem da metodologia MBT em relação à metodologia BDD. O facto de serem geradas de sequências de testes de forma aleatória, simulam a execução de testes exploratórios. De acordo com a referência [47] os testes exploratórios encontram problemas mais críticos no software do que a forma tradicional. Então, ao utilizar o *Model-Based Testing* existe uma maior possibilidade de serem encontrados problemas mais críticos no *software.*

CAPÍTULO 7

O capítulo 7 faz uma revisão e um resumo relativamente à investigação realizada com esta tese de mestrado, para responder essencialmente as duas principais questões que motivaram a sua realização.

7.1 CONCLUSÃO

O aumento substancial de empresas de desenvolvimento de software, motivam a investigação de novas metodologias que apresentem resultados eficazes na realização de testes dentro de um processo de desenvolvimento de software. O MBT é uma técnica avançada de realização de testes. É uma técnica recente e pouco conhecida. Por esse motivo, é também pouco utilizada.

Com a realização deste estudo foi possível conhecer as ferramentas que suportam a metodologia e quais as suas potencialidades, aplicar a técnica do MBT a cenários de trabalho real e comparar o seu funcionamento com outra metodologia mais comumente utilizada.

Foi possível verificar que uma grande parte do que é possível alcançar, através da utilização da metodologia, depende quase que inteiramente das capacidades das ferramentas que a suportam. Sendo de grande importância, quando se utiliza a metodologia a escolha de uma ferramenta que a suporta, adaptada às especificidades e tecnologias dos produtos desenvolvidos a serem testados.

Para tentar obter alguma informação sobre a viabilidade da utilização da metodologia MBT, foram definidas duas principais questões a serem respondidas com realização desta tese:

Questão 1: Como pode ser aplicada a metodologia *Model-Based Testing* utilizando cenários de trabalho real?

Questão 2: Quais as vantagens e as desvantagens da metodologia MBT em relação à metodologia BDD?

Para responder à primeira questão foram apresentados três casos práticos de forma detalhada, onde foi aplicada a metodologia MBT, utilizando como ferramenta de suporte o *GraphWalker*, e cenários de trabalho reais. Um caso prático foi utilizando um *website*, outro uma aplicação móvel e outro utilizando serviços *web*. Para os casos práticos, utilizando o *website* e a aplicação móvel, foi possível implementar todo um processo de testes seguindo a metodologia MBT. Desde a especificação de testes, passando pela implementação dos testes automatizados até à sua execução e integração numa ferramenta de entrega contínua. Foi também possível

comprovar que a existência de incoerências entre a descrição do modelo e o funcionamento SUT durante a execução dos testes são encontradas.

Relativamente ao caso prático utilizando serviços web foi encontrada uma limitação na implementação. A tecnologia utilizada para seu desenvolvimento dos serviços *web* foi o REST. Dada a sua natureza *stateless*, não guarda, nem utiliza estados e por esse motivo não foi possível desenhar o modelo. O *GraphWalker* para criação de modelos utiliza a notação de máquinas de estados finitos e requer estados para o seu desenho. Sendo o modelo a base para a utilização do MBT, não sendo possível desenhar o modelo não foi possível implementar um processo de testes para serviços web implementados utilizando o MBT. Comprovando, desta forma, o que foi mencionado anteriormente nesta conclusão. O sucesso da utilização da metodologia MBT depende das potencialidades da ferramenta selecionada para a suportar e da sua capacidade de adaptação às especificidades e tecnologias do produto desenvolvido a ser testado.

Para responder à segunda questão foi realizada uma comparação em três diferentes fases, especificação de casos de testes, implementação de testes automatizados e execução dos testes utilizando a metodologia MBT em relação à metodologia BDD. A partir dessa comparação foram obtidas conclusões importantes. Utilizando a metodologia MBT existe uma importante vantagem sobre a metodologia BDD. Existe um ganho de tempo com utilização do modelo para descrever as funcionalidades de um sistema. Os algoritmos utilizados, geram as sequências de testes e métodos de forma automática a partir do modelo. Sendo o código dos testes automatizados implementado em cada método gerado, para permitir a sua execução de forma automática durante a geração. Contrariamente ao realizado utilizando a metodologia BDD que inicialmente pressupõe uma geração de casos de teste de forma manual. Por outro lado, a metodologia BDD permite uma adição mais fácil e mais rápida de novos casos de teste a um projeto, surgindo aqui como uma vantagem sobre a metodologia MBT, que na fase de desenho do modelo requer bastante mais tempo. Também a natureza aleatória de geração de sequências de testes representa uma vantagem da metodologia MBT sobre a metodologia BDD, simulando a execução de testes exploratórios e aumentando a probabilidade de serem encontrados problemas mais críticos utilizando a metodologia MBT.

7.2 TRABALHO FUTURO

Como trabalho futuro seria interessante o desenvolvimento de uma ferramenta para suportar a metodologia MBT que tivesse a capacidade de se adaptar a todo o tipo de notações de modelação, aplicações, tecnologias e linguagens de programação. Cujo modelo fosse gerado de forma automática a partir de um protótipo do SUT. Bem como, os restantes artefactos, também eles gerados de forma automática, e de acordo com a linguagem de programação utilizada para desenvolvimento do SUT. A implementação desta ferramenta intensificaria as capacidades de da metodologia MBT e aumentaria a sua adoção durante a implementação de processos de desenvolvimento de software e de testes. Resolveria limitações encontradas durante a implementação da metodologia MBT em relação às ferramentas que a suportam.

Referências

[1] ISTQB®, 2015. Foundation Level Certified Model-Based Tester Syllabus

[2] Anne Kramer, A. Bruno Legeard, B. 2016. Model-Based Testing Essentials Guide to the ISTQB® Certified Model-Based Tester Foundation Level

[3] Mark Utting, M. Bruno Legeard, B. 2007. Practical Model-Based Testing Tools approach. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA

[4] [Online] https://www.guru99.com. Disponível em: https://www.guru99.com/model-based-testing-tutorial.html [Último acesso 30 de Março de 2018].

[5] Paul C. Jorgensen, P. 2017. The Craft of Model Based Testing. CRC Press Taylor & Francis Group 6000 Broken Sound Parkway NW, Suite 300 Boca Raton, FL 33487-2742

[6] Robert Shapiro, R. Stephen A White, S. Conrad Bock, C. Nathaniel Palmer, N. Michael zur Muehlen, M. Marco Brambilla, M. Denis Gagné. D. 2011. BPMN 2.0 Handbook Second Edition: Methods, Concepts, Case Studies and Standards in Business Process Modeling Notation (BPMN). Future Strategies, Incorporated

[7] Joiner Associates Staff, J. 1995. Flowcharts Plain & Simple: Learning & Application Guide. Joiner Associates Inc. USA

[8] John R. Metzner, J. Bruce H. Barnes, B. 1977. Decision table languages and systems. Association for Computing Machinery Inc. USA

[9] Ferdinand Wagner, F. Ruedi Schmuki, R. Thomas Wagner, T. Peter Wolstenholme, P. 2006. Modeling with Finite State Machines, A Practical Approach. CRC Press Taylor & Francis Group 6000 Broken Sound Parkway NW, Suite 300 Boca Raton, FL 33487-2742

[10] F. Wagner, F. 2006. What's All This State Machine Stuff?

[11] James L. Peterson, J. 1981. Petri net theory and the modeling systems. Prentice-Hall, Inc. Englewood Cliffs, USA

[12] Kurapati Venkatesh, K. MengChu Zhou, M. 1999. Modeling, simulation, and control of flexible manufacturing systems A Petri Net Approach. World Scientific Publishing Co Pte Ltd, USA

[13] Paulo Sousa, P. Introdução à linguagem UML. Instituto Superior de Engenharia do Porto

[14] James Rumbaugh, J. Ivar Jacobson, I. Grady Booch, G. 2004. The Unified Modeling Language Reference Manual Second Edition. Association for Computing Machinery Inc. USA

[15] Kirill Fakhroutdinov, K. (2009). The Unified Modeling Language. [Online] https://www.uml-diagrams.org. Disponível em: https://www.uml-diagrams.org/examples/use-case-example-online-shopping.png [Último acesso: 28 de Janeiro de 2018].

[16] Donald Bell, D. UML basics: An introduction to the Unified Modeling Language

[17] Adriana Silva. A. (2018). Análise Orientada a Objetos. [Online] https://profadrisilva.files.wordpress.com. Disponível em: https://profadrisilva.files.wordpress.com/2018/08/eb7wb.png [Último acesso: 28 de janeiro de 2018].

[18] Kelly Sganderla. K. (2013). Respondendo dúvidas em BPMN: Desenhar processo na vertical ou horizontal? [Online] http://blog.iprocess.com.br/2013/05/respondendo-duvidas-em-bpmn-desenhar-processo-na-vertical-ou-horizontal. Disponível em: http://blog.iprocess.com.br/wp-content/uploads/2013/05/BPMN-pools-e-lanes-diagrama-horizontal1.png [Último acesso: 28 de janeiro de 2018].

[19] Contributing Writer, C. (2018). The Disadvantages of UML. [Online] https://www.techwalla.com. Disponível em: https://www.techwalla.com/articles/the-disadvantages-of-uml [Último acesso: 11 de março de 2018].

[20] BPMN Diagram Symbols & Notation. [Online] https://www.lucidchart.com. Disponível em: https://www.lucidchart.com/pages/bpmn-symbols-explained [Último acesso: 11 de março de 2018].

[21] Georgina, G (2014). Test Automation: How Flowcharts Reduce Cost And Improve Efficiency, Making Complete Testing A Reality. [Online] https://huddle.eurostarsoftwaretesting.com. Disponível em: https://huddle.eurostarsoftwaretesting.com/test-automation-how-flowcharts-reduce-cost-and-improve-efficiency-making-complete-testing-a-reality [Último acesso: 24 de Abril de 2018].

[22] Philip Howard, P. (2016). Test Case Generation Market Report. Paper by Bloor. 20-22 Wenlock Road London United Kingdom

[23] Ulf Eriksson, U. (2012). A guide to Using Decision Tables. [Online] https://reqtest.com/requirements-blog. Disponível em: https://reqtest.com/requirements-blog/a-guide-to-using-decision-tables [Último acesso: 5 de maio de 2018]

[24] David Coghlan, D. Teresa Brannick, T. 2014. Doing Action Research in Your Own Organization. Sage Publications Ltd

[25] Kristopher Sandoval, K. (2017). Defining Stateful vs Stateless Web Services. [Online] https://nordicapis.com. Disponível em: https://nordicapis.com/defining-stateful-vs-stateless-web-services [Último acesso: 5 de Agosto de 2018].

[26] GraphWalker org, G. (2018). GraphWalker. [Online] http://graphwalker.github.io. Disponível em: http://graphwalker.github.io

[27] Zoltán Micskei, Z. (2018). Model-Based-Testing (MBT). [Online] http://mit.bme.hu. Disponível em: http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html [Último acesso: 5 de agosto de 2018].

[28] JSXM. (2017). [Online] http://www.jsxm.org. Disponível em: http://www.jsxm.org/jsxm-maven-plugin/jsxmcore.html [Último acesso: 5 de agosto de 2018].

[29] Dimitris Dranidis, D. Konstantinos Bratanis, K. and Florentin Ipate, F. (2012). JSXM: A Tool for Automated Test Generation. [Online] http://www.ifsoft.ro. Disponível em: http://www.ifsoft.ro/~florentin.ipate/publications/SEFM2012_CR.pdf [Último acesso: 5 de Agosto de 2018].

[30] Alex Belifante, A. (2010). JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. [Online] https://link.springer.com. Disponível em: https://link.springer.com/content/pdf/10.1007%2F978-3-642-12002-2_21.pdf [Último acesso: 5 de Agosto de 2018].

[31] (2015). [Online] http://cs.boisestate.edu. Disponível em: http://cs.boisestate.edu/~dxu/research/MBT.html [Último acesso: 5 de agosto de 2018].

[32] Cyrille Artho, C. Armin Biere, A. (2013). Modbat: A Model-based Tester. [Online] http://fmv.jku.at. Disponível em: http://fmv.jku.at/modbat [Último acesso: 25 de agosto de 2018].

[33] Cyrille Artho, C. Armin Biere, A. (2013). Modbat: A Model-based Tester. [Online] http://fmv.jku.at. Disponível em: http://fmv.jku.at/modbat/README [Último acesso: 25 de agosto de 2018].

[34] Cyrille Valentin Artho, C Armin Biere, A. Masami Hagiya, M. Eric Platon, E. Martina Seidl, M. Yoshinori Tanabe, Y. Mitsuharu Yamamoto, M. (2013) Modbat: A Model-based API Tester for Event-driven Systems. [Online] https://link.springer.com. Disponível em: https://link.springer.com/chapter/10.1007/978-3-319-03077-7_8 [Último acesso: 25 de Agosto de 2018].

[35] Austrian Institute of Technology, AIT (2018). About. [Online] https://momut.org. Disponível em: https://momut.org [Último acesso: 25 de agosto de 2018].

[36] Antti Kervinen, A. (2018). FMBT. [Online] https://01.org. Disponível em: https://01.org/fmbt [Último acesso: 20 de agosto de 2018].

[37] Bernhard Aichernig, B. Harald Brandl, H. Elisabeth Jobstl, E. Willibald Krenn, W. Rupert Schlick, R. Stefan Tiran, S. (2016). MoMuT::UML Model-based Mutation Testing for UML. [Online] http://www.ist.tugraz.at. Disponível em: http://www.ist.tugraz.at/aichernig/publications/papers/icst15-tools.pdf [Último acesso: 25 de Agosto de 2018].

[38] Bernhard K. Aichernig, B. Jakob Auer, J. Elisabeth Jöbstl, E. Robert Korošec, R. Willibald Krenn, W. Rupert Schlick, R. Birgit Vera Schmidt, B. (2014). Model-Based Mutation Testing of an Industrial Measurement Device. [Online] https://pdfs.semanticscholar.org. Disponível em: https://pdfs.semanticscholar.org/9333/2db9a23b3beb7dd3426daad229d2fb790b59.pdf [Último acesso: 25 de Agosto de 2018].

[39] David Adamo, D. (2016). [Online] https://github.com/mukatee. Disponível em: https://github.com/mukatee/osmo [Último acesso: 25 de agosto de 2018].

[40] Matii Vuori, M. (2014). User-centered model-based testing tool design. [Online] http://www.cs.tut.fi. Disponível em: http://www.cs.tut.fi/tapahtumat/testiautomaatio14/kalvot/ATAC_report_user_centered_MBT_tool_design.pdf [Último acesso: 25 de Agosto de 2018].

[41] Kerry Kimbrough, K. Juglar, J. Thibault Kruse, J. (2018). A model-based test case generator. [Online] https://github.com/Cornutum. Disponível em: https://github.com/Cornutum/tcases [Último acesso: 25 de agosto de 2018].

[42] Wenbin Li, W. Franck Le Gall, F. Naum Spaseski, N. A Survey on Model-Based Testing Tools for Test Case Generation [Online] https://www.researchgate.net. Disponível em: https://www.researchgate.net/publication/319141331_A_Survey_on_Model-

Based_Testing_Tools_for_Test_Case_Generation/download [Último acesso: 25 de agosto de 2018]

[43] Daisen Wei, D. Longye Tang, L. Xueqing Li, X. Ling Shang, L. (2014). [Online] http://www.jsoftware.us. Disponível em: http://www.jsoftware.us/vol9/jsw0911-13.pdf [Último acesso: 27 de agosto de 2018].

[44] Kerry Kimbrough, K. Juglar, J. Thibault Kruse, J. (2018). [Online] http://www.cornutum.org. Disponível em: http://www.cornutum.org/tcases/docs/Tcases-Guide.htm#intro [Último acesso: 28 de agosto de 2018].

[45] Axel Belinfante, A. (2016) JTorX: a Tool for On-Line Model-Driven Test Derivation and Execution [Online] https://link.springer.com. Disponível em: https://link.springer.com/chapter/10.1007/978-3-642-12002-2_21 [Último acesso: 29 de agosto de 2018].

[46] Axel Belinfante, A. (2016). JTorX: a tool for Model-Based Testing [Online] https://fmttools.ewi.utwente.nl. Disponível em: https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki [Último acesso: 19 de agosto de 2018].

[47] Ulf Eriksson, U. 3 Simple Tricks to Make Exploratory Testing More Efficient [Online] https://reqtest.com. Disponível em: https://reqtest.com/testing-blog/3-simple-tricks-to-make-exploratory-testing-more-efficient [Último acesso: 3 de setembro de 2018].

Anexo A

```
package com.company;
import org.graphwalker.core.machine.ExecutionContext;
import org.graphwalker.java.annotation.GraphWalker;
import org.junit.Assert;
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import java.util.List;
@GraphWalker
public class LoginTest extends ExecutionContext implements LoginSearch {
   WebDriver driver=null;
   @Override
   public void e_SearchSapoEmailPage() {

      driver.get("http://www.sapo.pt/");
   }

   @Override
   public void e_CleanCache() {

      driver.manage().deleteAllCookies();
   }

   @Override
   public void v_SapoSearchResult() {

      System.out.println("Obtido o resultado da pesquisa!");
   }

   @Override
   public void v_BrowserNotStarted() {

      System.out.println("O browser ainda não foi iniciado!");
   }

   @Override
   public void v_SearchResult() {

      System.out.println("Efetuar pesquisa!");
   }

   @Override
   public void e_Logout() {
      WebDriverWait wait = new WebDriverWait(driver, 20);
      wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='applogout']"))).click();
      driver.manage().deleteAllCookies();
   }
   @Override
   public void v_LoginPrompted() {

      System.out.println("Efetuar login!");
   }
```

```
@Override
  public void e_ClickOpenSapo() {
    WebElement username_txt = driver.findElement(By.xpath("//*[@id='bsu-todo-o-sapo']/ul/li[1]/a"));
    username_txt.click();

  }

  @Override
  public void e_ValidCredentials() {

    WebDriverWait wait = new WebDriverWait(driver, 20);

    Boolean isPresent =
driver.findElements(By.xpath("//*[@id='sapo_widget_login_form']/fieldset/div[1]/a")).size() > 0;
    if (isPresent==true){
      WebElement box =
driver.findElement(By.xpath("//*[@id='sapo_widget_login_form']/fieldset/div[1]/a"));
      box.click();
      wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='text-
email']"))).sendKeys("testesgraphwalker@sapo.pt");
      wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='text-
password']"))).sendKeys("Testesgraphwalker!199");
      WebElement button_login = driver.findElement(By.xpath("//*[@class='ink-button green large-
100']"));
      button_login.click();
    }else {
      WebElement username_txt = driver.findElement(By.xpath("//*[@id='text-email']"));
      username_txt.clear();
      wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='text-
email']"))).sendKeys("testesgraphwalker@sapo.pt");
      wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='text-
password']"))).sendKeys("Testesgraphwalker!199");
      WebElement button_login = driver.findElement(By.xpath("//*[@class='ink-button green large-
100']"));
      button_login.click();
    }
  }

  @Override
  public void v_BrowserOpened() {
    System.out.println("O browser está agora aberto!");
  }

  @Override
  public void e_Exit() {

    driver.close();
  }
  @Override
  public void e_OpenBrowser() {
    System.setProperty("webdriver.chrome.driver",
"/Users/manue/testAutomation_GraphWalker_MasterThesis/chromedriver.exe");
    driver = new ChromeDriver();
    driver.manage().window().maximize();

  }
```

```
@Override
public void v_EmailOpened() {
    System.out.println("O email foi aberto!");

    WebDriverWait wait = new WebDriverWait(driver, 20);
    wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@class='horde-subnavi-
point']/a[contains(text(), 'Caixa de Entrada')]")));


}

@Override
public void e_searchEmail() {

    WebDriverWait wait = new WebDriverWait(driver, 20);

        WebElement box =
wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@class='horde-subnavi-
point']/a[contains(text(), 'Caixa de Entrada')]")));
        box.click();
        wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='horde-search-
input']"))).sendKeys("sapo");
        WebElement enter_search = driver.findElement(By.xpath("//*[@id='horde-search-input']"));
        enter_search.sendKeys(Keys.ENTER);

}

@Override
public void e_CloseBrowser() {
    driver.quit();
}

@Override
public void v_LoginMade() {
    System.out.println("O login foi efetuado!");
}

@Override
public void v_CacheCleaned() {
    System.out.println("A cache foi limpa");
}

@Override
public void e_Init() {

    System.out.println("Início da execução");
}

@Override
public void v_SapoPageOpened() {
    System.out.println("Página sapo aberta!");
}
```

```
@Override
public void e_ClickToOpen() {

    List<WebElement> email = driver.findElements(By.xpath("//*[@id='VProw_3']/div[3]"));

    for (WebElement emailsub : email) {
      if (emailsub.getText().equals("Bem-vindo ao SAPO Mail") == true) {
        emailsub.click();
        break;
      }
    }
  }

  @Override
  public void e_ClickSignIn() {
    WebDriverWait wait = new WebDriverWait(driver, 20);

wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='loginModalTrigger']"))).click
();

  }

  @Override
  public void e_InvalidCredentials() {
    WebDriverWait wait = new WebDriverWait(driver, 20);

    Boolean isPresent =
driver.findElements(By.xpath("//*[@id='sapo_widget_login_form']/fieldset/div[1]/a")).size() > 0;

    if (isPresent==true){
      WebElement box =
driver.findElement(By.xpath("//*[@id='sapo_widget_login_form']/fieldset/div[1]/a"));
      box.click();
      wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='text-
email']"))).sendKeys("sdfsdfsdf@sapo.pt");
      wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='text-
password']"))).sendKeys("sdfsdf!dsfs");

      WebElement button_login = driver.findElement(By.xpath("//*[@class='ink-button green large-
100']"));
      button_login.click();
    }else {
      wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='text-
email']"))).sendKeys("sdfsdfsdf@sapo.pt");
      wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='text-
password']"))).sendKeys("sdfsdf!dsfs");

      WebElement button_login = driver.findElement(By.xpath("//*[@class='ink-button green large-
100']"));
      button_login.click();
    }

    WebElement errorMessage = driver.findElement(By.xpath("//*[@class='login_failed'][contains(text(),
'O login falhou.')]"));
    Assert.assertEquals(errorMessage.getText(), "O login falhou.");

  }
}
```

Anexo B

```
@GraphWalker
public class LoginTest extends  ExecutionContext implements LoginSearchApp {
    AppiumDriver<?> driver;
    DesiredCapabilities capabilities = new DesiredCapabilities();

    @Override
    public void e_OpenApp() {
        String apkpath =
"/Users/manue/testAutomation_GraphWalker_MasterThesis_Appium/src/main/resources/apkandroid2.
apk";
        File app = new File(apkpath);

        capabilities.setCapability("deviceName", "Nexus 5 API 27");
        capabilities.setCapability("PlatformVersion", "8.1.0");
        capabilities.setCapability("PlatformName", "Android");
        capabilities.setCapability("app", app.getAbsolutePath());
        capabilities.setCapability("appiumVersion", "1.8.0");
        capabilities.setCapability("AutomationName","Appium");
        capabilities.setCapability("fullReset","true");
        try {
            driver = new AndroidDriver(new URL("http://0.0.0.0:4723/wd/hub"), capabilities);

        } catch (MalformedURLException e) {
            e.printStackTrace();
        }

        System.out.println("A app foi iniciada com sucesso.");

    }

    @Override
    public void e_invalidCredentials() {

    }
    @Override
    public void v_SearchResult() {

    }

    @Override
    public void v_LoginPrompted() {

    }

    @Override
    public void e_Exit() {
        driver.resetApp();
        driver.quit();
        System.out.println("Passei pelo exit!");

    }
```

```
    @Override
    public void v_AppOpened() {

        System.out.println("A APP está agora aberta!");
    }

    @Override
    public void e_selectEmailProvider() {
        driver.manage().timeouts().implicitlyWait(15,TimeUnit.SECONDS);

        MobileElement el1 = (MobileElement)
driver.findElementByXPath("/hierarchy/android.widget.FrameLayout/android.widget.LinearLayout/andro
id.widget.FrameLayout/android.widget.FrameLayout/android.widget.FrameLayout/android.widget.Relat
iveLayout/android.widget.FrameLayout/android.widget.RelativeLayout/android.widget.LinearLayout[1]/
android.widget.ListView/android.widget.TextView");
        el1.click();


  }
  @Override
  public void v_EmailOpened() {

  }

  @Override
  public void v_AppNotStarted() {

     System.out.println("A app ainda não foi inicializada!");
  }

  @Override
  public void e_searchEmail() {

     MobileElement el1 = (MobileElement) driver.findElementByAccessibilityId("Pesquisar");
     el1.click();

     MobileElement el2 = (MobileElement) driver.findElementById("ru.mail.mailapp:id/search_text");
     el2.sendKeys("bem vindo"+"\n");

  }

  @Override
  public void v_LoginMade() {

  }

  @Override
  public void e_validCredentials() {
     MobileElement e12 = (MobileElement) driver.findElementById("ru.mail.mailapp:id/login");
     e12.sendKeys("testesgraphwalker@sapo.pt");

     MobileElement e13 = (MobileElement) driver.findElementById("ru.mail.mailapp:id/password");
     e13.sendKeys("Testesgraphwalker!199");
```

```
    MobileElement e14 = (MobileElement) driver.findElementById("ru.mail.mailapp:id/sign_in");
    e14.click();
  }

  @Override
  public void e_CloseApp() {
    driver.resetApp();
    driver.quit();
    System.out.println("Fechei a APP!");
  }

   @Override
  public void e_logout() {

      MobileElement el1 = (MobileElement) driver.findElementByAccessibilityId("Navegar para cima");
      el1.click();

      MobileElement el2 = (MobileElement)
driver.findElementByXPath("/hierarchy/android.widget.FrameLayout/android.widget.LinearLayout/andro
id.widget.FrameLayout/android.widget.FrameLayout/android.widget.FrameLayout/android.widget.Relat
iveLayout/ru.mail.fragments.view.drawer.DrawerLayout/android.widget.RelativeLayout[2]/android.widg
et.LinearLayout/android.widget.ListView/android.widget.LinearLayout[10]/android.widget.RelativeLayo
ut/android.widget.TextView");
      el2.click();

      MobileElement el3 = (MobileElement) driver.findElementById("android:id/button1");
      el3.click();
    System.out.println("Fiz logout na APP!");

  }
  @Override
  public void e_Init() {
    System.out.println("Iniciou a execução.");
  }
  @Override
  public void e_ClickToOpen() {
    MobileElement el9 = (MobileElement) driver.findElementById("ru.mail.mailapp:id/sender");
    el9.click();
  }
    @Override
    public void e_giveAccessApp() {
      MobileElement el5 = (MobileElement) driver.findElementById("ru.mail.mailapp:id/button_skip");
      el5.click();

      MobileElement el6 = (MobileElement) driver.findElementById("ru.mail.mailapp:id/button_skip");
      el6.click();
      MobileElement el7 = (MobileElement) driver.findElementById("ru.mail.mailapp:id/welcome_done");
      el7.click();

    }

    @Override
    public void v_appAccessGiven() { driver.manage().timeouts().implicitlyWait(5,TimeUnit.SECONDS);
      MobileElement e20 = (MobileElement)
driver.findElementById("ru.mail.mailapp:id/mail_fragment_frame");
      e20.click();
driver.manage().timeouts().implicitlyWait(2,TimeUnit.SECONDS);
    }
  }
```

Anexo C

[ERROR] org.openqa.selenium.NoSuchElementException: no such element: Unable to locate element: {"method":"xpath","selector":"//*[@id='text-email']"}
  (Session info: chrome=68.0.3440.106)
  (Driver info: chromedriver=2.40.565498 (ea082db3280dd6843ebfb08a625e3eb905c4f5ab),platform=Windows NT 10.0.16299 x86_64) (WARNING: The server did not provide any stacktrace informa
tion)

"totalFailedNumberOfModels": 1,
  "totalNotExecutedNumberOfModels": 0,
  "totalNumberOfUnvisitedVertices": 2,
  "verticesNotVisited": [
    {
      "modelName": "LoginSearchError",
      "vertexName": "v_SearchResult",
      "vertexId": "n8"
    },
    {
      "modelName": "LoginSearchError",
      "vertexName": "v_EmailOpened",
      "vertexId": "n9"
    }
  ],
  "totalNumberOfModels": 1,
  "totalCompletedNumberOfModels": 0,
  "totalNumberOfVisitedEdges": 10,
  "totalIncompleteNumberOfModels": 0,
  "edgesNotVisited": [
    {
      "modelName": "LoginSearchError",
      "edgeId": "e8",
      "edgeName": "e_Exit"
    },
    {
      "modelName": "LoginSearchError",
      "edgeId": "e10",
      "edgeName": "e_searchEmail"
    },
    {
      "modelName": "LoginSearchError",
      "edgeId": "e11",
      "edgeName": "e_ClickToOpen"
    },
    {
      "modelName": "LoginSearchError",
      "edgeId": "e12"
    }
  ],
  "vertexCoverage": 77,
  "totalNumberOfEdges": 14,
  "totalNumberOfVisitedVertices": 7,
  "edgeCoverage": 71,
  "totalNumberOfVertices": 9,
  "totalNumberOfUnvisitedEdges": 4
}

Anexo D

```
Início da execução
O browser ainda não foi iniciado!
Abrir o browser!
Starting ChromeDriver 2.40.565498 (ea082db3280dd6843ebfb08a625e3eb905c4f5ab) on port 28844
Only local connections are allowed.
set 09, 2018 9:44:16 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
O browser está agora aberto!
Limpar a cache do website!
A cache foi limpa
Abrir página do sapo!
Obtido o resultado da pesquisa!
Clicar para abrir página do sapo!
Página sapo aberta!
Clicar para fazer o login!
Efetuar login!
Introduzir credenciais válidas!
O login foi efetuado!
Done: [{
  "totalFailedNumberOfModels": 0,
  "totalNotExecutedNumberOfModels": 0,
  "totalNumberOfUnvisitedVertices": 2,
  "verticesNotVisited": [
    {
      "modelName": "LoginSearchError",
      "vertexName": "v_SearchResult",
      "vertexId": "n8"
    },
```

```
  {
   "modelName": "LoginSearchError",
   "vertexName": "v_EmailOpened",
   "vertexId": "n9"
  }
 ],
 "totalNumberOfModels": 1,
 "totalCompletedNumberOfModels": 1,
 "totalNumberOfVisitedEdges": 7,
 "totalIncompleteNumberOfModels": 0,
 "edgesNotVisited": [
  {
   "modelName": "LoginSearchError",
   "edgeId": "e5",
   "edgeName": "e_InvalidCredentials"
  },
  {
   "modelName": "LoginSearchError",
   "edgeId": "e8",
   "edgeName": "e_Exit"
  },
  {
   "modelName": "LoginSearchError",
   "edgeId": "e9",
   "edgeName": "e_CloseBrowser"
  },
  {
   "modelName": "LoginSearchError",
   "edgeId": "e10",
   "edgeName": "e_searchEmail"
  },
```

```
    {
      "modelName": "LoginSearchError",
      "edgeId": "e11",
      "edgeName": "e_ClickToOpen"
    },
    {
      "modelName": "LoginSearchError",
      "edgeId": "e12"
    },
    {
      "modelName": "LoginSearchError",
      "edgeId": "e13",
      "edgeName": "e_logout"
    }
  ],
  "vertexCoverage": 77,
  "totalNumberOfEdges": 14,
  "totalNumberOfVisitedVertices": 7,
  "edgeCoverage": 50,
  "totalNumberOfVertices": 9,
  "totalNumberOfUnvisitedEdges": 7
}]
```