Maria Adriana Neto Fonseca

Licenciada em Ciências da Engenharia

Desenvolvimento de testes automatizados para *backend*

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: David Ribeiro, Head of Products Development, Thales Group Portugal

Co-orientador: João Baptista da Silva Araújo Junior, Professor Associado, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri

Presidente: Professor Doutor Hervé Miguel Cordeiro Paulino, FCT-UNL
Arguente: Professor Doutor Artur Miguel de Andrade Vieira Dias, FCT-UNL
Vogal: Engenheiro David Ribeiro, Thales Group Portugal

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

fevereiro, 2021

Desenvolvimento de testes automatizados para *backend*

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Agradecimentos

Quero deixar o meu primeiro agradecimento ao professor João Araújo que, tal como os testes automatizados desta dissertação, facultou o seu parecer rapidamente e de forma útil, ajudando a realizar o potencial deste trabalho. Aos professores da FCT–UNL que demonstraram paixão pelo seu trabalho, assim como respeito pelos seus alunos, deixo também o meu agradecimento pela motivação que me proporcionaram.

Aos meus três orientadores da Thales, que me apoiaram em momentos diferentes, em concreto ao David Ribeiro, agradeço o apoio fornecido, sei que foi uma *uphill battle*.

Agradeço aos meus pais, cujo esforço, aliado à sorte, me possibilitou forjar o meu caminho, uma e outra vez; e à minha irmã, por me mostrar que é possível.

Sem dúvida que esta vida seria bem diferente sem as amizades que experienciei, não só durante a dissertação, mas também durante todo o percurso universitário. O meu agradecimento ao Marco M., por ter aparecido no momento certo e ter ficado para todos os outros; aos amigos de sangue nortenho, Marta G. e Miguel P., por partilharmos tantas aventuras; ao André R., por me mostrar o que um amigo realmente deve ser; e à Sofia S., por ser a melhor amiga bem antes de eu saber o que era uma universidade.

Resumo

Devido à evolução da tecnologia e da dependência do ser humano da mesma, pode afirmar-se que garantir o bom funcionamento de um software é crucial. Para desenvolver um programa robusto é necessário haver um investimento em diversas áreas da engenharia, porém, a execução de testes é a maneira mais eficaz de avaliar esse investimento, comprovando a qualidade e credibilidade do software.

O paradigma atual do mercado exige entregas regulares de partes selecionadas do software, ao longo de todo o desenvolvimento. A automatização de testes diminui significativamente o tempo de execução dos mesmos. Para além disto, aumenta a consistência entre os testes automatizados e liberta tempo aos profissionais para se concentrarem noutros tipos de testes.

A corrente dissertação foi desenvolvida no ambiente empresarial da *Thales Group Portugal*, empresa que fornece produtos e serviços para a indústria dos transportes terrestres. O caso de estudo foca-se no Advanced Passenger Information System 8, um projeto que fornece uma plataforma multifuncional de mensagens informativas ao passageiro. As técnicas desenvolvidas na presente dissertação são aplicadas a esse produto, construindo um conjunto de casos de testes automatizados para garantir a qualidade do *backend* do mesmo.

Para além dos benefícios inerentes à automatização de testes, pretende-se também construir os mesmos usando uma sintaxe muito semelhante à linguagem natural, para melhorar a comunicação entre todos os *stakeholders*. Para este propósito a construção dos testes automatizados será feita no *Robot Framework*, usando a linguagem *Gherkin*.

Palavras chave: Automatização de testes, Robot Framework, Gherkin, *backend*, behavior-driven development, keyword-driven testing, testes de aceitação.

Abstract

Due to the evolution of technology and human dependence on it, it can be said that ensuring the proper functioning of software is crucial today. To develop a robust program it is necessary to invest in several areas of engineering, however the execution of tests is the most effective way to evaluate this investment, proving the quality and credibility of the software.

The current market paradigm requires regular deliveries of selected pieces of software throughout the development. Automating tests significantly decreases test execution times. In addition, it increases the consistency between automated tests and frees up time for professionals to focus on other types of tests.

The current dissertation will be developed in the business environment of Thales Group Portugal, a company that provides products and services for the land transport industry. The case study focuses on the Advanced Passenger Information System 8, a project that provides a multifunctional platform for informational messages to the passenger. The techniques developed in this dissertation are applied to this product, building a set of automated test cases to ensure the quality of the backend.

In addition to the benefits inherent to the automation of tests, it is also intended to build them using a syntax very similar to natural language, to improve communication between all stakeholders. For this purpose, the construction of automated testing will be done on the Robot Framework, using the Gherkin language.

Keywords: Test automation, Robot Framework, Gherkin, backend, behavior-driven development, keyword-driven testing, acceptance test.

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Siglas

AAT	*Automated Acceptance Testing*.
API	*Application Programming Interface*.
APIS	*Advanced Passenger Information System*.
APIS8	*Advanced Passenger Information System 8*.
BDD	*Behavior-Driven Development*.
CI	*Continuous Integration*.
CTT	Correios, Telégrafos e Telefones.
GDP	*Ground Transportation Systems – Digital Platform*.
HMI	*Human-Machine Interface*.
IDE	*Integrated Development Environment*.
IVVQ	*Integration, Verifcation, Validation and Qualification*.
IVVQP	*Integration, Verification, Validation and Qualification Plan*.
JSON	*JavaScript Object Notation*.
QA	*Quality Assurance*.
REST	*Representational State Transfer*.
RF	*Robot Framework*.
STD	*Software Test Description*.
STP	*Software Test Plan*.
UI	*User Interface*.
URL	*Uniform Resource Locator*.
UX	*User Experience*.

Capítulo 1

Introdução

1.1 Contexto e motivação

À medida que a tecnologia evolui e que o software se difunde por todos os aspetos da vida, os requisitos tornam-se mais exigentes relativamente à fiabilidade, manutenção e segurança [1].

"Modern (2008) households have over 50 processors, and some new cars have over 100; all of them running software that optimistic consumers assume will never fail!"[1].

Um software confiável depende de múltiplos fatores de engenharia como, por exemplo, *design* cuidado, boa gestão de processos, entre outros. Contudo, a execução de testes é a melhor maneira de avaliar o software e assegurar a sua credibilidade [1].

O método de desenvolvimento de produtos informáticos tem vindo a evoluir significativamente em resposta às expetativas do mercado, regendo-se pela valorização das necessidades do cliente. O espaço de tempo em que é desenvolvido e entregue uma parte selecionada do software é denominado por ciclo e, ao longo de todo o desenvolvimento de um produto, existem múltiplos ciclos. Como é esperada grande adaptabilidade a novos requisitos a qualquer momento, quanto menor for o ciclo de uma entrega intermédia, menor é o tempo esperado de *feedback*. Deste modo, com *feedback* atempado e respetivas alterações feitas, o produto avança no desenvolvimento com maior confiança [10]. A elaboração e execução de testes de software é uma parte vital do desenvolvimento visto que fornece informação sobre a qualidade do mesmo e, como tal, deve estar inserida em todas as fases, de modo a amadurecer com o produto em questão [12]. Assim, é assegurado o bom funcionamento do software e reduzido o risco de falhas durante a fase de produção [12]. Embora a execução dos testes seja vital, a sua execução manual tem desvantagens que podem ser colmatadas por uma abordagem automática. Testes automatizados são

executados sem interação humana e, possivelmente, de modo cíclico. Em geral, a automatização de testes selecionados possibilita a redução dos ciclos de desenvolvimento, tornando cada um mais eficiente, dado que [11]:

- Existe um maior número de testes a serem executados por cada ciclo;
- É possível executar testes complexos, que poderiam mesmo ser inviáveis de executar manualmente;
- São executados mais rapidamente;
- Não são sujeitos a erros induzidos pelo utilizador;
- Utilizam eficientemente os recursos de testes, quer sejam eles tecnológicos, de tempo ou humanos;
- Possibilitam rapidamente um *feedback* sobre a qualidade do software;
- Aumentam a fiabilidade do sistema, devido à maior consistência dos testes e a sua repetibilidade;
- Existe maior consistência de testes entre ciclos pois têm menor intervenção humana;
- Resultam na redução de custo/tempo na execução de testes manuais.

Devido à evolução da tecnologia e da dependência do ser humano da mesma, é possível afirmar que testar software é crucial nos dias de hoje. Um programa que não desempenhe as funções esperadas pode, no melhor dos casos, resultar em custos financeiros, desperdício de tempo, perda de reputação e, no pior dos casos, em sistemas críticos, resultar em danos físicos ou morte do utilizador [12].

Considerando que o código desenvolvido tem vindo a ser mais complexo e vasto, testar o mesmo é um processo cuja complexidade dá origem ao consumo de múltiplos recursos, como o tempo, resultando em custos monetários. O esforço associado a testar um software perfaz cerca de 50% das despesas totais de desenvolvimento, podendo ser um valor mais elevado no caso de aplicações críticas. Existe uma vasta necessidade em testar código o mais antecipadamente para evitar custos desnecessários na correção tardia de defeitos em elementos básicos do sistema. Porém, quase a totalidade destes testes é passível de ser automatizada, permitindo minimizar os erros induzidos, facilitar os testes de regressão e reduzir os custos de longo prazo [1].

O estado atual do mercado de software implica uma enorme variedade de profissionais envolvidos em cada projeto, onde a comunicação é crucial para se desenvolver não só o produto corretamente, mas também para se desenvolver o produto correto. Na presente dissertação é abordado também este ponto, concretamente, os testes automatizados desenvolvidos têm ênfase na facilidade de interpretação por qualquer profissional associado ao projeto.

Embora os testes automatizados se encontrem a ser elaborados na fase de manutenção do produto e não em conjunto com o desenvolvimento, são uma boa base de automatização para a empresa. O projeto onde esta dissertação se insere é um de muitos projetos desenvolvidos e mantidos pela empresa. Como tal, ao construir uma base de testes automatizados, bem como toda a mentalidade associada, é feito um investimento que facilitará futuros projetos, ou projetos existentes, que queiram incorporar esta vertente. Deste modo, a empresa consegue diminuir os variados custos a longo prazo, quer no desenvolvimento ou na manutenção.

1.2 Objetivos

Na presente tese foram desenvolvidos testes automatizados para *backend* usando técnicas da metodologia *Behavior-Driven Development* (BDD), através da utilização de *keywords* e da sintaxe *Gherkin*.

Pretende-se alcançar um grande volume de testes automatizados que corram frequentemente, dando garantias que o software continua a funcionar como esperado e, caso haja alguma possível falha, esta é capturada assim que é despoletada.

O desenvolvimento destes testes automatizados tem ênfase na sua legibilidade por qualquer pessoa, sendo que será desenvolvido código numa linguagem de alto nível e semelhante à linguagem natural. Assim, espera-se que os testes possam ser compreendidos por todos os elementos envolvidos no projeto e, por conseguinte, minimizar os problemas induzidos por uma má comunicação.

Dado o nível de abstração da linguagem e a sua legibilidade, pretende-se que o conjunto de casos de teste automatizados seja o ponto de partida para uma nova documentação de testes. A documentação existente de testes de *backend* para um projeto do APIS8 não é de fácil compreensão e não é suficientemente geral para se aplicar a todos os outros projetos.

A meta final para a empresa será garantir a qualidade dos seus produtos, através de documentação útil e comunicação eficaz e, consequentemente, obter ainda maior destaque no mercado.

1.3 Principais contribuições

À medida que a quantidade de testes automatizados aumenta, o tempo que previamente era investido em testes manuais poderá ser investido na realização de outros tipos de testes, tais como, testes de *performance* ou segurança.

Outra contribuição inerente à elaboração deste tipo de testes, será a poupança de tempo investido, quer na execução dos testes, quer na correção dos erros detetados, sendo que este último resulta diretamente em poupança monetária.

Para além da contribuição direta para o projeto em estudo – *Advanced Passenger Information System 8* (APIS8)–, considera-se a construção de código de teste consistente

e reutilizável para outros projetos baseados no *Advanced Passenger Information System* (APIS), visto que este produto não tinha qualquer nível de automatização. Isto tem bastante valor pois já existem múltiplos projetos relativos ao APIS em produção e também se visiona construir novas versões futuramente.

De notar que qualquer produto de qualidade eleva o nível do mercado, sendo que marca novos padrões de qualidade e práticas a adotar. Espera-se que a componente de automatização de testes no APIS8 seja um exemplo a seguir, não só em produtos internos à empresa, mas também no mercado em geral.

1.4 Estrutura

A presente dissertação encontra-se dividida em seis capítulos, adotando a seguinte organização:

- Capítulo 2 – Enquadramento: Visão geral sobre a empresa, o projeto e a equipa onde a tese foi elaborada. Explicação de diversos conceitos relacionados com o método de desenvolvimento praticado na empresa. Introdução ao mundo dos testes e à necessidade da sua automatização;
- Capítulo 3 – Trabalho relacionado: Exposição de cinco projetos que abordam temas semelhantes ao projeto em foco, nomeadamente, o impacto da automatização de testes, a execução de testes de aceitação, a utilização de *frameworks* de automatização e a transição para um desenvolvimento *Agile*;
- Capítulo 4 – Processo de automatização: Apresentação dos objetivos a alcançar e demonstração de um protótipo, assim como exposição das principais ferramentas utilizadas e descrição das atividades necessárias para alcançar os objetivos traçados;
- Capítulo 5 – Avaliação: Exposição e análise do volume de trabalho através de múltiplas métricas, quer do código desenvolvido, como da execução dos testes;
- Capítulo 6 – Conclusão: Reflexão sobre o trabalho realizado e concretização dos objetivos traçados inicialmente. Exposição de alguns pontos possíveis para a continuidade do trabalho.

Capítulo 2

Enquadramento

Neste capítulo são apresentados e explicados conceitos relativos ao trabalho elaborado. Desde o contexto em que a presente dissertação se insere, passando por metodologias de trabalho e finalizando em abordagens concretas de testes.

2.1 *Thales Group*

A *Thales Group* é uma empresa internacional que se foca no desenvolvimento de serviços para indústrias de defesa e segurança, identidade digital e segurança, aeroespacial, espaço (sistemas para satélites) e transporte. Em Portugal, a empresa centraliza-se na indústria dos transportes terrestres, fornecendo produtos e soluções para sistemas de segurança e informação ao passageiro. Os múltiplos projetos englobam aspetos de sinalização, comunicação e supervisão, gestão da cobrança de tarifas (*Fare Collection Management*), serviços de manutenção, cibersegurança na infraestrutura ferroviária e digitalização dos serviços da ferrovia (*Railways Digitalisation*) [29, 30].

2.1.1 APIS8

O *Advanced Passenger Information System* (APIS) desenvolvido pela Thales é definido como sendo uma plataforma multifuncional de mensagens informativas ao passageiro. Tem como objetivo gerir e fornecer uma variedade de informações aos passageiros distribuídos geograficamente, através de dispositivos de áudio e visuais, como exemplificados na figura 2.1 [33].

Para tal, o APIS incorpora, numa só *framework*, todas as capacidades, infraestruturas e recursos de processamento necessários. Existe um conjunto de funcionalidades que destacam o APIS [33]:

- Mensagens automáticas;

- Reutilização do conteúdo de mensagens;
- Suporte de múltiplas linguagens para mensagens de texto, de áudio e de imagens;
- Sistema de mensagens programável e cíclico;
- Representação da rede de transportes;
- Arquitetura modular.

Figura 2.1: Visão geral da solução APIS

Este produto base está no mercado há cerca de 20 anos, disperso por todo o globo. Tal é possível devido à capacidade de personalização do produto para cada cliente e à adaptabilidade a novas tecnologias. Atualmente o APIS está presente, por exemplo, na Arábia Saudita (APIS3 e APIS7), na Índia (APIS3), no Panamá (APIS7) e no Brasil (APIS7), entre outros.

De entre os múltiplos projetos em desenvolvimento ou manutenção, o projeto elaborado trabalha sobre uma versão específica do produto, o APIS8, cujo desenvolvimento se iniciou há 6 anos, tendo sido colocado no mercado há 3 anos (em 2017). No momento de elaboração da presente tese, o projeto do APIS8 encontra-se em Portugal, no Qatar, no Dubai, na Polónia, em Taiwan, no Senegal e na Austrália.

As interfaces que o APIS incorpora são essencialmente de *back-office*, onde é possível controlar o sistema. A figura 2.2 representa a interface do APIS utilizado no Panamá, onde podemos ver o *layout* da aplicação, concretamente, a interface apresenta a disposição geográfica de uma linha específica [33].

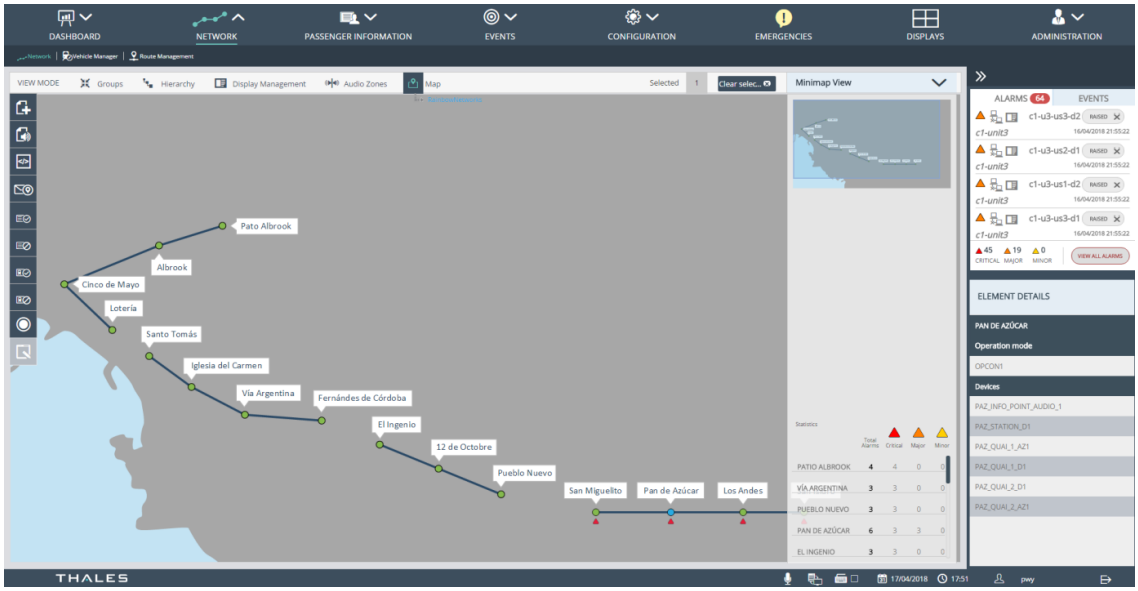


Figura 2.2: APIS, versão 7, projeto do Panamá

2.1.2 Atividades IVVQ

Existe um conjunto de atividades no âmbito da integração, verificação, validação e qualificação – *Integration, Verifcation, Validation and Qualification* (IVVQ) – que devem ser executadas ao longo da vida útil de um produto. No ambiente da Thales existe uma definição concreta dessas atividades para cada produto, sendo que se encontra no documento *Integration, Verification, Validation and Qualification Plan* (IVVQP). As atividades incluem análise de requisitos, engenharia de sistemas, criação de procedimentos de testes e garantir que todas as falhas são identificadas, documentadas e controladas [14].

Ao longo do ciclo de vida do software, todos os elementos do projeto devem assegurar a qualidade do mesmo, em todas as vertentes. Contudo, dada a fase de manutenção em que o produto se encontra, existe uma equipa multidisciplinar que se foca na elaboração e execução de testes, automatizados e manuais, assim como na atualização da respetiva documentação. Para auxiliar as atividades da equipa nesta fase de testes, existe documentação que deve ser tida em consideração, nomeadamente, o *Software Test Plan* (STP) , que define a abordagem de testes para ambientes, níveis e classes distintas, ou seja, define quais os testes de interesse. O *Software Test Description* (STD) descreve em detalhe os casos de teste – escolhidos no STP – e os correspondentes procedimentos a executar.

O planeamento e a definição de prioridades das atividades IVVQ é feita pelo *Product Owner*, em colaboração com a equipa. Existe uma auto-gestão por parte da equipa relativamente à resolução de problemas e à divisão de tarefas, quer de testes ou de documentação. Todas as semanas é avaliada, de modo interno, a produtividade da equipa e definido o plano para a semana seguinte, de modo a existir um acompanhamento claro das tarefas por toda a equipa.

2.2 Desenvolvimento *Agile*

Durante as duas últimas décadas houve uma grande necessidade de alterar o modo de desenvolvimento de software. O mercado digital para programas de software exige facilidade na alteração de requisitos e entregas cada vez mais rápidas desses mesmos requisitos. Para atender esta crescente necessidade, as metodologias praticadas necessitaram tornar-se mais leves e ágeis, usando desenvolvimento iterativo, protótipos, *templates* e requisitos de documentação menores [16].

"The Agile Manifesto" foi escrito no início do milénio (2001) para definir estas novas necessidades. Segundo esse documento, *Agile* pode ser considerada uma filosofia de desenvolvimento, que se rege por quatro valores base (e os consequentes doze princípios)[19]:

1. *"Indivíduos e interações sobre processos e ferramentas;*
2. *Software utilizável sobre documentação abrangente;*
3. *Colaboração do cliente sobre negociação de contratos;*
4. *Reagir à mudança sobre seguir um plano."*

A filosofia *Agile* tem múltiplas implementações concretas, nomeadamente, *Extreme Programming*, *Crystal Methodologies*, SCRUM, entre outras [6].

Para além destas *frameworks* originais, é normal empresas evoluírem para novas *frameworks* que se adaptem ainda melhor às necessidades dos seus projetos e colaboradores, chegando a ser adotadas por outras empresas com necessidades semelhantes. O conceito *Agile* visa um uso mais consciente dos recursos disponibilizados, sejam eles horas úteis de trabalho ou a criação de documentação. *"Acolhemos a documentação, mas não para desperdiçar resmas de papel em volumes nunca mantidos e raramente usados"* [6].

O foco está nas pessoas e no modo real como trabalham; onde é normal existir imprevistos, a comunicação é a chave.

2.3 *Behavior-driven development*

Behavior-Driven Development (BDD), traduzido para "desenvolvimento orientado ao comportamento", é uma técnica na qual desenvolvedores, *testers* e clientes trabalham em grupo para analisar os requisitos do sistema de software, formulando os mesmos através de uma linguagem compreendida por todos e, por fim, verificando esses requisitos automaticamente. Os requisitos são expostos através de *user stories*, possíveis de serem transpostos diretamente para cenários de testes automatizados [13].

Em técnicas anteriores, existia uma grande falha de comunicação entre as pessoas envolvidas no processo e também uma ausência de testes automatizados. O BDD veio colmatar estas lacunas, definindo as especificações em termos do comportamento desejado da aplicação, e não do estado desejado, e sempre com um nível de abstração que facilite a transposição dos mesmos para testes automatizados [26].

Segundo Dan North, que descobriu a necessidade do BDD [8]: *"Se pudéssemos desenvolver um vocabulário consistente para analistas, testers, desenvolvedores e para o negócio, estaríamos num bom caminho para eliminar parte da ambiguidade e das falhas de comunicação que ocorrem quando os pessoas da área técnica falam com empresários."*

No projeto APIS8 não foi aplicado o processo de BDD por completo, visto que o mesmo já se encontrava em fase de manutenção, ou seja, completamente desenvolvido, no momento da transição. Todavia, na elaboração dos testes podem-se encontrar técnicas caraterísticas de BDD, nomeadamente [26]:

1. Os testes são de aceitação, passíveis de serem automatizados;
2. Os testes verificam o comportamento do software;
3. Os testes são escritos numa linguagem ubíqua.

2.3.1 Linguagem *Gherkin*

O *Gherkin* é uma linguagem de domínio específico construída para descrever o comportamento da aplicação. A base do *design* desta linguagem de programação é ser compreensível por qualquer ser humano que saiba linguagem natural, em qualquer uma das 70 línguas suportadas [5], sendo assim um componente importante no âmbito BDD.

Existe um conjunto de palavras-chave reservadas para estruturar frases em *Gherkin*, tais como: *Given*, *When*, *Then*, *And*, *But* [32].

Geralmente as frases construídas em *Gherkin* tomam a seguinte ordem [32]:

- "*Given*/Dado o contexto inicial" – pré-condição;
- "*When*/Quando um evento acontece" – *trigger*/ação catalisadora;
- "*Then*/Então acontece o resultado esperado" – pós-condição.

Supondo que estamos perante um software de uma máquina de café, existe uma funcionalidade que é servir café. Face a esta funcionalidade, podem ser escritos múltiplos casos de testes, tanto para testar casos positivos como casos negativos.

```
Scenario: Buy last coffee
  Given there are 1 coffees left in the machine
  And I have deposited 1 dollar
  When I press the coffee button
  Then I should be served a coffee
```

Figura 2.3: Teste com *Gherkin* – *Buy last coffee*

No exemplo apresentado na figura 2.3, é testada a funcionalidade de servir café quando existe apenas mais uma dose de café [3].

No contexto desta dissertação, a nomenclatura do *Gherkin* será aplicada à automatização. Apesar da *framework* escolhida para escrever os testes não interpretar a linguagem, a sua sintaxe será utilizada para estruturar o código dos mesmos. A utilização desta estrutura, aliada a uma escrita de alto nível garante a legibilidade e compreensão do código de teste.

2.3.2 *Robot Framework*

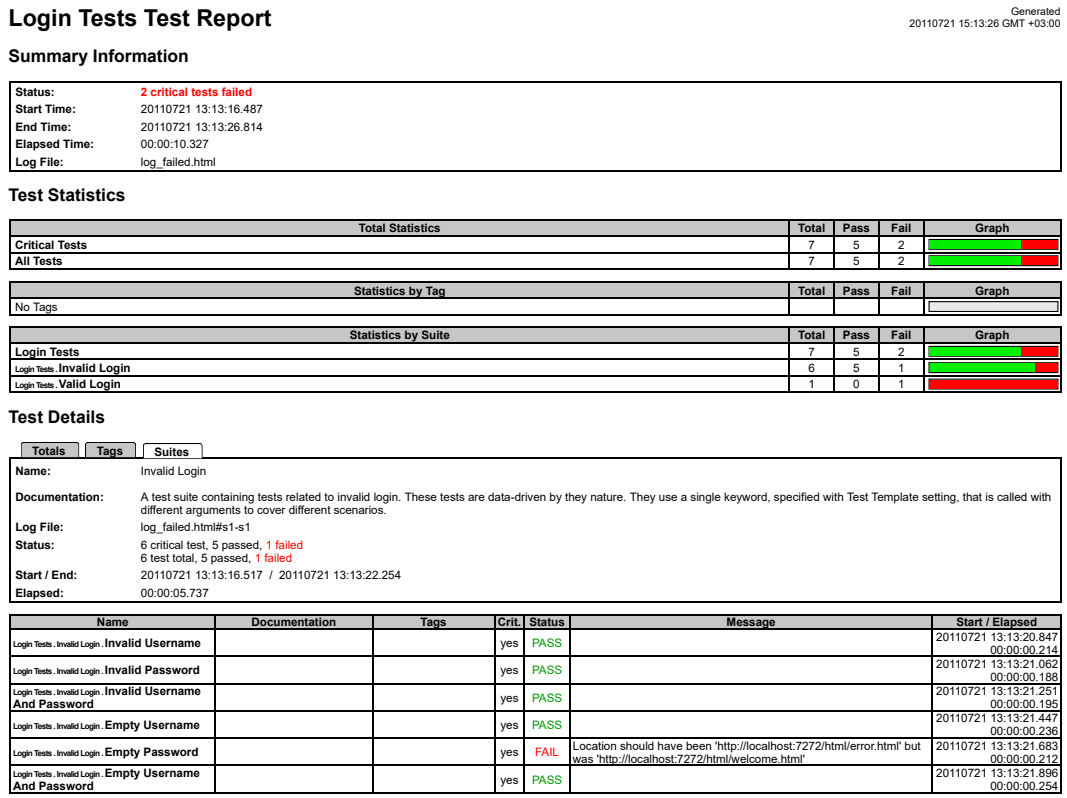
Esta *framework* de automatização de testes de aceitação tem por base uma sintaxe tabular e uma abordagem através de palavras-chave (*keyword-driven*).

O *Robot Framework* (RF) oferece um conjunto de *keywords* através de bibliotecas internas e de bibliotecas externas, mediante importação daquelas consideradas úteis. Consoante as necessidades dos testes a serem realizados, o *Robot Framework* possibilita a criação de novas *keywords*, através da elaboração de código em *Java* ou *Python*, originando assim as próprias bibliotecas do utilizador.

Possibilita também a produção de *keywords* de alto nível que representem ações mais complexas, denominadas de *keywords* de utilizador – *user keywords*. Estas têm por base *keywords* já existentes, abstraindo detalhes técnicos de teste e aumentando a extensibilidade e a reutilização da *framework* [27], sendo esta uma das mais valias do *Robot Framework*.

Dada a manipulação de *keywords* facilitadas pelo RF, é possível atingir a essência da prática BDD que é uma comunicação eficaz, mesmo através de documentação. Concretamente, possibilita escrever código mais organizado, detalhado e com diferentes níveis de abstração, sendo que o mais elevado se assemelha a linguagem natural.

Outro aspeto positivo desta *framework* é o tipo de relatório gerado no final de cada execução. As informações são fornecidas numa página *web* como se pode analisar de seguida. É possível ver na imagem 2.4 um exemplo de um relatório do RF [18]. Este apresenta um conjunto de métricas sobre os conjuntos de casos de teste, como por exemplo, o estado dos testes – neste caso, chumbaram dois testes críticos –, o tempo total de execução (19,327 segundos), quantos e quais os testes que passaram ou chumbaram, assim como o tempo de execução individual.



Login Tests Test Report

Generated
20110721 15:13:26 GMT +03:00

Summary Information

Status:	**2 critical tests failed**
Start Time:	20110721 13:13:16.487
End Time:	20110721 13:13:26.814
Elapsed Time:	00:00:10.327
Log File:	log_failed.html

Test Statistics

Total Statistics	Total	Pass	Fail	Graph
Critical Tests	7	5	2	
All Tests	7	5	2	

Statistics by Tag	Total	Pass	Fail	Graph
No Tags				

Statistics by Suite	Total	Pass	Fail	Graph
Login Tests	7	5	2	
Login Tests . **Invalid Login**	6	5	1	
Login Tests . **Valid Login**	1	0	1	

Test Details

Totals | Tags | Suites

Name:	Invalid Login
Documentation:	A test suite containing tests related to invalid login. These tests are data-driven by they nature. They use a single keyword, specified with Test Template setting, that is called with different arguments to cover different scenarios.
Log File:	log_failed.html#s1-s1
Status:	6 critical test, 5 passed, 1 failed 6 test total, 5 passed, 1 failed
Start / End:	20110721 13:13:16.517 / 20110721 13:13:22.254
Elapsed:	00:00:05.737

Name	Documentation	Tags	Crit.	Status	Message	Start / Elapsed
Login Tests . Invalid Login . **Invalid Username**			yes	PASS		20110721 13:13:20.847 00:00:00.214
Login Tests . Invalid Login . **Invalid Password**			yes	PASS		20110721 13:13:21.062 00:00:00.188
Login Tests . Invalid Login . **Invalid Username And Password**			yes	PASS		20110721 13:13:21.251 00:00:00.195
Login Tests . Invalid Login . **Empty Username**			yes	PASS		20110721 13:13:21.447 00:00:00.236
Login Tests . Invalid Login . **Empty Password**			yes	FAIL	Location should have been 'http://localhost:7272/html/error.html' but was 'http://localhost:7272/html/welcome.html'	20110721 13:13:21.683 00:00:00.212
Login Tests . Invalid Login . **Empty Username And Password**			yes	PASS		20110721 13:13:21.896 00:00:00.254

Figura 2.4: Relatório RF – Vista geral

No caso de um teste chumbado, o relatório mostra facilmente qual o passo falhado, como demonstrado na figura 2.5 [17].

- TEST CASE: **Empty Password**
  - **Full Name:** Login Tests.Invalid Login.Empty Password
  - **Start / End / Elapsed:** 20110721 13:13:21.683 / 20110721 13:13:21.895 / 00:00:00.212
  - **Status:** FAIL (critical)
  - **Message:** Location should have been 'http://localhost:7272/html/error.html' but was 'http://localhost:7272/html/welcome.html'
  - + SETUP: **html_resource.Go To Login Page**
  - - KEYWORD: **Login With Invalid Credentials Should Fail** ${VALID USER}, ${EMPTY}
    - **Start / End / Elapsed:** 20110721 13:13:21.757 / 20110721 13:13:21.895 / 00:00:00.138
    - + KEYWORD: **html_resource.Input Username** ${username}
    - + KEYWORD: **html_resource.Input Password** ${password}
    - + KEYWORD: **html_resource.Submit Credentials**
    - - KEYWORD: **html_resource.Login Should Have Failed**
      - **Start / End / Elapsed:** 20110721 13:13:21.880 / 20110721 13:13:21.894 / 00:00:00.014
      - - KEYWORD: **SeleniumLibrary.Location Should Be** ${ERROR URL}
        - **Documentation:** Verifies that current URL is exactly `url`.
        - **Start / End / Elapsed:** 20110721 13:13:21.881 / 20110721 13:13:21.894 / 00:00:00.013
        - 13:13:21.894 FAIL Location should have been 'http://localhost:7272/html/error.html' but was 'http://localhost:7272/html/welcome.html'

Figura 2.5: Relatório RF – Detalhe sobre caso de teste

De seguida, é apresentada uma breve explicação da estrutura de um *script* de testes no RF, assim como um exemplo prático (figuras 2.6 e 2.7 [23]):

- *settings*: Bibliotecas usadas, recursos, configurações de testes, entre outros. Na figura 2.6, são definidas ações que ocorrem antes, *Suite Setup*, e depois, *Suite Teardown*,

do conjunto de testes ser executado. De forma idêntica, existe o *Test Setup*, que é executado antes de cada teste. Neste caso também está presente um *Test Template*, que define a estrutura para os casos de teste apresentados, sendo que o que varia são os valores de *input* como o *User Name* e *Password*. Para além disso, é indicada uma biblioteca criada pelo utilizador, "*resource.txt*", representada na figura 2.7 que, por sua vez, utiliza a biblioteca externa "*SeleniumLibrary*";

- *variables*: Variáveis definidas para o documento. As variáveis, definidas na figura 2.7, são valores comuns a serem usados por qualquer *keyword*;

- *test cases*: Descrição dos cenários de teste. No exemplo apresentado, é possível observar, na figura 2.6, o conjunto de casos de teste, assim como os seus argumentos e respetivos valores;

- *keywords*: Definição das *user keywords*. Na figura 2.6 encontram-se definidas as *keywords* de alto nível, concretamente, o *template* "*Login With Invalid Credentials Should Fail*" e a *keyword* "*Login Should Have Failed*" que é usada como pós-condição. A seguir, na figura 2.7, é definida um conjunto de *user keywords* de mais baixo nível do que as anteriores.

```
*** Settings ***
Suite Setup       Open Browser To Login Page
Suite Teardown    Close Browser
Test Setup        Go To Login Page
Test Template     Login With Invalid Credentials Should Fail
Resource          resource.txt

*** Test Cases ***                 User Name        Password
Invalid Username                   invalid          ${VALID PASSWORD}
Invalid Password                   ${VALID USER}    invalid
Invalid Username And Password      invalid          whatever
Empty Username                     ${EMPTY}         ${VALID PASSWORD}
Empty Password                     ${VALID USER}    ${EMPTY}
Empty Username And Password        ${EMPTY}         ${EMPTY}

*** Keywords ***
Login With Invalid Credentials Should Fail
    [Arguments]    ${username}    ${password}
    Input Username    ${username}
    Input Password    ${password}
    Submit Credentials
    Login Should Have Failed

Login Should Have Failed
    Location Should Be    ${ERROR URL}
    Title Should Be    Error Page
```

Figura 2.6: *Script* de teste – casos de teste e definição de *keywords*

```
*** Settings ***
Library           SeleniumLibrary

*** Variables ***
${SERVER}         localhost:7272
${BROWSER}        Firefox
${DELAY}          0
${VALID USER}     demo
${VALID PASSWORD}    mode
${LOGIN URL}      http://${SERVER}/
${WELCOME URL}    http://${SERVER}/welcome.html
${ERROR URL}      http://${SERVER}/error.html

*** Keywords ***
Open Browser To Login Page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Maximize Browser Window
    Set Selenium Speed    ${DELAY}
    Login Page Should Be Open

Login Page Should Be Open
    Title Should Be    Login Page

Go To Login Page
    Go To    ${LOGIN URL}
    Login Page Should Be Open

Input Username
    [Arguments]    ${username}
    Input Text    username_field    ${username}

Input Password
    [Arguments]    ${password}
    Input Text    password_field    ${password}

Submit Credentials
    Click Button    login_button

Welcome Page Should Be Open
    Location Should Be    ${WELCOME URL}
    Title Should Be    Welcome Page
```

Figura 2.7: *Script* de teste – definição de *keywords*

Como foi anteriormente referido na presente tese, a sintaxe escolhida para descrever o comportamento esperado do software não é interpretada pela *framework* pois esta é de sintaxe livre. O uso das palavras reservadas da sintaxe do *Gherkin* é exclusivamente para auxiliar a leitura e compreensão do código.

2.4 Testes

2.4.1 Teste funcional ou não-funcional

É possível agrupar os testes consoante as características do sistema que se pretendem testar. O teste pode avaliar as qualidades funcionais do sistema, tais como, a completude, a correção e a adequação. De igual modo, os requisitos não-funcionais também podem ser analisados, tais como, a fiabilidade do sistema, a eficiência da *performance*, a segurança, a compatibilidade e a usabilidade do sistema. Resumindo, testes funcionais verificam o comportamento do sistema do ponto de vista do utilizador e testes não-funcionais verificam os atributos de qualidade do sistema. Todo o sistema também deve ser verificado, de modo a garantir a completude do mesmo, tal como especificado [12].

Visto que testes funcionais consideram o comportamento do sistema, estes podem ser elaborados usando técnicas de *black-box*, explicitadas na secção 2.4.7.

2.4.2 Teste de confirmação ou regressão

É sempre benéfico avaliar se qualquer mudança ao sistema não corrompe o mesmo (*change-related testing*). Os testes que analisam estas mudanças visam expor se as alterações tomadas têm o impacto esperado no sistema. Neste âmbito, podem ser executados testes de confirmação, ou seja, os testes que falharam são executados novamente, depois da correção do defeito correspondente. Por outro lado, podem ser executados testes de regressão, ou seja, ao adicionar novas funcionalidades, são executados todos os testes, para confirmar que o comportamento esperado do sistema se mantém [12].

2.4.3 Teste manual ou automático

É possível distinguir dois tipos de tarefas executadas por engenheiros de software, tarefas *revenue* e tarefas *excise* [1]. Tarefas *excise* são todas aquelas que são inerentes à elaboração de código, por exemplo, compilação de classes *Java*. Contrariamente, tarefas *revenue* contribuem diretamente para a solução do problema, por exemplo, decidir quais os métodos que uma classe *Java* deve ter.

No âmbito de teste, a manutenção de *scripts*, a sua execução e comparação de resultados são tarefas *excise* e, como tal, uma boa escolha para se automatizar. Este tipo de tarefas consome regularmente um nível considerável de tempo ao profissional envolvido, o que levanta outros problemas, sendo a automatização uma solução para muitos deles.

O benefício mais óbvio será o tempo poupado ao engenheiro de software que, deste modo, poderá concentrar-se em tarefas que não possam ser automatizadas e que obrigatoriamente precisem de ação humana. Tarefas *excise* tendem a ser rotineiras, o que faz com que rapidamente deixem de ser motivantes de executar. Para além de monótonas, este tipo de tarefas, quando realizadas manualmente, facilitam a introdução de erros durante a execução, levando a mesma a ser heterogénea, o que resulta num défice na garantia da qualidade de software [1].

2.4.4 *Continuous testing*

Uma abordagem de *continuous testing* no desenvolvimento de software implica que o produto seja testado, de forma automatizada, com antecedência, frequentemente e por todas as componentes, de modo a obter *feedback* pertinente o quanto antes. Em *continuous testing*, o desenvolvedor tem *feedback* assim que modifica parte do código, pois qualquer alteração despoleta a execução dos testes automatizados. Isto pode ser aplicado em várias situações [13], por exemplo, apenas relativo ao código desenvolvido por uma pessoa, em *Integrated Development Environment* (IDE), ou à totalidade do código, em *Continuous Integration* (CI) – ver secção 2.4.5.

Com a antecedência da análise da qualidade do software, é possível adequar a direção do trabalho, nomeadamente, caso os testes indiquem uma boa qualidade, o desenvolvimento pode continuar com confiança, caso contrário, os eventuais defeitos são corrigidos atempadamente, o que resulta num aumento de produtividade.

De acordo com D. Saff e M. D. Ernst. [25], a diminuição de produtividade é diretamente proporcional ao tempo de vida de um erro. Quando o erro introduzido não é descoberto imediatamente, isto leva a que seja necessário um maior número de mudanças para a sua resolução. Dado o código já não estar "fresco" na memória do desenvolvedor, torna-se complexa a compreensão e correção desse código. Este ponto agrava-se quando é elaborado código novo com base no código erróneo.

Aplicando *continuous testing* no desenvolvimento, conclui-se que o uso de recursos importantes, como o tempo e a energia do desenvolvedor, é menor, como se comprovou num estudo [25]: "*participants using continuous testing were three times as likely to complete the task before the deadline*". Para além disto, o estudo também concluiu que os participantes que utilizaram uma ferramenta de *continuous testing* tinham três vezes mais probabilidade de completar a tarefa corretamente comparativamente àqueles que não tinham suporte de *continuous testing*.

2.4.5 *Continuous integration*

A integração contínua – *Continuous Integration* (CI) – consiste na fusão regular de código produzido ou alterado pelos desenvolvedores. Todas as ações necessárias ao software, como a compilação, o *deployment* e os testes, devem integrar um processo único, automático e cíclico [10].

Este processo automático, que inclui vários tipos de testes ao software, deve ser executado o mais frequentemente possível, consoante as prioridades da empresa ou produto, possibilitando assim a deteção e correção de defeitos o mais rapidamente possível.

A integração contínua pode ser vista como um contínuo controlo de qualidade que, para além de ajudar a aumentar a qualidade do produto, reduz os ciclos de entrega, visto que o controlo de qualidade não é feito extensivamente apenas no fim de cada ciclo mas sim continuamente [10].

Continuous integration deve incorporar *continuous testing* no seu processo, o *feedback* atempado dado por *continuous testing* é essencial no *pipeline* de CI.

2.4.6 Testes de aceitação

Existem vários níveis de testes relativamente ao software de teste (*testware*) produzido: testes de componentes, testes de integração, testes de sistema e testes de aceitação. Estes níveis de teste estão relacionados com a etapa de desenvolvimento do sistema e, portanto, cada um tem objetivos específicos [10, 12]:

- Testes de componentes – foco em componentes que podem ser testadas independentemente, por exemplo, estruturas de dados, classes, módulos de base de dados;
- Testes de integração – foco nas interações entre componentes, sejam elas bases de dados, interfaces, micro-serviços;
- Testes de sistema – foco no comportamento do sistema como um todo, avaliando tanto a vertente funcional como a não-funcional;
- Testes de aceitação – foco no comportamento do sistema como um todo, relativamente às expetativas do cliente.

Os testes elaborados para a presente dissertação enquadram-se no último nível – testes de aceitação – portanto, esta secção dedica-se a aprofundar uma explicação sobre os mesmos.

O principal objetivo de testes de aceitação é confirmar que o sistema funciona como esperado, satisfazendo os requisitos definidos pelo cliente. Esta análise tem foco no comportamento e capacidades do sistema completo, sendo o ambiente ideal de testes o mesmo do que em produção. Qualquer teste de aceitação deve [12]:

- Instituir confiança na qualidade do produto como um todo;
- Validar a completude e expetativa das funcionalidades do sistema;
- Verificar que o comportamento funcional e o comportamento não-funcional encontram-se dentro do especificado.

O objetivo não é encontrar defeitos, mas sim avaliar a adequação/capacidade do sistema ao ser usado pelo utilizador final em ambiente operacional, real ou simulado. O sistema deve ser utilizado com o mínimo de esforço, custo e risco. É relevante perceber que um nível de teste pode integrar todos os tipos de testes descritos anteriormente. Por exemplo, supondo um sistema bancário, para o nível concreto de aceitação podem existir [12]:

- Testes funcionais – Podem ser testes desenhados com base em como o banqueiro lida com a aprovação ou recusa de um pedido de crédito;

- Testes não-funcionais – Podem ser testes de usabilidade, desenhados para avaliar a acessibilidade da interface de processamento de crédito do banqueiro para pessoas com incapacidades.

A especificação dos requisitos do sistema não é prova de que esses requisitos estão implementados nem que funcionam como esperado. Aliás, um estudo comprovou que as especificações iniciais dos requisitos estão apenas 15% completas e 7% corretas, sendo que não era eficiente a nível de custos completar ou corrigir as mesmas [21]. Visto que testes de aceitação são validados diretamente, perante um resultado positivo, estes indicam que o sistema satisfaz o requisito de utilização documentado.

"Os testes de aceitação são um 'contrato' entre os desenvolvedores e o cliente. Preservar esses testes, executá-los com frequência e corrigi-los conforme os requisitos mudam, prova que não houve quebra do contrato." [21]

Em suma, os testes de aceitação representam os interesses do cliente, dando confiança que o software faz o que é esperado de maneira correta. Isto é de extrema importância, pois um sistema que, embora funcione corretamente, não tenha as funcionalidades pretendidas pelo cliente, é completamente inútil [21].

2.4.7 Técnicas de testes *black-box*, *white-box* e *experience-based*

A técnica utilizada por um teste pode ser dividida em três categorias: *black-box*, *white-box* e *experience-based* [12].

A técnica de testes *black-box* analisa apenas a relação entre *inputs* e *outputs* do objeto sob teste. Deste modo, como não é exposta a estrutura interna, considera-se que a técnica estuda a vertente comportamental, podendo ser utilizada em testes funcionais ou não-funcionais.

A técnica de testes *white-box* analisa a arquitetura interna do objeto de teste, ou seja, estuda a vertente estrutural do mesmo e implica o acesso ao código implementado.

A técnica de teste *experience-based* apoia-se no conhecimento existente do desenvolvedor ou *tester* para elaborar e executar os casos de teste. As técnicas consideradas *experience-based* podem conter também técnicas *white-box* e *black-box*.

2.4.8 Resumo

Com base na categorização de testes apresentados nesta secção, a dissertação tem foco em testes no nível de aceitação, cujo objetivo é avaliar a não-regressão do sistema pela perspetiva funcional do mesmo.

Na presente dissertação foram elaborados testes no nível de aceitação, pois são os que melhor se enquadram na fase de vida do produto – manutenção. Os testes são descritos numa linguagem semelhante à linguagem natural para que a comunicação com o cliente seja eficaz, ou seja, os testes definem os critérios de aceitação pretendidos pelo cliente [9].

Apesar de não serem introduzidas novas funcionalidades no produto, dada a maturidade do mesmo, os testes de regressão possibilitam garantir que nenhuma correção ou melhoria ao software danifica as funcionalidades estáveis.

Os testes incidem sobre a funcionalidade da *Application Programming Interface* (API), ou seja, pretende-se avaliar o comportamento da mesma através das respostas dadas a um determinado pedido. Embora seja possível aplicar técnicas *black-box* ou *white-box* [20], foi escolhida a primeira, visto que foram analisados os pedidos (*inputs*) e respostas (*outputs*) e não o código fonte.

Com base na caraterização dos testes pretendidos, a automatização é um caminho viável e que pode trazer benefícios, comparativamente a uma execução manual. Para além dos benefícios inerentes a execução automática, esta transição é crucial num ambiente de *continuous integration*, que se pretende atingir.

Capítulo 3

Trabalho relacionado

Neste capítulo são apresentados projetos semelhantes ao projeto APIS8. O primeiro projeto exposto encontra-se a ser desenvolvido pela *Thales Group Portugal*; o segundo apresenta um relatório feito sobre um conjunto de projetos que utilizaram testes automatizados; o terceiro fala sobre a utilização do RF na automatização de testes funcionais de regressão; o quarto aborda concretamente testes de aceitação automatizados; o quinto, e último, demonstra alguns problemas e soluções encontradas na transição para um desenvolvimento *Agile*.

3.1 Produto GDP

A Thales desenvolve múltiplos projetos e, como tal, é benéfico existir um conjunto de normas de trabalho e *frameworks* comuns, especialmente dentro da *Thales Group Portugal*. Dado isto, existe um produto, o *Ground Transportation Systems – Digital Platform* (GDP), que já passou pela fase de transição para testes automatizados em que o APIS se encontra, usando exatamente as mesmas metodologias e *frameworks*. Com uma análise dos resultados obtidos no GDP depois de implementar a automatização de testes, é esperado atingir resultados semelhantes no APIS ou até mesmo resultados superiores, caso haja oportunidade para se tomar melhores decisões nesta transição para automatização.

De modo sucinto, o GDP é uma plataforma que permite desenvolver e disponibilizar (*deploy*) aplicações modernas, robustas, seguras e escaláveis. Para isto, a plataforma providencia um conjunto de componentes reutilizáveis para a construção da aplicação assim como mecanismos modernos para gerir aplicações disponibilizadas (*deployed*). Em acréscimo, o GDP especifica múltiplas metodologias e tecnologias modernas, tal como especifica boas práticas e manuais de instruções.

Este produto introduziu a automatização de testes de raiz no início de janeiro de 2020

e, apesar do investimento inicial para a transição de testes manuais para automáticos, passados dois meses já era possível ver resultados bastante positivos. Em concreto, dos quase 900 testes descritos (843) sobre as múltiplas áreas da plataforma, 28% foram automatizados (233) e os restantes 72% eram executados manualmente (610). Pode não ser intuitivo o impacto gerado pela automatização de testes, visto ser uma percentagem menor, contudo os benefícios já eram visíveis relativamente aos recursos poupados. Concretamente, a duração da execução desses 233 testes automatizados demorava cerca de 40 minutos, um resultado que pode parecer pouco impressionante mas que seria impossível de obter manualmente. Ao analisar em mais detalhe, concluímos que a execução manual dos mesmos testes demoraria quase 6 dias completos, contando com o esforço a tempo inteiro de um *tester*.

Para além da drástica diminuição de tempo investido, os quase 40 minutos de execução automática não necessitam da supervisão de qualquer *tester*, por isso pode-se afirmar que os testes manuais poupam, a longo prazo, cerca de 35 minutos por cada teste descrito [7 Fevereiro 2020 – *IVVQ Team Automation Release Notes CW6*].

Finalmente, pode-se reparar numa tendência para se produzirem mais testes em cada semana, isto porque a escrita dos testes é um trabalho lento inicialmente mas, à medida que se vão construindo *keywords* mais complexas e bibliotecas pertinentes, a escrita dos testes é cada vez mais intuitiva. Esta aceleração de produção estima-se ser mais acentuada no momento inicial da transição, não só pelo descrito acima, mas devendo-se também ao facto desta equipa não ter anteriormente um contexto profissional de automatização, logo também se considera a curva de aprendizagem dos profissionais.

Atualmente, passados cerca de onze meses desde a transição para a automatização, o produto conta com 1784 testes descritos, sendo que 962 são executados manualmente (54%) e 822 são executados automaticamente (46%). Analisamos que, não só a percentagem de testes automatizados aumentou como também o volume total de testes, o que leva a crer que a análise feita para a escrita dos casos de testes automatizados expôs novos casos de testes pertinentes. A cada ciclo de entrega, cujo período é de três meses, cria-se entre 200 a 250 novos cenários.

3.2 Estudo sobre testes automatizados

Foi elaborado um relatório em 2005 [4], no âmbito de *software testing* e automatização de testes, relativo às observações de doze projetos de software num período de três anos. Os autores são profissionais envolvidos nesses mesmos projetos, sendo que, não só analisaram os erros dos seus colegas, como os seus próprios.

Ao longo do tempo formaram múltiplas conclusões, sendo as mais relevantes apresentadas e discutidas de seguida:

1. A repetição de testes é mais comum do que o suposto;

2. A aptidão para executar testes automatizados diminui se não for feita regularmente;

3. A substituição total de testes manuais por automatizados não é possível;

4. A manutenção de software de teste (*testware*) é complicada;

3.2.1 Repetição de testes

Dado o investimento inicial em automatizar testes, é considerado que cada vez que um teste é executado sem intervenção manual, teoricamente, o investimento é amortizado. Tendo isto em conta, é benéfico que a quantidade de execuções do teste seja elevada, porém, segundo o estudo, esse valor é grandemente subestimado. Apesar disso, a principal causa do abandono da automatização, não é a falta de repetição de cada teste, mas sim a estratégia aplicada ou a arquitetura do código. Dadas as observações feitas, a maioria das automatizações são executadas entre cinco a dez vezes e, pelo menos, 25% das automatizações são executadas muito mais do que vinte vezes. O esforço de automatizar um teste manual é cerca do dobro do esforço de o executar, ou o triplo, para testes mais complexos de automatizar, como testes de interface. Contudo, observou-se que essa relação aumenta com o número total de testes automatizados. Isto é, torna-se mais difícil executar um teste manual quando existem mais testes. Pode-se explicar este fenómeno uma vez que, com mais testes, torna-se mais complexo manter a consistência entre todos eles.

Em concreto, verificou-se num projeto que o número esperado de execuções de testes manuais era oito vezes menor do que o real. Isto resultou com que uma avaliação de regressão ao software, com cerca de 90 casos de teste, durasse uma semana, sendo executada por três profissionais. Para além disso, como referido anteriormente, verificou-se uma subida na relação entre o esforço de automatizar e o de executar manualmente, representado na figura 3.1.

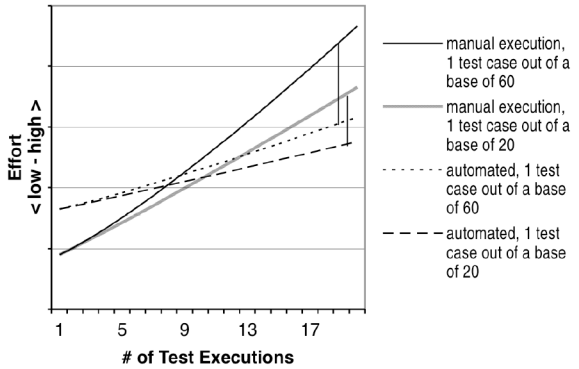


Figura 3.1: Relação entre o esforço e a quantidade de execuções

3.2.2 Aptidão para executar testes automatizados

Muitas vezes a automatização de testes falha, pois, independentemente da razão, os testes não são executados com suficiente frequência. Isto leva a que o teste atinja um estado inconsistente e de difícil compreensão, cujo custo de correção pode não se justificar. É

relatado que o custo de execução e manutenção de um teste automatizado aumenta de forma desproporcional ao longo do tempo que não é executado.

Pode-se repartir este padrão em quatro fases ao longo de um projeto, exposto na figura 3.2. Na primeira fase, os testes são fáceis de automatizar e são úteis, não só para detetar erros iniciais, como para definir o escopo do software. Na fase seguinte, a automatização é feita com maior facilidade devido a uma melhor compreensão do software desenvolvido. Isto leva a que os erros introduzidos sejam detetados nesta fase de escrita dos testes e não na sua execução, dando a sensação aos profissionais de que os testes são desnecessários e levando a uma falta de manutenção dos mesmos. Na terceira fase, volta-se a sentir a necessidade de avaliar a qualidade do software, contudo, nessa fase, a compreensão dos testes automatizados é baixa e a probabilidade de não ser viável investir novamente na automatização é alta. Na fase final, no pior dos casos, a capacidade de automatização perdeu-se por completo, embora seja nesta fase que os testes possuem maior importância, devido à necessidade de se testar a integração entre componentes, o *deploy*, entre outros.

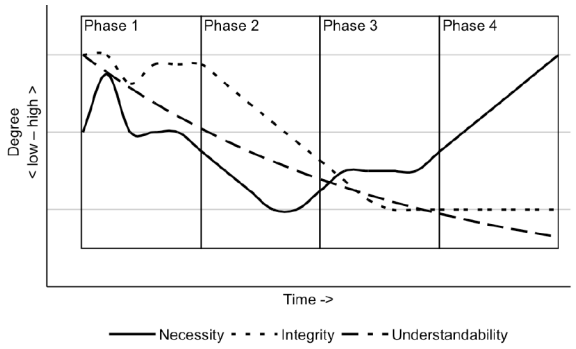


Figura 3.2: Fases de falha da automatização

3.2.3 Substituição total de testes manuais

Neste estudo foi corroborada a ideia de que testes manuais detetam a maioria dos defeitos, enquanto que em testes automatizados, entre 60% a 80% dos defeitos são detetados durante a elaboração da automatização e não na sua execução. Assim, na execução da automatização, não são detetados erros novos mas sim a introdução de erros semelhantes aos detetados anteriormente, o que leva a que certos testes manuais mantenham a sua importância. Dado que o *tester* não tem de correr testes de modo manual, é conquistado tempo para desenvolver testes automatizados mais complexos. O *loop* entre desenvolver um teste automatizado, executá-lo e analisar a qualidade do mesmo permite ao *tester* estar mais envolvido na tarefa, criando melhores casos de teste, com maior qualidade e menor esforço.

Concretamente num projeto, determinou-se a cobertura de cinco casos de testes, a nível de *branch* e de *statement*, em oito fases onde a complexidade foi aumentando. Na figura 3.3 pode-se confirmar que, conforme os testes foram mais trabalhados, aumentaram os níveis de cobertura para ambas as componentes.

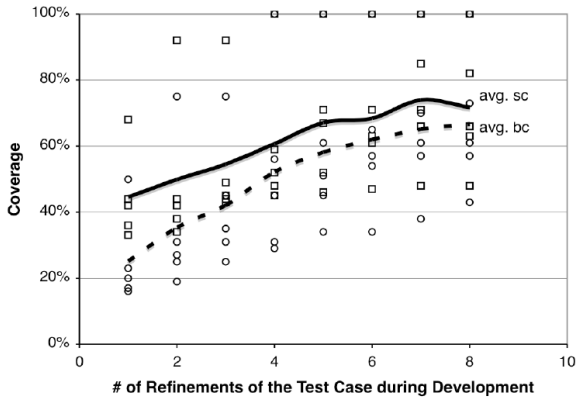


Figura 3.3: Evolução dos testes durante o seu refinamento. *statement cover*(sc), *branch cover*(bc)

3.2.4 Manutenção de software de teste

Um software de teste, denominado de *testware*, deve ter as mesmas garantias de qualidade como qualquer outro software. Contudo, observou-se que isso raramente acontece, a negligência acontece a vários níveis, como por exemplo, na arquitetura e respetiva documentação, fraca reutilização de componentes, estruturação do código não intuitiva, inexistência de testes, entre outros.

Concretamente, mesmo quando apenas um destes componentes é tido em maior consideração no *testware* é evidente uma melhoria. Num projeto verificou-se que, com uma abordagem de reutilização de código, perante um conjunto de cinquenta casos de teste, o esforço inicial de automatização é de seis horas, mas diminui abruptamente para uma hora apenas para cada teste. Isto contrasta evidentemente com um esforço de quatro horas para cada teste, anteriormente à reutilização, como pode ser notado na figura 3.4.

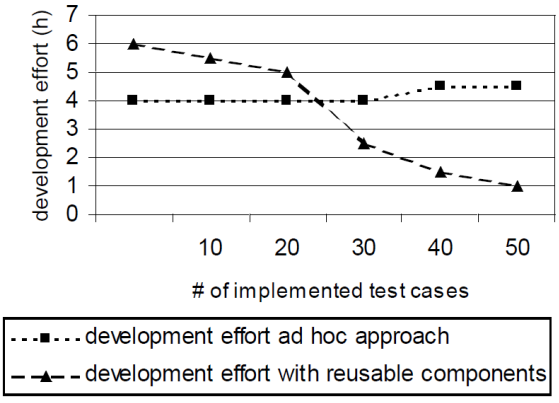


Figura 3.4: Comparação do esforço de automatização com e sem reutilização

3.2.5 Comparação

Dado que não foram realizados quaisquer testes manuais ao *backend* do APIS8, a avaliação errónea do nível de repetição dos testes manuais não se aplica. Todos os novos casos de teste foram logo automatizados e executados diariamente.

Visto que a automatização de testes do APIS8 só foi introduzida depois do desenvolvimento do mesmo, estes testes não sofreram um abandono como verificado neste estudo, pois nunca foram criados. A manutenção dos testes será considerada fácil, visto que o sistema já se encontra estável e os mesmos são executados diariamente.

Os casos de teste desenvolvidos no APIS8 – todos automatizados – não acompanharam o desenvolvimento do produto e, como tal, o seu nível de refinamento é inferior caso isso tivesse acontecido.

A escolha do *Robot Framework* como *framework* de automatização prevê-se que facilite a manutenção dos testes. Para além disto, a abordagem BDD aumenta a legibilidade dos testes e possibilita que mais profissionais tenham capacidades para manter os testes.

3.3 Utilização do *Robot Framework* para automatização de testes funcionais de regressão

Este trabalho elabora uma análise de como o RF pode ser utilizado na automatização de testes funcionais de regressão já existentes [27]. Os autores verificaram que o volume de testes necessário para manter um determinado nível de qualidade do sistema está relacionado com a complexidade do mesmo. Deste modo, para assegurar a qualidade desejada, é imperativo integrar o processo de automatização de testes no processo de desenvolvimento e manutenção do sistema. Sobre o produto em foco existiam já centenas de casos de testes relativos à integração, o que é considerado um volume elevado para se executar manualmente. Para além disto, a procura de erros nos relatórios gerados é algo trabalhoso. Tendo em conta estes dois aspetos, a execução e verificação dos casos de testes é um trabalho dado a erros, considerando também o nível de repetição dos testes. De seguida, são enunciados os três aspetos a ter em conta na escolha da ferramenta, assim como a razão do RF incluir todas estas características:

- Tipo de licença;
- Adequação para testes *black-box*;
- Independência de plataforma.

Para o primeiro ponto houve preferência por uma ferramenta *open-source*, uma vez que o investimento monetário normalmente é um dos fatores mais importantes. O RF tem uma licença que permite o uso grátis e, portanto, verifica a primeira característica necessária.

A maioria das ferramentas para automatização de testes tem como utilizador alvo desenvolvedores que estão familiarizados com diversas linguagens de programação. Contrariamente, o RF pode ser utilizado por alguém sem *background* de programação sendo que é *keyword-driven* – os testes têm potencial para serem bastante descritivos e utilizarem linguagem natural. Deste modo, o RF adequa-se a testes do tipo *black-box* pois um utilizador não tem de compreender o funcionamento interno do sistema ao nível de um desenvolvedor, mas sim num nível mais alto de abstração.

Por último, considerando o conjunto de ferramentas *open-source* para o tipo de utilização pretendida até à data de escrita do artigo, o RF era das poucas que suportam várias plataformas. Adicionalmente, a ferramenta tem manutenção regular.

De seguida, são expostos os benefícios ao transitar para a automatização, tendo em conta a utilização do RF. O primeiro ponto referido no artigo é a maior velocidade de execução dos testes, que está diretamente relacionado com a velocidade a que se detetam defeitos. Esta deteção é mais rápida, não só pelos testes correrem a maior velocidade, mas também pela fácil análise dos relatórios gerados pela ferramenta. Neste caso, notou-se especial melhoria, pois a análise de resultados de testes manuais recorria a vários *logs* dos diferentes componentes sob teste. Adicionalmente, a frequência com que se correm testes também aumenta, visto não serem necessários tantos recursos para correr um teste automatizado.

Ao analisar os tempos apresentados no artigo, relativamente ao tempo de preparação, automatização e de uma execução de um caso de teste, quer em modo manual quer em modo automatizado, foi elaborada a tabela 3.1.

Tabela 3.1: Tempos de testes manuais e automatizados

	Preparação e automatização	Ciclo de execução	Primeira execução	*Breakeven*
Manual	8:00h	0:10h (2% manual)	8:10h (100% manual)	120/10 minutos = 12 execuções
Automático	08:00h + 02:00h (20% para automatizar)	0:03h (0,5% automático)	10:03h (99,5% manual)	

Para um teste manual, 100% do trabalho é feito manualmente e a sua execução (10 minutos) corresponde a cerca de 2% do tempo total; Para um teste automatizado, foram investidas duas horas na escrita do teste, ou seja, cerca de 20% do tempo de trabalho total e o tempo de execução é, aproximadamente, apenas 0,5% da totalidade. Com estes dados percebemos que o benefício vem com um volume maior de execuções dos casos de teste. Concretamente, como pode ser observado na tabela 3.1, é benéfico automatizar o caso de teste em questão se o mesmo for executado manualmente mais de 12 vezes – demoraria 120 minutos –, sendo que este tempo corresponde ao tempo de automatização.

Ao considerar que uma suíte de testes é composta por 100 casos de teste e que a mesma é executada 20 vezes, contabilizando o *overhead* necessário para automatização

dos testes, concluímos que os tempos investidos não são assim tão distintos – diferença de 33 horas e 20 minutos – embora seja um resultado bastante positivo.

Mas a realidade é que, no modo manual, o *tester* está envolvido durante todo o processo, enquanto que em modo automático o *tester* está consideravelmente menos envolvido, neste caso, ao fim de 20 execuções de uma suíte composta por 100 casos, o *tester* poupou 133:20 horas – cerca de três semanas de trabalho (3 semanas, 1 dia, 5 horas e 20 minutos).

Em conclusão, os autores consideram que o uso do RF beneficia a escrita dos casos de teste, sendo que a estrutura e forma de escrita dos casos de teste se assemelha à linguagem natural. A existência de várias bibliotecas externas e a possibilidade de criar bibliotecas confere ao RF bastante flexibilidade, especialmente por ser acessível a profissionais não formados na área de informática. A escrita e leitura dos casos de teste é acessível a esses profissionais, mas também os relatórios gerados pelo RF são legíveis por qualquer profissional. O RF foi uma *framework* que possibilitou a automatização e consequentemente auxiliou a integração contínua do produto. Os autores consideram que a transição para a automatização pode ser feita em curto espaço de tempo e de modo eficiente pois, à medida que se aumenta o volume de execuções, a fração de tempo relativa ao esforço manual de um profissional é cada vez menor, contrariamente a uma execução manual dos testes.

3.3.1 Comparação

Durante a utilização do RF nesta dissertação comprovaram-se alguns destes pontos. A utilização de *keywords* facilitou a escrita dos casos de testes e definição de *keywords* de utilizador, visto ser semelhante à linguagem natural. A existência de bibliotecas externas minimizou a criação de *keywords* de bibliotecas de utilizador.

Como os casos de testes de *backend* para o APIS8 foram desenvolvidos para se automatizarem imediatamente, a execução manual de uma bateria de testes de *backend* previamente à automatização nunca ocorreu. Como tal, relativamente à maior facilidade em detetar os defeitos nos relatórios face ao sistema, neste caso, nada se pode concluir. Porém, ao longo do desenvolvimento das suítes de teste, a análise dos relatórios gerados foi suficiente para detetar eficazmente qualquer erro cometido.

3.4 Estudo sobre testes de aceitação automatizados

O trabalho estudado efetua uma revisão da literatura e apresenta um caso de estudo da indústria, onde foram utilizados testes de aceitação automatizados [7]. A premissa apresentada no artigo correspondente é que realizar testes de aceitação manualmente é uma tarefa monótona, cara e morosa. Portanto, com a abordagem *Automated Acceptance Testing* (AAT), o funcionamento do sistema encontra-se documentado, num formato que pode ser automatizado e, consequentemente, pode ser testado de forma cíclica. Nesta dissertação, o foco será no caso de estudo, que se organiza em três secções. Primeiro,

como é que a automatização de testes de aceitação foi elaborada; segundo, quais os efeitos e resultados obtidos; terceiro, qual a experiência de utilização, expondo aspetos positivos e negativos.

3.4.1 Elaboração dos testes

- Quem escreveu – neste caso de estudo, os testes foram escritos pelos desenvolvedores e não por *testers* dedicados. A seleção dos testes partiu do conhecimento dos desenvolvedores, que consideram ter melhor conhecimento dos requisitos necessários do produto do que os próprios clientes. Portanto, a escrita dos testes ocorreu paralelamente ao desenvolvimento das funcionalidades, sendo que o trabalho foi todo levado a cabo pelos desenvolvedores;

- Como foram escritos – no início da aplicação dos testes de aceitação, os desenvolvedores optaram por definir tantos testes quanto conseguiam, contudo, esta abordagem alterou-se. Os testes definidos passaram a ser apenas para funcionalidades mais complexas;

- Quando foram escritos – não existiu uma definição da fase de desenvolvimento em que os testes seriam elaborados, a sua elaboração veio quando necessário;

- Treino e introdução – a introdução da prática de testes automatizados de aceitação ocorreu através de um *workshop* de um dia, realizada por um especialista externo;

- Testes positivos e negativos – foram poucos os testes que contemplam casos negativos, isto é, testes que evocam mensagens de erro propositadamente;

- Manutenção – Para além do desenvolvimento normal do código de teste, este também sofreu alterações quando o teste não representava o comportamento esperado. Verificou-se que, nos projetos onde os testes incidiam nas interfaces gráficas, as mudanças ao sistema implicavam mudanças na definição dos testes. Contrariamente, em projetos cujos testes incidiam em "*web-services*", as mudanças ao sistema não implicavam tanto a alteração dos testes. Notou-se que testes mais próximos dos serviços e componentes base do sistema eram mais estáveis e, por oposição, testes às interfaces gráficas eram mais suscetíveis a alterações, aumentando o custo de manutenção.

3.4.2 Efeitos e resultados obtidos

- Efeito na comunicação de equipa – o código coberto pelos testes foi considerado mais seguro para ser trabalhado em cooperação, ou seja, a existência dos testes forneceu um sentimento de segurança aos desenvolvedores para trabalharem num determinado aspeto do sistema. Adicionalmente, em alguns casos, quando um desenvolvedor verificava que tinha de alterar o código de um colega e que isto faria

com que um teste não passasse, o desenvolvedor tomava iniciativa para comunicar com o colega, de forma a aprofundar o seu conhecimento e tomar melhores decisões;

- Compreensão do sistema – como os testes apresentam com maior clareza o funcionamento esperado do sistema do que o código em si, os desenvolvedores concordam que conseguem compreender melhor o funcionamento do sistema. Especialmente quando os testes são elaborados com antecedência;
- Gestão de requisitos – os testes de aceitação não foram usados por parte do cliente para expressar requisitos. Os requisitos foram expressos em formas tradicionais e, posteriormente, traduzidos em testes;
- Controlo do processo – a utilização dos testes teve uma aparente contribuição para a visibilidade e controlo do processo SCRUM utilizado nos dois projetos;
- Satisfação do cliente – os desenvolvedores concordam que os testes são uma ferramenta importante na deteção de defeitos e, portanto, menos defeitos no sistema geralmente traduzem-se numa maior satisfação do cliente;
- Qualidade do produto – a qualidade aumenta com base no ponto anterior, mas também porque, ao usar os testes para avaliar a não-regressão do sistema, liberta tempo comparativamente à mesma avaliação de modo manual;
- Documentação – os casos de teste são um apoio à documentação, pois descrevem o comportamento esperado do sistema. Alguns desenvolvedores julgam que integrar os testes em documentação formal seja benéfico;
- Complexidade – alguns testes são de maior complexidade e, portanto, para manter a compreensão, são divididos em testes de menor escala.

3.4.3 Experiência

- Experiências positivas – o *feedback* obtido culmina em algumas opiniões, expressas nas seguintes afirmações: é mais seguro alterar código; existem menos erros introduzidos; é mais fácil partilhar capacidades devido à segurança aparente; é minimizada a necessidade de testes manuais; os testes são mais compreensivos e é relativamente fácil adicionar novos testes; a escrita dos testes implica reflexão sobre o comportamento do sistema que, consequentemente, facilita encontrar casos de teste "especiais";
- Frequência de deteção de falhas – caso o teste seja desenvolvido em paralelo com a funcionalidade, a deteção de um defeito é dada pouco tempo depois da elaboração do teste. Mudanças posteriores a uma funcionalidade, quando já existem testes correspondentes documentados, não geram tantos testes falhados como se supunha, isto porque o desenvolvedor utiliza o teste como guia e desenvolve o código de modo a que este não falhe;

- Descoberta de novos requisitos – a atenção extra dada ao sistema, aquando do desenvolvimento dos testes, pode detetar novos requisitos, para além disto, um teste falhado pode indicar falhas a nível de integração e não só de código;
- Experiências negativas – um teste por si só não fornece garantias, depende da qualidade do mesmo; o desenvolvimento e custo de manutenção dos testes deve ser tido em conta; o volume de um teste pode facilmente ser excessivo e é necessário prática para que isto não aconteça;
- Continuação de uso – todos os oito desenvolvedores envolvidos concordaram que este tipo de teste deve ser adotado em novos projetos. Contudo, a utilização dos testes será mais eficaz quanto mais cedo for iniciada no projeto, para além de ser crucial fazer uma avaliação de quais os testes com maior retorno de investimento.

Em conclusão, foi observado que para iniciar AAT basta uma formação simples. Os testes são como uma documentação viva sobre o comportamento do sistema. Esta abordagem possibilitou encontrar mais defeitos e mais cedo no desenvolvimento. A segurança dada pelos testes facilitou a cooperação no desenvolvimento de código. A escrita dos testes aumentou a compreensão do sistema. "Os testes ajudam a minimizar o medo ou resistência de alterar o código mesmo que isso seja benéfico no desenvolvimento e manutenção do mesmo". O custo de manutenção pode ser elevado, portanto é necessário efetuar uma boa escolha dos testes a automatizar. Os testes podem fornecer uma falsa sensação de segurança, embora o teste possa ter um resultado positivo, não quer dizer que a sua cobertura do código seja suficiente.

3.4.4 Comparação

Contrariamente ao descrito no estudo apresentado, os testes automatizados para o APIS8 foram elaborados por um *tester* sem apoio de algum desenvolvedor. Contudo, também decorreu um curto *workshop* sobre a *framework* de automatização e qual a estrutura dos casos de testes de aceitação.

O levantamento de requisitos do APIS8 também foi feito de forma tradicional, contudo, não foi possível fazer a tradução desses requisitos para os testes automatizados de *backend* devido à impossibilidade de acesso ao respetivo documento.

A seleção dos casos de teste, tal como no estudo, não apresenta nenhuma metodologia e a sua elaboração e automatização foram feitas apenas no fim do desenvolvimento do produto. Como não existe um mapeamento de requisitos para os casos de testes nem um método sistemático de seleção dos mesmos, de forma semelhante ao estudo apresentado, os casos de testes com resultados positivos podem fornecer uma falsa sensação de qualidade do produto.

Por fim, de forma semelhante ao apresentado, a complexidade dos casos de teste foi algo a ter em consideração, por vezes foi necessário reestruturar determinados casos de teste para melhorar a legibilidade.

3.5 Adoção de testes automatizados em projetos de desenvolvimento *Agile*

O estudo apresentado visa compreender os diferentes desafios presentes num ambiente *Agile* em transição para a automatização, assim como as respetivas possíveis soluções [31]. Ao longo do estudo foram anotados os principais desafios e soluções apresentadas por trinta e oito profissionais. Identificaram-se cinco desafios, expostos na tabela 3.2, sendo que quatro deles são enfrentados recorrendo a duas estratégias, e o quinto através de uma estratégia.

Tabela 3.2: Estratégias adotadas em diferentes projetos *Agile*

Desafios	Estratégias
Escolher a ferramenta certa	– Conhecer os requisitos de testes automatizados – Análise custo-benefício
Gerir o ambiente de testes	– Planeamento atempado – Virtualização
Manter *scripts* de teste	– *Framework* de automatização – *Page Object Model*
Mentalidade	– Fomentar consciência de automatização – Avaliação do ROI
Comunicação eficaz	– Abordagem "equipa única"

O primeiro desafio apresentado é a escolha da ferramenta de automatização indicada para o projeto e para a empresa. Isto pode ser endereçado por duas estratégias:

- Conhecer os requisitos da automatização dos testes – é importante compreender que não há uma ferramenta 100% indicada para todos os projetos e empresas, é necessária uma avaliação das necessidades específicas. Nomeadamente, é essencial perceber que tipo de testes são relevantes realizar para o projeto e a sua fase. Adicionalmente, é necessário compreender as capacidades da equipa, relativamente a linguagens de programação. Para além disto, é necessário entender os *trade-offs* ao obter uma ferramenta *open-source* contrariamente a uma comercial;

- Analisar o custo-benefício – o custo é um fator impossível não considerar. Como tal, investir numa ferramenta paga, implica compreender se o retorno se adequa às necessidades dos testes e do projeto em causa. Nomeadamente, se os testes precisam realmente de ser automatizados e se a utilização da ferramenta implica formação adicional.

O segundo desafio é gerir o ambiente de teste. Para os testes terem significado, num desenvolvimento *Agile* é importante que o ambiente de teste seja semelhante ao ambiente de produção. São apresentadas duas estratégias para minimizar este problema:

- Planeamento atempado – é normal existirem múltiplas configurações para múltiplos ambientes de teste e, como tal, a sua organização e planeamento atempado é crucial para que tudo possa correr como esperado;

- Virtualização – a utilização de máquinas virtuais possibilita a "criação" de espaço novo de teste, para que, tanto desenvolvedores como *testers*, consigam testar o sistema. "*Above all it is scalable and has on demand access which reduces our burden of managing test environment*".

Os testes dependem das funcionalidades e, como tal, alterações nas funcionalidades implicam a manutenção dos *scripts* de testes correspondentes. Foram identificados dois aspetos a considerar:

- *Framework* de automatização – a utilização de uma boa *framework* facilitou a manutenção dos *scripts* de teste. As equipas procuraram *frameworks* com as seguintes características: modular, *keyword-driven* e *behaviour-driven*;

- *Page Object Model* (POM) – a segunda abordagem que pode facilitar a manutenção dos *scripts* de teste é a utilização do *page object model*, que mapeia cada elemento da página *web* para uma classe de objeto.

O penúltimo desafio encontrado foi a mentalidade associada a automatização de testes. É importante que toda a equipa perceba que a automatização é um investimento de longo prazo e que transitar para um desenvolvimento *Agile*, concretamente, ao automatizar testes, implica uma curva de aprendizagem. Para isto podem ser adotadas duas estratégias:

- Fomentar consciência de automatização – para que a equipa não fique constantemente frustrada, é necessário compreender os riscos e recompensas da automatização de testes. Isto pode ser feito através de *workshops* ou formações.;

- Avaliação do ROI – a aplicação da fórmula ROI – *return on investment* –, tendo em conta o custo das ferramentas, o custo de trabalho e o tempo necessário para construir todo o suporte à automatização, fornece dados concretos sobre o tipo de investimento a ser feito.

Por último, um desafio presente por toda a parte é como ter uma comunicação eficaz. Os participantes concordam que uma má comunicação resultou num mau planeamento de automatização.

- Abordagem "equipa única" – foi necessário fomentar o sentimento de equipa, através de uma comunicação pró-ativa, tanto verbal como escrita, entre todos os membros da equipa, mas especialmente entre *testers* e desenvolvedores.

Sendo que a execução dos mais variados testes automatizados é regular, normalmente diária, esta atividade tem de estar aperfeiçoada. Isto é algo difícil de atingir, mas os pontos anteriormente mencionados visam facilitar o caminho para a perfeição.

3.5.1 Comparação

Dentro do ambiente da Thales, considera-se que se escolheu a *framework* de automatização com base em características fundamentadas, nomeadamente, o tipo de testes, o conhecimento dos profissionais e as diferentes utilizações por vários projetos. O tema é aprofundado no capítulo seguinte, em 4.3.1.

Não foi possível obter informação sobre o planeamento e gestão dos ambientes de testes aplicados ao APIS8.

Para facilitar a manutenção dos *scripts* de teste é utilizado o *Robot Framework* que contém as características apresentadas neste estudo – ver secção 2.3.2. A utilização do *page object model* não se aplica para os testes de *backend*.

A Thales é uma empresa que se encontra na transição para um desenvolvimento *Agile*. Isto é feito gradualmente e em projetos selecionados. Por exemplo, o aspeto da automatização foi aplicado na fase de manutenção do APIS8 mas espera-se que seja um elemento base no desenvolvimento da próxima versão.

Para alcançar uma melhor comunicação dentro do projeto, utilizaram-se técnicas do BDD na escrita dos casos de teste, facilitando a compreensão dos mesmos por todos os membros da equipa.

3.6 Resumo

Ao longo deste capítulo foram apresentados diversos estudos e aplicações concretas de conceitos também utilizados nos testes automatizados deste projeto. Foi feito um paralelismo entre o apresentado em cada um dos cinco trabalhos e o trabalho concreto levado a cabo durante a dissertação – a automatização de testes de *backend*. É possível concluir que muitos desafios e resultados apresentados são semelhantes ao exposto ao longo da tese. Deste modo, a aplicação da automatização a casos de testes referentes ao *backend* teve um desfecho esperado.

Capítulo 4

Processo de automatização

Neste capítulo, primeiro é determinado qual o objetivo pretendido com a elaboração dos testes automatizados de *backend*. De seguida, é feita uma breve análise das ferramentas escolhidas e os seus principais adversários. No terceiro passo, é descrito o conjunto de atividades que integram o processo e, por fim, são apresentadas em mais detalhe as principais atividades, com auxílio de um exemplo prático.

4.1 Objetivo

A presente dissertação tem como principal objetivo a elaboração de testes automatizados para *backend*, cujo desenvolvimento é orientado ao comportamento esperado do produto – *Behavior-Driven Development*. Concretamente, foi utilizada a *framework Robot Framework* (RF), onde foram desenvolvidos os casos de testes automatizados na linguagem *Gherkin*. A automatização de testes de *backend* gera pedidos REST e verifica se as respostas em formato JSON são as esperadas, assim como confronta essas respostas com conteúdo da base de dados, verificando a consistência de dados no *backend*. Nesta solução, os testes realizados são executados regularmente no servidor de automatização *Jenkins*.

Com a transição para a automatização espera-se minimizar o tempo de execução dos testes, criando a possibilidade de se executarem mais testes, assim como facilitar a elaboração de novos casos de testes automatizados, através da abordagem BDD.

É de salientar que nenhum projeto APIS, independentemente da versão, tem testes automatizados relativos ao *backend*. Deste modo, os testes são pioneiros para todos os projetos que têm este produto como base, sejam eles projetos já desenvolvidos e em manutenção ou em desenvolvimento. Adicionalmente, para a versão 8 do APIS, não existem documentos *Software Test Description* (STD) relevantes para *backend*, no sentido em que a maioria dos testes documentados para outros projetos não se possam aplicar ao projeto

de Doha, no qual a tese se insere. A futura produção de documentos STD mais robusta e útil pode ser auxiliada pelos testes automatizados, sendo que os casos de teste estão escritos de forma semelhante à linguagem natural e têm uma estrutura bem definida.

4.2 Protótipo

Tendo em conta que a estrutura para a realização de testes automatizados de *backend* e *frontend* é a mesma – mesma *framework*, mesma linguagem e mesmo nível de abstração – e tendo em conta a fase de desenvolvimento do produto dentro da empresa, a prova de conceito será feita relativa ao *frontend*. Deste modo, é apresentado um protótipo de um teste automatizado que interage com o site oficial dos Correios, Telégrafos e Telefones (CTT).

Supondo a seguinte *user story*, relativa à utilização do site dos CTT: como utilizador, eu quero navegar na página dos CTT, para encontrar a secção relativa a vantagens para empresas. Para executar esta ação, o utilizador terá de se encontrar na página inicial dos CTT e interagir com os elementos corretos do site numa determinada ordem, até chegar à secção do site que pretende.

Para traduzir esta *user story* num caso de teste automatizado, é necessário estruturar a ação, utilizando a sintaxe *Gherkin* (*Given*, *When* e *Then*) e utilizar *keywords* de utilizador com o nível de abstração correspondente à *user story*. Estas *keywords* de utilizador serão compostas por outras *keywords* de mais baixo nível.

A figura 4.1 representa um caso de teste automatizado, onde é simulada a ação no site oficial dos CTT. Neste caso de teste é possível observar:

- O título – apresentado a preto;
- As palavras reservadas para a sintaxe *Gherkin* – apresentadas a cinzento;
- As *keywords* de alto nível – apresentadas a azul.

```
T01-Vantagens empresas
    Given I am in the "Home Page"
    When Click when available on "Recibos Online"
    And Click when available on "Vantagens Empresas"
```

Figura 4.1: Caso de teste – protótipo

O primeiro passo apresenta a pré-condição, indicada pela sintaxe *Gherkin* – "*Given I am in the 'Home Page'*". De seguida, é apresentada a ação, dividida em dois passos – "*When Click when available on 'Recibos Online' And Click when available on 'Vantagens Empresas'*" – que representa a sequência de botões que devem ser clicados para se chegar à secção pretendida. Idealmente existiria no caso de teste uma pós-condição, indicada por "*Then*", que verificaria que de facto o utilizador se encontrava na secção pretendida. Isto poderia ser atingido verificando a existência de um elemento exclusivo àquela secção.

Cada uma destas *keywords* teve de ser definida por outras *keywords* com um nível de abstração inferior. A *keyword* da pré-condição, representada na figura 4.2, é composta por outras duas *keywords*, nomeadamente "*Title Should Be*" e "*Wait Until Element Is Visible*" que, em conjunto com os argumentos utilizados, verificam que o título do site naquele momento é "CTT" e, assim que possível, verificam se dado elemento existe na página.

```
I am in the "${name_page}"
    Title Should Be    CTT
    Wait Until Element Is Visible          ${initialPage['${name_page.lower().replace(" ", "_")}']}
```

Figura 4.2: Definição da *keyword* de utilizador presente na pré-condição

A primeira *keyword* de acção é composta por outras duas *keywords* de baixo nível, como representado na figura 4.3, a primeira "*Wait Until Element Is Visible*" e a segunda "*Click Element*" que, em conjunto com os argumentos correspondentes, garantem que dado elemento está presente no site e interagem com ele através de um clique.

```
Click when available on "${element}"
    [Documentation]  The keyword waits until the element is visible and clicks on it
        Wait Until Element Is Visible          ${initialPage['${element.lower().replace(" ", "_")}']}
        Click Element                          ${initialPage['${element.lower().replace(" ", "_")}']}
```

Figura 4.3: Definição da *keyword* de utilizador presente na ação

Foi necessário criar múltiplas camadas de abstração, neste caso só se necessitou de duas, cada uma com uma lógica de verificação mais concreta. As *keywords* de baixo nível pertencem a bibliotecas externas, sendo que a criação de novas bibliotecas não foi necessária. Para os testes de *backend* elaborados, existe a mesma estrutura e semelhante divisão dos níveis de abstração, mas os passos são referentes à interação com elementos do *backend*, como a API e a base de dados.

4.3 Análise de ferramentas

A área de *Quality Assurance* (QA) onde a tese se insere, concretamente a área de automatização de casos de testes, interage com diversas ferramentas. Esta secção analisa duas possíveis *frameworks* de automatização e duas possíveis ferramentas de integração contínua.

Para cada vertente, são apresentadas as opções, seguidas da avaliação das características consideradas essenciais para a *Thales* e terminando com o motivo da escolha da ferramenta utilizada. As características essenciais apresentadas para as ferramentas de automatização e de integração contínua foram facilitadas pelo IVVQ *Manager*, embora o mesmo não tenha participado na discussão e decisão das ferramentas.

4.3.1 *Framework* de automatização

Uma *framework* de automatização de testes é um conjunto de normas e regras com o intuito de estruturar todos os testes necessários para um dado software. A *framework*

selecionada para a elaboração de testes automatizados é o RF e pode-se considerar que o *Cucumber Open* seria uma alternativa plausível.

São brevemente apresentadas as duas opções para a *framework* de automatização, seguidas da exposição das características selecionadas, em paralelo com a comparação, e terminando com a conclusão para a escolha do RF.

4.3.1.1 *Robot Framework*

O RF é uma *framework* genérica de automatização para testes de aceitação, *acceptance test driven development* e *robotic process automation* [23]. Faz uso de uma sintaxe simples com *keywords* em linguagem natural, o que simplifica o seu uso. A quantidade de bibliotecas disponíveis e a possibilidade de criar novas, usando *Python* ou *Java*, fornece uma grande flexibilidade à *framework*. Dito isto, é possível salientar três aspetos principais do RF: clareza – tem uma arquitetura modular que permite facilmente a integração de novas bibliotecas e possibilita uma boa organização para os testes; facilidade – a utilização de *keywords* é bastante fácil, especialmente devido à linguagem natural que as mesmas possibilitam. Os relatórios gerados estão bem estruturados e, portanto, são bastante legíveis; comunidade – existem várias possibilidades de suporte, por exemplo, fóruns online, conferência anual *RoboCon* e encontros de comunidade.

Atualmente, a manutenção e crescimento do RF é suportado pela *Robot Framework Foundation* que agrega mais de quarenta entidades, que acreditam no trabalho proporcionado pelo RF [24].

Na figura 4.4, encontra-se um exemplo de uma suíte de testes, desenvolvida através do RF, com apenas um caso de teste – "*Valid Login*" – e, na figura 4.5, encontra-se a definição da *keyword* "*Submit Credentials*" usada nesse mesmo teste [23]. Visto que o RF pode ser usado em múltiplos editores de texto ou IDE's, a formatação dos testes pode sofrer ligeiras alterações de uma ferramenta para outra.

```
*** Settings ***
Documentation     A test suite with a single test for valid login. This test
has a workflow that is created using keywords in the imported resource file.
Resource          resource.txt

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username          demo
    Input Password          mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]              Close Browser
```

Figura 4.4: Exemplo de uma suíte de testes

```
*** Keywords ***

Submit Credentials
    Click Button    login_button
```

Figura 4.5: Exemplo da definição de uma *keyword*

4.3.1.2 *Cucumber Open*

O *Cucumber Open* é uma *framework open source* que suporta BDD, através da escrita e execução de testes de aceitação em *Gherkin*. Esta *framework* possibilita a escrita de testes em linguagem natural. Contrariamente ao RF, cada passo do caso de teste é considerado um "*Gherkin step*" e não uma *keyword*. A definição de um "*step*" implica sempre elaboração de código, quer seja em *Ruby*, *Java* ou *Javascript*.

Na listagem 4.1, encontra-se um exemplo do cenário "*Some cukes*", ou seja, um caso de teste. É apresentado apenas um passo, "*Given I have 48 cukes in my belly*", cuja definição é feita em *Java*, como ilustrado na listagem 4.2 [5].

Listagem 4.1: Cenário *Gherkin*

```
1 Scenario: Some cukes
2   Given I have 48 cukes in my belly
```

Listagem 4.2: Definição do *step* em *Java*

```
1  package com.example
2  import io.cucumber.java8.En
3
4  class StepDefinitions : En {
5
6      init {
7          Given("I␣have␣{int}␣cukes␣in␣my␣belly") { cukes: Int ->
8                  println("Cukes:␣$cukes")
9          }
10     }
11 }
```

4.3.1.3 Comparação

No ambiente da Thales Portugal, foi definido um conjunto de características importantes para a escolha desta *framework*. De seguida, são sucintamente comparadas as duas opções, em paralelo com a apresentação das características decisivas para a escolha de uma *framework* de automatização:

1. Amplitude – a Thales pretende ter uma *framework* transversal a todos os seus projetos e que possibilite tanto testes de *User Interface* (UI) como de *Application Programming Interface* (API). Deste modo, procurou uma *framework* com potencial, ou seja,

com suporte para construção de diferentes níveis de abstração e de novas bibliotecas;

Ambas as opções suportam técnicas de BDD, embora o *Cucumber Open* interprete a sintaxe *Gherkin* e o RF não. Ambas podem ser utilizadas para testar *User Interface* (UI) assim como API, através da adição de bibliotecas. Contudo, o RF já inclui algumas bibliotecas de teste. A tabela 4.1 apresenta o resumo desta comparação.

Tabela 4.1: *Robot Framework* contra *Cucumber Open* – Amplitude

	Robot Framework	*Cucumber Open*
Linguagem	*Python*	*Ruby*
Suporta técnicas BDD	Indiferente	Interpreta *Gherkin*
Testa UI assim como API	Sim	Sim
Bibliotecas incorporadas	Sim	Não

2. Curva de aprendizagem – dado que os constituintes da equipa de testes são maioritariamente profissionais sem *background* de desenvolvimento ou linguagens de programação, a Thales priorizou *frameworks* cuja linguagem fosse de mais alto nível assim como *keyword-driven*;

   Tanto o RF como o *Cucumber Open* possibilitam escrever testes focados no comportamento e possibilitam uma escrita de alto nível. Estes pontos facilitam a escrita dos testes, especialmente para o tipo de profissionais que integram as equipas de QA da empresa. A curva de aprendizagem para a escrita de testes é semelhante, contudo, o *Cucumber Open* tem maior foco em linguagens de programação, podendo ser algo que dificulte a compreensão da *framework*. A tabela 4.2 apresenta o resumo desta comparação.

   Tabela 4.2: *Robot Framework* contra *Cucumber Open* – Curva de aprendizagem

	Robot Framework	*Cucumber Open*
Utilização de *keywords*	Sim	Não
Foco no comportamento	Sim	Sim

3. Suporte e comunidade – a busca da Thales focou-se em *frameworks open source* e com custo nulo, como tal, todo o suporte vem da comunidade e por isso procuraram-se comunidades bastante ativas. Esta questão também beneficia o primeiro ponto, no sentido em que a comunidade deve participar com bibliotecas bastante variadas que satisfaçam as mais diversas necessidades.

   Tanto o *Cucumber Open* como o RF são totalmente *open source* e sem custos. A comunidade do RF é considerada ativa, o que pode colmatar o facto de não haver um sistema de suporte formal. A tabela 4.3 apresenta o resumo desta comparação.

Tabela 4.3: *Robot Framework* contra *Cucumber Open* – Suporte e comunidade

	Robot Framework	*Cucumber Open*
Open source	Sim	Sim
Suporte pela comunidade	Sim	-

4. *User Experience* (UX) – a usabilidade de um software é sempre tida em conta, neste caso, procuram-se *frameworks* que produzissem relatórios bastante visuais, para além de serem completos e detalhados.

   Os relatórios gerados pelo RF são legíveis e organizados, fornecem recursos importantes, como detalhe sobre os casos de testes, capturas de ecrã relevantes no caso de testes de UI, entre outros, tudo feito automaticamente. Os relatórios gerados pelo *Cucumber Open* possuem o mesmo tipo de informação, porém, pode-se considerar que têm um visual mais agradável. A tabela 4.4 apresenta o resumo desta comparação.

Tabela 4.4: *Robot Framework* contra *Cucumber Open* – UX

	Robot Framework	*Cucumber Open*
Produção de relatórios	Sim	Sim
Estética e usabilidade	Sim	Sim

4.3.2 Ferramenta de integração contínua

Existem vários tipos de ferramentas que auxiliam o processo de integração contínua, neste caso são analisados diferentes servidores de automatização. Este tipo de ferramenta é uma peça crucial para que um software tenha integração contínua, pois possibilita a execução de testes automáticos sem qualquer intervenção humana e de forma cíclica. O servidor de automatização escolhido foi o *Jenkins* e pode-se considerar que o *Bamboo* seria uma alternativa possível a essa ferramenta.

Nas duas secções seguintes ambas as ferramentas são brevemente apresentadas, seguidas da comparação entre elas, nas características selecionadas.

4.3.2.1 *Jenkins*

O *Jenkins* é dos principais servidores de automatização *open source*. O *Jenkins* foi desenvolvido com cinco aspetos principais em consideração: CI, fácil instalação, fácil configuração, diversidade de *plugins* e gestão de sistemas distribuídos [15]. Na figura 4.6 está presente um exemplo da interface do *Jenkins* [22]. No painel do lado direito encontram-se os projetos que o servidor tem de correr de forma cíclica, assim como o seu resultado de execução.

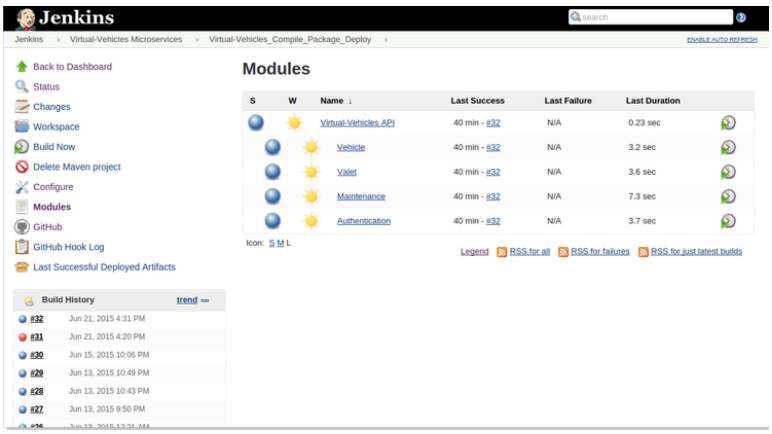


Figura 4.6: Exemplo da interface *Jenkins*

4.3.2.2 *Bamboo*

O *Bamboo* é um servidor que possibilita a execução automática de *builds*, testes e *releases* em conjunto. Considera-se a melhor escolha para ambientes de CI, *continuous deployment* ou *continuous delivery*. Foca-se em seis conceitos: testes em paralelo, agentes *Docker*, permissões, quarentena de testes, deteção de *branches* e *triggers* [2]. A figura 4.7 apresenta um exemplo de interface do *Bamboo* [28].

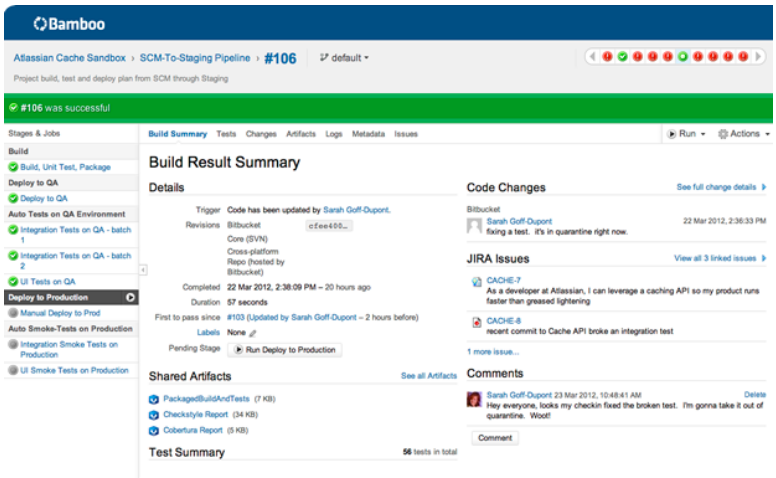


Figura 4.7: Exemplo da interface *Bamboo*

4.3.2.3 Comparação

Dentro da Thales Portugal foram definidas três características a considerar na escolha da ferramenta, sendo que a sua exposição será feita paralelamente com a avaliação do *Jenkins* e do *Bamboo*:

1. *Plugins* e integração – a existência de *plugins* que facilitem a integração com outras ferramentas é uma mais valia num ambiente de integração contínua.

Por um lado, existe o *Jenkins* com mais de mil e quinhentos *plugins*, oferecendo bastante flexibilidade ao utilizador, por outro lado, existe o *Bamboo* que já integra as plataformas *Jira* e *Git*, embora também existam centenas de *plugins* adicionais possíveis de comprar. A tabela 4.5 apresenta as principais integrações de cada uma das ferramentas.

Tabela 4.5: *Jenkins* contra *Bamboo* – *Plugins* e integração

	Jenkins	*Bamboo*
Integração *Jira*	Não	Sim
Integração *Bitbucket*	Não	Sim
Plugins	Sim	Sim

2. UX – a experiência de utilizador é algo que deve ser sempre tido em conta e, embora não se considere a interface do *Jenkins* desagradável, a interface do *Bamboo* tem uma aparência mais moderna e fluida.

Tabela 4.6: *Jenkins* contra *Bamboo* – UX

	Jenkins	*Bamboo*
Estética e Usabilidade	Sim	Sim

3. Suporte e comunidade – uma das preocupações da Thales, como de tantas outras empresas, é o investimento monetário e, como tal, também nesta área procurou-se uma ferramenta *open source* e que possuísse uma comunidade vasta e participativa, para facilitar o suporte.

   O *Jenkins* é uma ferramenta totalmente gratuita e, por conseguinte, o suporte é dado pela grande comunidade existente. O *Bamboo* é pago, para além de se pagar *plugins* adicionais, mas, por sua vez, facilita suporte profissional para os clientes, assim como documentação online e existência de fóruns de ajuda. A tabela 4.7 apresenta o resumo desta comparação.

Tabela 4.7: *Jenkins* contra *Bamboo* - Suporte e comunidade

	Jenkins	*Bamboo*
Open source	Sim	Não
Comercial	Não	Sim
Suporte pela comunidade	Sim	Sim
Suporte como serviço	Não	Sim

4.3.3 Escolha de ferramentas

Após esta análise, é percetível a semelhança das *frameworks* de automatização, contudo, a empresa decidiu que o RF era a *framework* que melhor se integrava no seu ambiente de trabalho. Embora seja normal que se tenham considerado várias nuances das ferramentas, o fator decisivo julga-se ter sido a existência de *keywords* do RF e a flexibilidade que elas trazem ao projeto de automatização. As *keywords* aumentam a legibilidade de todo o código de teste e não apenas os casos de teste de aceitação. Outro fator importante será a variedade de bibliotecas que oferecem as *keywords*, assim como a possibilidade de criar novas *keywords*, nos níveis de abstração desejados.

Relativamente aos servidores de automatização, nota-se também bastantes semelhanças nas características de interesse para a Thales. Deste modo, será lógico optar pela vertente com menor investimento monetário, o *Jenkins*, visto que trará o mesmo retorno que a sua alternativa.

4.4 Processo

De seguida, na figura 4.8, pode ser visto um diagrama representativo das atividades a serem realizadas durante a automatização de testes de um dado projeto.

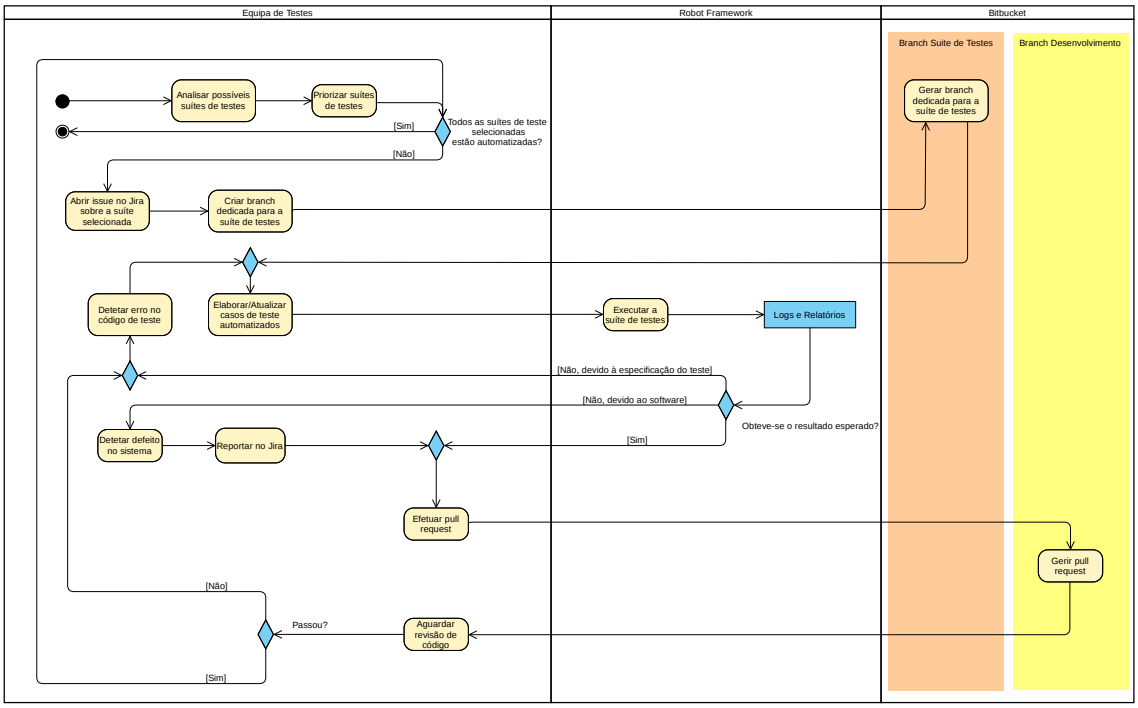


Figura 4.8: Diagrama de atividades de automatização

O primeiro passo a tomar é **analisar as possíveis suítes de teste**, onde são estudadas as diversas funcionalidades do produto que se pretende testar. A partir dessa análise, é definido um **critério de priorização** da automatização – de suítes e de testes – relevante para o projeto em questão.

Inicialmente nenhuma das suítes selecionadas está automatizada, portanto o próximo passo será a elaboração das suítes de testes, conforme a prioridade definida.

É necessário anunciar o início do trabalho dessa suíte de testes, ou seja, **criar um *issue*** – um formulário de trabalho – na plataforma *Jira*.

Para além disto, é necessário criar no sistema de controlo de versões uma ***branch* exclusiva para o desenvolvimento** dessa suíte de testes. A *branch* deverá ter um número identificador igual ao *issue* criado no passo anterior.

Na plataforma de controlo de versões é gerada a *branch* dedicada ao desenvolvimento de uma suíte de testes.

Após esta fase de preparação, é iniciada a fase de **elaboração dos casos de teste**. Neste caso, serão descritos os casos de teste no RF, usando a sintaxe *Gherkin*. Esta atividade supõe não só a elaboração dos casos de teste em *Gherkin*, como também a criação de novas *keywords* de utilizador e de bibliotecas de utilizador, conforme necessário.

Quando a suíte de testes estiver aparentemente concluída, **o conjunto de casos de teste da suíte é executado pelo RF** onde será gerado um conjunto de documentos – *Logs* e Relatórios – com as demais métricas da execução da suíte de testes. Nomeadamente, essas métricas indicam se a suíte de testes passou, caso todos os testes tenham passado, ou se falhou, caso pelo menos um teste tenha falhado.

Existem três análises possíveis do resultado gerado pelo RF. A primeira é que se obteve o resultado esperado, sendo que as outras duas implicam que não se obteve o resultado esperado. Isto poderá ter acontecido, ou pela má especificação de um caso de teste, ou devido a um defeito no software. Aferir se o resultado negativo acontece devido ao teste ou ao sistema depende do conhecimento do profissional sobre o sistema em causa.

Caso o relatório mostre um resultado negativo devido à especificação do caso de teste, é necessário analisar o caso de teste que contém o defeito e **detetar esse mesmo defeito no código**. A atividade seguinte deve ser novamente a elaboração/atualização de casos de teste, mas num contexto de correção, ou seja, quando é necessário corrigir determinado teste já feito e executado, para ir ao encontro do comportamento real do sistema em teste.

Caso o relatório mostre um resultado negativo devido ao software, é necessário **encontrar o defeito concreto do sistema**. Depois de saber qual o defeito no sistema, é necessário **reportar o mesmo na plataforma *Jira***, para que possa ser corrigido, permitindo assim que o sistema tenha o comportamento esperado.

Por fim, caso o relatório mostre um resultado positivo, ou um resultado negativo devido a um defeito no sistema e já devidamente reportado, **é feito o *pull request***. Isto é um pedido de junção da *branch* da suíte de testes elaborada com a *branch* de desenvolvimento, sendo que esta última tem suítes de testes finalizadas. As suítes de testes na *branch* de desenvolvimento são executadas de modo automático e periodicamente, sem qualquer intervenção humana, no servidor de automatização *Jenkins*.

A gestão do *pull request* implica que seja designado um colega para rever o código submetido.

Na fase de espera do *pull request* é feita a **revisão do código por terceiros**. A revisão pode adicionar algumas sugestões e correções ao código, sendo que é necessário retornar à atividade de elaboração/atualização dos casos de teste e alterar os testes em questão, seguindo o fluxo normal.

Assim que a revisão tiver um resultado positivo, o código da suíte de testes é automaticamente integrado com o código na *branch* de desenvolvimento, ou seja, passa a pertencer a bateria de testes executados regularmente no *Jenkins*.

Este ciclo repete-se até não existirem mais suítes e, consequentemente, não existirem mais testes para automatizar, sendo que a *branch* de desenvolvimento integra agora todas as suítes selecionadas na atividade de priorização.

4.5 Automatização

Em seguida, são aprofundadas algumas das atividades enunciadas na secção 4.4, nomeadamente, a análise inicial e priorização de possíveis suítes a automatizar, tal como a sua consequente automatização. Para compreender melhor o processo de escrita dos testes automatizados, é explicada a estrutura do projeto de testes no RF, assim como um exemplo prático da construção de um teste automatizado.

4.5.1 Análise de possíveis suítes de teste

Como já referido no diagrama de atividades de automatização 4.8, a automatização de testes de um dado software inicia-se com uma análise das possíveis suítes de testes e consequente priorização. Inicialmente o plano era partir de um STD relativo ao *frontend* e traduzi-lo, com as devidas alterações, em testes de *backend*. Contudo, devido a limitações técnicas, nomeadamente a inexistência de operações disponibilizadas na API relativas aos serviços testados no *frontend*, a abordagem teve de ser alterada.

Sendo que o volume de operações disponíveis foi o fator mais restritivo, os serviços disponibilizados pela API foram o foco da análise de possíveis suítes de testes e não o STD com testes de *frontend*, como inicialmente suposto. Nesta análise foram estudados quais os serviços mais intuitivos de trabalhar, no sentido em que existe uma relação direta entre a utilização dos serviços e o que é disponibilizado visualmente no *Human-Machine Interface* (HMI). Contrariamente, existem serviços que não têm impacto direto no que é apresentado no HMI. Para além disto, verificou-se que existem serviços que permitem as várias operações de manipulação de um recurso, como criação, alteração, visualização e remoção. Sendo que uma suíte de testes referente a esse serviço, para além de ser mais completa, assegura que o estado da base de dados no fim da execução da suíte é igual ao estado da base de dados antes da execução.

Da análise anterior, sabemos quais os serviços mais intuitivos a testar e quais os menos, sendo que se priorizou os testes dos serviços mais intuitivos e que possuem manipulações

mais completas dos recursos disponibilizados. De entre esses serviços não houve uma priorização.

Para cada serviço selecionado – uma suíte de testes –, todos os *endpoints* terão pelo menos um caso de teste associado, quando a API retorna uma mensagem positiva, e poderão ter múltiplos casos de testes onde é retornada uma mensagem de erro prevista.

4.5.2 Estrutura das suítes de teste no *Robot Framework*

A automatização de um caso de teste supõe a criação de *keywords*, para além de se utilizarem bibliotecas internas, externas e de utilizador. Os testes implicam a construção de mensagens em formato JSON e o seu envio em pedidos REST para a API e devem verificar se a mensagem de resposta é a esperada. Esta mensagem da API deve não só ser a esperada como deve estar em concordância com as respetivas mudanças na base de dados.

Para melhor ilustrar a construção de uma suíte de testes, segue-se o diagrama de componentes 4.9 de um projeto do *Robot Framework*.

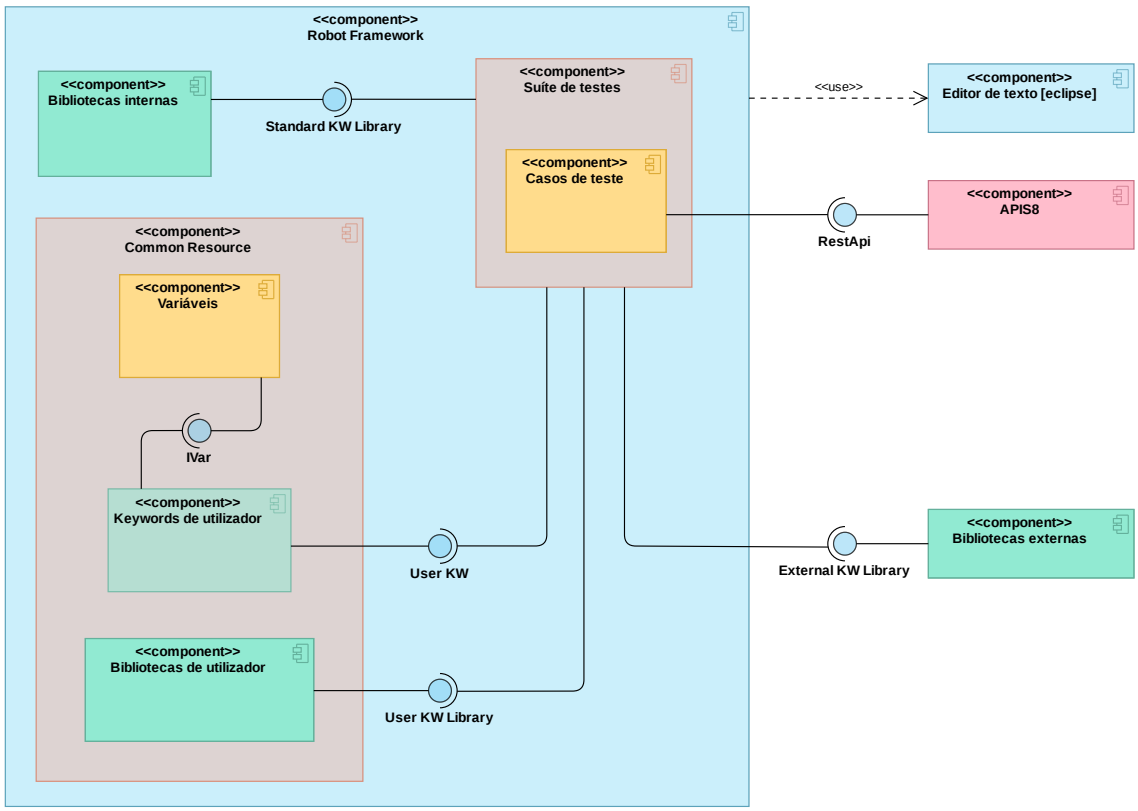


Figura 4.9: Diagrama de componentes do RF

Neste diagrama de componentes é representado o projeto de testes automatizados no RF, incorporando a estrutura relativa aos testes de *backend* e a estrutura em comum com outros tipos de testes, não representados no diagrama.

Sendo que é possível utilizar o RF em múltiplos editores de texto/IDE's, para este projeto é utilizado o *Eclipse* IDE. Existem três áreas principais no projeto de testes automatizados. A primeira área é relativa exclusivamente aos testes de *backend*, a segunda área é relativa a todos os recursos partilhados entre os testes de *backend* e outros tipos de testes e a última área do projeto é relativa a esses outros tipos de testes, como por exemplo, testes de *frontend*, que não estão representados no diagrama.

Uma suíte de testes é composta por um ou mais casos de teste, sendo que os testes são feitos sobre o sistema, o APIS8-Doha, através de pedidos REST à API ou através de acessos à base de dados. A suíte de testes utiliza, como recurso direto, um ficheiro de *keywords* de utilizador, exclusivo para testar o *backend*. Estas *keywords* de utilizador por sua vez utilizam três ficheiros principais: as variáveis, outras *keywords* de utilizador com diferente nível de abstração e *keywords* de utilizador comuns para vários tipos de testes. As *keywords* de utilizador com diferente nível de abstração são um conjunto de *keywords* que não são usadas diretamente nos casos de teste, mas possuem todas as características das restantes *keywords* de utilizador.

A razão deste ficheiro usar as *keywords* de utilizador de alto nível deve-se, não por utilizar diretamente as *keywords*, mas sim por ser uma maneira desse ficheiro poder aceder a todas as bibliotecas importadas. As *keywords* de utilizador comuns para vários tipos de testes estão incluídas na área comum do projeto – *Common Resources* – e importam todas as bibliotecas necessárias para qualquer teste. Isto inclui bibliotecas internas, ou seja, integradas já no RF; bibliotecas externas, neste caso para testes de *backend* foi utilizada a biblioteca *RequestsLibrary*; e bibliotecas de utilizador, escritas em *Python*, essencialmente para acessos à base de dados.

4.5.3 Automatização de um caso de teste

Para ilustrar o processo de automatização vai ser utilizado o caso de teste "*Send content template*" da suíte de testes "*MessageService*". Este teste simula a criação de uma mensagem ao passageiro usando um *template*. Uma mensagem ao passageiro pode ser visual, exposta em ecrãs, ou áudio, através de altifalantes. Neste caso é uma mensagem visual em que já existe uma organização prévia dos vários elementos visuais. Para além disso, é definida a duração da mensagem, assim como os dispositivos onde será emitida – ecrãs.

Uma suíte de testes é sempre referente ao mesmo serviço sendo que um caso de teste é referente a um par *endpoint*/operação. Neste caso, o recurso que será manipulado é a "*Message*" através de uma operação de *post*, em que é criada uma instância desse recurso. Como podemos ver na figura 4.10, o caso de teste é escrito em *Gherkin*, seguindo a estrutura *Given*, *When* e *Then* e utilizando uma linguagem natural.

```
SEND CONTENT TEMPLATE
    [Documentation]  Sends a post request to send a content template message
    ...    and expects response status 200
           Given there is a session created
           When a post request to send a template message is made to the API
           Then the API should respond with status "200"
           And the API should respond with the message ID
           And the database should contain the template message
```

Figura 4.10: Caso de teste - *Send content template*

Cada uma destas frases, excluindo as palavras reservadas do *Gherkin*, corresponde a uma *keyword* de utilizador, sendo que algumas foram feitas especificamente para este caso de teste e outras podem ser reutilizadas, quer dentro da própria suíte de testes, quer noutras suíte de testes relativas ao *backend*. Ao analisar o caso de teste, temos:

- *Given there is a session created* – *"Given"* indica a pré-condição, neste caso deve existir uma sessão para a API do APIS8-Doha, sendo que essa sessão será utilizada durante o teste;
- *When a post request to send a template message is made to the api* – *"When"* indica a ação principal, neste caso é feito um pedido *post* à API cujo corpo contém informação necessária para criar uma mensagem tipo;
- *Then the API should respond with status "200"* – *"Then"* indica o resultado esperado, neste caso é esperado que a API responda de forma positiva, indicando o estado "200";
- *And the API should respond with the message ID* – *"And"* indica mais um passo, neste caso no resultado esperado, onde se espera que a API responda também com o número identificador da mensagem criada;
- *And the database should contain the new template message* – Para além da verificação da resposta da API em si, é verificado também se a base de dados contém a nova mensagem criada.

Cada uma destas *keywords* pode ser desdobrada em uma ou mais *keywords*, quer sejam de utilizador ou de bibliotecas (internas, externas e de utilizador).

Por exemplo, como se pode analisar na figura 4.11, *"a post request to send a template message is made to the api"* é composta por mais duas *keywords*. *"Post Request"*, uma *keyword* de uma biblioteca externa (*RequestsLibrary*) que efetua o pedido na sessão previamente criada, para o URL correspondente e utilizando parâmetros pré-definidos, sendo que a resposta deste pedido é guardada numa variável. *"Set Test Variable"* é uma *keyword* de uma biblioteca interna que guarda o valor da resposta durante o caso de teste para, por exemplo, verificar nos passos seguintes o valor do estado da resposta e o número identificador.

```
a post request to send a template message is made to the API
    [Documentation]  Sends a post request to send a template message
    ...    Saves the request response for later usage
           ${resp}=       Post Request          apis8      /message/sendContentTemplate?
                                                           contentTemplateId=${content_template_id}
                                                           &deviceIds=${device_id}
                                                           &duration=30
           Set Test Variable        ${resp}
```

Figura 4.11: *Keyword* de utilizador – exemplo 1

"And the database should contain the new template message" também é composta por múltiplas *keywords*, como exposto na figura 4.12. *"Template Message Exists"* pertence a uma biblioteca de utilizador, ou seja, é uma *keyword* escrita pelo utilizador em *Python*, e devolve verdadeiro ou falso, consoante o objeto em questão exista na base de dados. A *keyword "Should be true"* pertence a uma biblioteca interna e verifica se o valor passado como argumento tem um valor verdadeiro.

```
the database should contain the template message
    [Documentation]  Checks if the database contains the template message
       ${exists}              template Message Exists         ${messageId}
       Should be true     ${exists}
```

Figura 4.12: *Keyword* de utilizador – exemplo 2

4.6 Resumo

Ao longo desde capítulo foi acompanhado o processo de automatização dos casos de testes. Inicialmente, foram expostos os objetivos pretendidos com esta dissertação, seguidos de um protótipo para ilustrar melhor o que seria o produto final do trabalho. Antes de se iniciar o trabalho, foi apresentada a *framework* de automatização utilizada, assim como uma comparação com a sua possível alternativa. Foi também apresentado o servidor de automatização escolhido e uma possível alternativa. Embora este não tenha sido utilizado diretamente no decorrer da tese, é através dele que os testes automatizados são executados autonomamente. Foi exposto gradualmente o processo de escolha de casos de testes, assim como a sua automatização no ambiente empresarial da Thales, seguido de uma secção mais detalhada sobre a elaboração concreta dos casos de testes no RF.

Capítulo 5

Avaliação

Esta secção engloba a avaliação dos resultados obtidos. Primeiro são ilustradas as métricas retiradas e a razão das mesmas, depois é feita a exposição dessas métricas em detalhe e, em último, encontra-se uma breve discussão sobre os valores obtidos, aliados ao uso do *Robot Framework*.

5.1 Planeamento

A avaliação da automatização dos casos de teste encontra-se dividida em duas partes lógicas. As primeiras cinco secções expõem métricas relativas ao volume de trabalho elaborado, sendo que a sexta e última secção compara os tempos de execução manual com os de execução automática.

São apresentadas métricas relativas aos serviços e operações, resumidamente, é feita a contabilização e relação entre os serviços e operações disponibilizados e os testados. A análise segue para as suítes de testes, que têm uma relação de um para um com os serviços existentes, onde é analisada a composição de cada suíte elaborada. A avaliação aprofunda-se e são analisados os casos de teste, também referente à sua composição, assim como as *keywords* constituintes. Para terminar a primeira parte, é analisado o volume total de trabalho relativo às novas *keywords* geradas.

Na última parte da avaliação, foi selecionada uma suíte de testes cujos tempos de execução manual e automática foram comparados, terminando com uma inferência de tempos de execução para o futuro.

5.2 Resultados

Nesta secção são analisados os resultados relativos aos serviços e operações testados, às suítes de testes elaboradas, aos casos de teste correspondentes, às *keywords* empregues e às *keywords* de utilizador criadas. De seguida, é descrita a execução manual de uma suíte de testes e é realizada a comparação e análise dos tempos de execução relativamente à execução automática.

5.2.1 Serviços e operações

A API do APIS8 disponibiliza vinte e oito (28) serviços num total de cento e vinte e duas (122) combinações entre recursos e operações. Ao longo da dissertação a palavra "operação" será usada para designar uma operação de qualquer tipo (*post*, *put*, *get*, *delete*) sobre um recurso específico. Deste modo, existem cinquenta e duas operações de *get* (42,62%), trinta e sete de *post* (30,32%), vinte de *put* (16,39%) e treze de *delete* (10,65%). Como não houve uma avaliação global *a priori* de quais os serviços e operações possíveis/relevantes a serem testados para o projeto de Doha, não se pode considerar que seria relevante fazer testes para todos os vinte e oito (28) serviços disponibilizados.

Exposto isto, foram testados cinco (5) serviços, num total de trinta e três (33) operações, sendo que dezassete são operações de *get*, o que perfaz mais de metade (52%) das operações testadas. Em menor quantidade encontram-se os testes de operações *post* (24%), seguidos por operações *delete* (15%) e, por fim, testes a operações de *put* (9%).

No diagrama de barras da figura 5.1, pode-se observar a distribuição do tipo de operações, quer nos serviços totais, quer nos serviços testados.

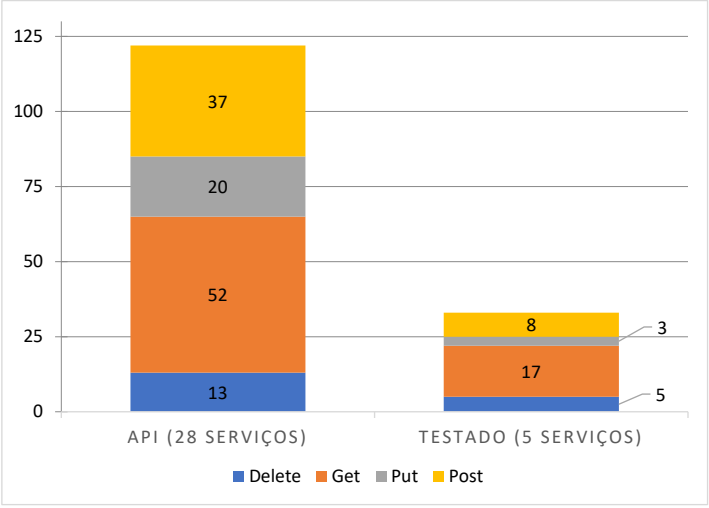


Figura 5.1: Distribuição de operações

Com base nesta informação, é contabilizada a cobertura dada pelos testes relativamente aos serviços e ao tipo de operações, exposta na tabela 5.1. Dos vinte e oito serviços

disponibilizados, cinco foram testados, resultando numa cobertura de 17,86%. Relativamente às operações, foram testadas trinta e três num total de cento e vinte e duas, resultando numa cobertura de 27,05%. Os tipos de operações que apresentam maior cobertura são as de *delete* e *get*, 38,46% e 32,69% respetivamente. Por um lado, isto deve-se ao facto de existirem menos operações de *delete*, em geral uma por recurso, ou seja, as que foram testadas têm maior peso. Por outro lado, apesar de existirem mais operações de *get*, quer seja de um recurso específico, quer seja de um conjunto de recursos, são operações relativamente fáceis de testar. Os restantes testes conferem uma cobertura de 21,62% às operações de *post* e uma cobertura de 15,00% às operações de *put*.

Tabela 5.1: Cobertura da API

	Cobertura dada pelos testes
Serviços	17,86%
Operações Totais	27,05%
Operações de *post*	21,62%
Operações de *put*	15,00%
Operações de *get*	32,69%
Operações de *delete*	38,46%

5.2.2 Suítes de testes

Uma suíte de testes é referente a um serviço, assim, existem cinco suítes de testes distintas e cuja quantidade de testes está associada à quantidade de recursos que cada serviço manipula.

Para cada operação sobre um recurso específico, deve existir um teste que valida o comportamento positivo desejado, ou seja, devem existir trinta e três (33) testes deste género. Por sua vez, também foram desenvolvidos testes que validam as respostas de erro esperadas, o que totaliza vinte (20) casos de teste deste género.

Visto que existem duas operações onde só foi possível testar o caso negativo esperado e não o positivo, existem trinta e um (31) casos de teste positivos. No total foram elaborados cinquenta e um (51) testes sobre as trinta e três (33) operações distintas.

Na tabela 5.2 são expostos os valores, para cada uma das diferentes suítes de testes, a quantidade de casos de teste positivos, de casos de teste negativos – em que é esperada uma mensagem de erro – e a quantidade de casos de teste totais. Em média, cada suíte de testes apresenta cerca de seis casos de teste positivos e quatro casos de teste negativos. Para os casos de teste negativos não foram exploradas todas as possibilidades de má construção do pedido para cada recurso, optou-se por verificar apenas o comportamento do sistema quando é feita referência a um recurso não existente.

Pode-se reparar que a suíte de testes C apresenta muito mais casos de teste que as restantes suítes, isto deve-se a facto de o serviço gerir mais recursos, concretamente este

Tabela 5.2: Constituição das suítes de teste

	Casos de teste positivos	Casos de teste negativos	Casos de teste totais
A	3	0	3
B	4	1	5
C	15	14	29
D	2	3	5
E	7	2	9
Total	31	20	51
Média	6,2	4	10,2

serviço manipula cinco recursos, enquanto que os restantes apenas manipulam um.

5.2.3 Casos de teste

A complexidade de um caso de teste pode ser medida pela quantidade de *keywords* utilizadas no mesmo. Sendo que todos os testes seguem a estrutura *Gherkin*, necessitando de pré-condição, ação e pós-condição, no mínimo existem três *keywords*, uma para cada etapa. Contudo, cada uma destas etapas pode ser constituída por múltiplas *keywords*. Por exemplo, a etapa da pré-condição pode ser composta por várias *keywords*, sendo que a sua estrutura seria como na listagem 5.1

Listagem 5.1: Caso de teste – etapa de pré-condição com duas *keywords*

```
1 Given first precondition
2 And second precondition
3 When some action
4 Then some postcondition
```

A tabela 5.3 indica, para cada suíte de testes, a quantia de casos de testes, o nível de utilização de *keywords* – repetições incluídas – e a complexidade média de um caso de teste da respetiva suíte. Por exemplo, a suíte B possui cinco casos de teste, num total de vinte e três *keywords*, sendo que, em média, cada um dos casos de teste é composto por 4,6 *keywords*.

Contabilizando estes valores para todas as suítes de testes, é obtida uma média de 10,2 casos de teste por suíte e uma utilização de *keywords* de 48,4 na totalidade da suíte. De notar que a suíte C tem valores bastante diferentes das restantes suítes e, por isso, se não fosse contabilizada, a média desceria para 5,5 casos de teste por suíte e uma utilização total de 24,5 *keywords* na totalidade.

Relativamente à complexidade dos casos de testes, nota-se que é um valor semelhante para todas as suítes de teste e, em média, cada caso de teste é composto por 4,75 *keywords*. Isto acontece pois, geralmente, a etapa da pós-condição possui duas *keywords*, uma para a verificação da mensagem dada pela API e outra para a verificação da base de dados.

Tabela 5.3: Constituição dos casos de teste

	Casos de teste totais	Utilização de *keywords*	Média de *keywords* por casos de teste
A	3	11	3,67
B	5	23	4,60
C	29	144	4,97
D	5	23	4,60
E	9	41	4,56
Total	51	242	-
Média	10,2	48,4	242/51=4,75

5.2.4 Utilização de *keywords*

De forma semelhante à análise da complexidade dos casos de teste face ao volume de *keywords* utilizadas, será analisada a complexidade das *keywords* utilizada face ao tipo e volume de *keywords* constituintes. Todas as *keywords* presentes nos casos de teste são *keywords* de utilizador, logo, são compostas por outras *keywords*. A sua composição pode ter *keywords* de bibliotecas internas/externas, de bibliotecas de utilizador ou mesmo outras *keywords* de utilizador, para proporcionar o nível de abstração desejado no teste.

Tabela 5.4: Complexidade de *keywords* utilizadas

	Média	
	keywords por caso de teste	Complexidade das *keywords* constituintes
A	3,67	2,43
B	4,60	2,27
C	4,97	2,69
D	4,60	1,92
E	4,56	2,71
Média	242/51=4,75	2,40

A tabela 5.4 indica o valor já mencionado da média de *keywords* por caso de teste (4,75) e a complexidade média de cada uma delas (2,40), assim como os valores totais para cada suíte de teste. Por exemplo, para a suíte B, um caso de teste é composto por 4,6 *keywords* e cada uma delas é composta por outras 2,27. Ou seja, se não se utilizasse a estrutura *Gherkin* ou uma separação de níveis de abstração, um caso de teste seria constituído por 10,4 *keywords* (4,6*2,27). Este valor pretende dar a entender a complexidade dos casos de testes elaborados, é relevante perceber que as *keywords* se desdobram noutras *keywords*, cada vez com nível de abstração menor.

A tabela 5.5 expõe a composição de cada *keyword* utilizada nos casos de teste. Relembrando que, em média, uma *keyword* é composta por outras 2,4 *keywords*, podendo a sua

composição ser mista relativamente ao tipo das mesmas.

Utilizando novamente a suíte B como exemplo, um teste é composto, em média, por 4,6 *keywords* e, segundo a tabela, 20% delas têm na sua composição *keywords* de utilizador, 100% têm na sua constituição *keywords* de bibliotecas internas/externas e 27% são constituídas por bibliotecas de utilizador.

Tabela 5.5: Composição de *keywords* utilizadas

	keywords constituídas por		
	keywords de utilizador	bibliotecas internas/externas	bibliotecas de utilizador
A	9%	55%	0%
B	20%	100%	27%
C	8%	97%	40%
D	15%	100%	23%
E	21%	100%	29%

A *keyword "a post request to save a new custom layout with a repeated name is made to the API"* é a única na etapa de ação – *when* – de um determinado caso de teste da suíte C. A definição desta *keyword* é apresentado na figura 5.2. Como pode ser observado, a sua composição é mista, integrando *keywords* de bibliotecas internas e externas, como acontece a 97% das *keywords* da suíte C, e integrando *keywords* de utilizador, como acontece a 21% das *keywords*. Fazendo a análise para todas as suítes, reparamos que as *keywords*

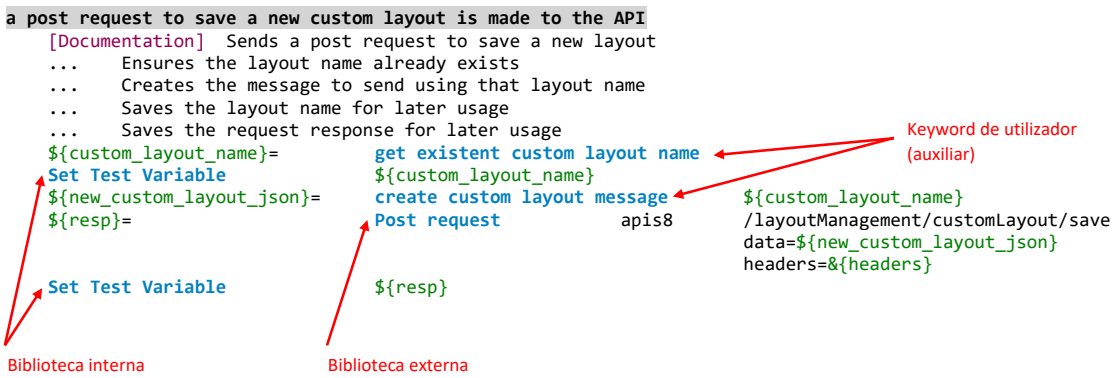


Figura 5.2: Definição da *keyword* de utilizador

são constituídas quase sempre por bibliotecas internas/externas. Normalmente todas as *keywords* necessitam de guardar valores em variáveis e para isso são utilizadas as bibliotecas internas, ou então fazem uso de *keywords* da biblioteca externa para os pedidos à API. Relativamente à restante composição, apenas *keywords* relativas à base de dados necessitam de *keywords* de bibliotecas de utilizador para o acesso à base de dados. Por fim, a razão de existirem *keywords* de utilizador que utilizam outras *keywords* de utilizador é que se pretende manter um determinado nível consistente de abstração nos casos de teste e, por vezes, isso apenas é possível criando essas *keywords* como auxílio.

5.2.5 *Keywords* elaboradas

Nesta parte é avaliada a quantidade de *keywords* geradas, contrariamente às avaliações anteriores que analisavam a utilização, com repetições, das *keywords*. No total foram elaboradas cento e vinte e duas (122) *keywords*, em que cento e nove (109) são utilizadas diretamente nos diversos casos de teste e as restantes treze (13) são consideradas auxiliares. Relativamente à biblioteca de utilizador construída, foram elaborados mais vinte e três (23) funções *Python*. De forma semelhante ao que foi analisado na secção anterior, a maioria das *keywords* (96,72%) é constituída por *keywords* de bibliotecas internas/externas, em menor valor (35,77%) são constituídas por *keywords* de bibliotecas de utilizador e ainda existem algumas *keywords* (13,11%) que se desdobram noutras *keywords* de utilizador de mais baixo nível.

Tabela 5.6: Constituição das *keywords* elaboradas

	Alto nível	Auxiliares	Total
keywords elaboradas	109	13	122
Complexidade das etapas (média)	2,62	2,08	-
Constituídas por *keywords* de utilizador	14 (12,84%)	2 (12,38%)	16 (13,11%)
Constituídas por bibliotecas internas/externas	106 (97,24%)	12 (92,31%)	118 (96,72%)
Constituídas por bibliotecas de utilizador	39 (35,77%)	0 (0%)	39 (35,77%)

5.2.6 Tempos de execução

Para efetuar a comparação de tempos manual e automático, foi selecionada uma suíte de testes bastante completa, no sentido em que existem testes para múltiplos *endpoints* (recursos), com múltiplas operações (*post*, *put*, *get*, *delete*) que esperam respostas positivas e também de erro. À semelhança da execução manual de testes do projeto APIS8-Dubai, a execução da suíte de teste em causa consiste em:

1. Preencher os campos necessários para construir a mensagem do pedido. Os valores devem estar predefinidos, sendo apenas necessário copiar de um ficheiro de texto e colocar na interface da API (*Swagger*);

2. Realizar o pedido;

3. Analisar a reposta enviada pela API. Consoante os testes, isto pode implicar verificar apenas o *status* da resposta ou também a mensagem de resposta, seja de erro ou não, neste caso, pode ser necessário verificar o tipo de objeto presente no corpo da resposta ou verificar quantidades ou mesmo analisar objetos por completo;

4. Verificar a concordância da base de dados com a resposta recebida. Isto implica aceder à base de dados e fazer as respetivas *queries*, confrontando esse resultado com o explícito na resposta da API.

A suíte de testes escolhida engloba um serviço analisado em dezassete (17) operações, resultando num total de vinte e nove (29) testes. A execução manual completa da suíte de testes selecionada demorou trinta (30) minutos e trinta e três (33) segundos. Em modo automático, contabilizando cinco execuções no servidor de automatização expostas na tabela 5.7, é obtida a média de 13,4 segundos para executar todas as suítes de testes e uma média de 4,2 segundos para executar apenas a suíte selecionada.

Tabela 5.7: Tempos de execução automatizada no *Jenkins*

Timestamp Jenkins	Todas as suítes	Suíte selecionada
20201013 10:50:09 UTC+01:00	12s	5s
20201014 15:28:02 UTC+01:00	11s	4s
20201016 20:22:04 UTC+01:00	17s	4s
20201017 20:22:58 UTC+01:00	18s	4s
20201018 20:22:37 UTC+01:00	9s	4s
Média	13,4s	4,2s

5.3 Discussão

Com os dados apresentados na secção 5.2.6, é possível inferir o tempo que será poupado ao transitar para testes automatizados. Concretamente para a suíte usada, o valor de execução manual é de 30 minutos e 33 segundos e de automática é de 4,2 segundos. O tempo que se investiu para executar manualmente corresponde a 436,43 execuções automáticas. Sendo que os testes automatizados correm uma vez por dia, a quantidade de execuções apresentada concede garantia de qualidade do serviço testado durante um ano, dois meses e dez dias.

Para além da óbvia diminuição do tempo de execução, não é necessária ação humana na execução, a automatização possibilita, não só executar os testes mais rapidamente e diminuir o tempo de *feedback*, como também libertar esse tempo ao profissional. Por exemplo, para assegurar a qualidade do produto no período referido, um profissional teria de dedicar exclusivamente cerca de um mês e uma semana (1,39 mês) do seu tempo para executar o mesmo teste em modo manual. É possível argumentar que, embora este investimento de tempo, e de outros recursos, garanta um certo nível de qualidade ao produto durante o mesmo tempo, o nível de garantia dado pela execução manual dos testes é inferior, devido à possível indução de erros humanos.

O *Robot Framework* permitiu que a descrição dos casos de teste fosse de alto nível e homogénea para todas as suítes elaboradas (cerca de 4,75 *keywords* por teste) através

da separação de *keywords* por níveis de abstração. Nem sempre foi possível compor as *keywords* dos casos de teste com apenas *keywords* de bibliotecas, deste modo, para manter o nível de abstração dos casos de teste, com o RF foi possível criar um nível mais baixo de abstração de *keywords* de utilizador (13 *keywords*). Para além disto, a criação de bibliotecas de utilizador foi fundamental para manter a abordagem BDD ao longo de todo o projeto de automatização, pois permitiu criar funções relativas à base de dados com nomes descritivos e em linguagem natural (23 *keywords*).

5.4 Resumo

Este capítulo abordou as demais métricas do volume de trabalho, com o intuito de esclarecer a composição das suítes de testes, dos casos de testes e de todas as *keywords* utilizadas. Foi abordado também o impacto da realização destas suítes de testes, quer a nível da cobertura da API, quer a nível dos tempos de execução.

Capítulo 6

Conclusão

6.1 Contribuições

A Thales Portugal encontra-se na transição para um desenvolvimento *Agile*, como tal, um dos pontos que adotou foi a automatização de determinados casos de teste. A presente dissertação focou-se na elaboração de testes automatizados, relativos ao *backend*, para o produto APIS8, que já se encontra em manutenção. Outro ponto da metodologia *Agile* que a empresa se debruçou foi o investimento numa melhor comunicação entre todos os profissionais envolvidos. Para isto, os testes desenvolvidos seguem a metodologia BDD que, através do uso da sintaxe *Gherkin* e da utilização de *keywords* facilitadas pelo RF, possibilita a escrita dos testes numa linguagem semelhante à linguagem natural. Os testes são considerados de aceitação pois, através da sua linguagem clara, permitem mostrar ao cliente se os requisitos definidos pelo mesmo são alcançados ou não.

Ao longo desta dissertação foram elaborados no total 51 casos de testes sobre cinco serviços, expostos na API do APIS8. Adicionalmente, a maioria dos testes verifica também a concordância da base de dados com o respetivo pedido realizado. Para a construção dos casos de teste foi necessário criar vários níveis de abstração, de modo a facilitar a abordagem BDD, resultando em 122 *keywords* de utilizador e uma biblioteca de utilizador composta por 23 *keywords*.

Foi cronometrada a execução manual de uma suíte de testes, cujo valor foi de trinta minutos e trinta e três segundos, e comparada com o tempo médio de execução, 4,2 segundos, no servidor de automatização *Jenkins*. Com estes valores inferiu-se que a automatização dessa única suíte pouparia a um profissional cerca de um mês e uma semana de trabalho, para garantir relativamente a mesma confiança sobre a qualidade do produto durante um ano, dois meses e dez dias. Será seguro assumir que, com as restantes suítes automatizadas, o tempo poupado pelos profissionais será superior ao tempo referido.

Para além do tempo que poderá ser reinvestido noutras áreas, como a execução de testes exclusivamente manuais, a garantia de qualidade fornecida por testes automatizados é superior a uma execução manual. Aquando da execução manual da suíte de testes em questão, foram introduzidos erros, o que implicou refazer o teste respetivo ou toda a suíte, somando ainda mais tempo à normal execução manual. É possível, no entanto, que não tenham sido detetados todos os erros, alterando assim o percurso esperado do teste e minimizando a garantia de qualidade do mesmo.

Esta tese focou-se, não só na automatização de casos de teste, como na própria descoberta de casos de teste, visto não existir nenhum STD que englobe testes ao *backend*. Contudo, espera-se que esta bateria de testes automatizados bastante descritivos seja motivante para a elaboração e manutenção do documento. De notar também que, ao longo de elaboração dos casos de testes, foram detetados e reportados defeitos do sistema, sendo que agora, com a execução regular dos testes, garante-se que esses defeitos ficaram corrigidos permanentemente e, caso apareçam novos, são detetados no espaço de um dia.

6.2 Limitações

A inexistência de um documento descritivo dos casos de testes a elaborar dificultou a automatização, dado que foi necessário, não só adaptar o trabalho a uma nova *framework* e linguagem, como também a um novo sistema, cuja API estava insuficientemente documentada.

Consequentemente, a seleção das suítes/testes a executar não sofreu nenhuma análise rigorosa, o que poderia ser prejudicial caso o produto estivesse numa fase de desenvolvimento crítico e precisasse de *feedback* relevante para avançar. Não foi o caso do APIS8, que está em fase de manutenção e não prevê novas funcionalidades, contudo, para garantir um bom investimento na automatização de casos de testes, os mesmos devem ser selecionados com maior rigor.

Durante parte da elaboração da dissertação foi sentida falta de recursos humanos e, consequentemente, da disponibilidade dos existentes. Isto resultou numa comunicação de menor qualidade que, por vezes, atrasou ou eliminou vertentes da dissertação.

6.3 Trabalho futuro

Existem alguns pontos onde é possível melhorar o trabalho realizado. Embora não entre no escopo da dissertação, a disponibilização de mais operações da API e o aperfeiçoamento das já existentes possibilitará uma maior e melhor bateria de testes para o sistema. Enquadrado no âmbito da tese, a continuação deste trabalho deve ter em conta:

1. Seleção dos casos de teste – a seleção dos novos casos de teste a automatizar deve ter um critério definido e o mesmo deve ser sistematicamente aplicado. A geração de novos casos de teste pode ser auxiliada com ferramentas para esse objetivo,

concretamente na geração de casos de teste para uma API. Caso esta via não seja possível, é necessário que sejam descritos os critérios dos testes às demais operações, isto é, idealmente deverão ser testados todos os casos positivos e negativos, com combinações de *input* relevantes.

2. Construção do STD de *backend* – a construção de um documento que relata todos os testes que garantem a qualidade do *backend* deve existir, integrando não só os testes automatizados, mas também testes manuais. Com a aplicação do BDD, a legibilidade dos casos de teste automatizados facilita a tradução dos mesmos para um STD formal.

3. Automatização na Thales – na presente dissertação foram elaborados casos de testes automatizados para o *backend* de um projeto específico, o APIS8 – Doha. Estes casos de teste podem ser aproveitados para outros projetos do APIS8 ou para futuras versões, com algumas modificações.

4. Melhoramento das suítes de testes – existe espaço para melhorar as suítes de testes existentes, quer seja em volume ou qualidade. Existem operações por explorar e, em alguns casos, mais operações poderão ser criadas para um dado serviço, como referido no início da secção. Para além disto, poderá ser benéfico, consoante o critério adotado, criar mais casos de teste para a mesma operação, alterando combinações de *inputs* e gerar, quer casos de teste positivos, quer casos de teste negativos. É possível também tirar mais partido do RF aumentando a reutilização de *keywords*, embora isto possa ter um impacto negativo na legibilidade dos testes, é um *trade-off* a considerar.

Bibliografia

[1] P. Ammann e J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[2] *Bamboo oficial site*. https://www.jenkins.io/. Accessed: 21/11/2020.

[3] *Behat oficial site – Writing features – gherkin language*. https://docs.behat.org/en/v2.5/guides/1.gherkin.html. Accessed: 30/11/2020.

[4] S. Berner, R. Weber e R. K. Keller. "Observations and lessons learned from automated testing". Em: *Proceedings of the 27th international conference on Software engineering*. 2005, pp. 571–579.

[5] *Cucumber – Gherkin Reference*. https://cucumber.io/docs/gherkin/reference/. Accessed: 10/02/2020.

[6] M. Fowler, J. Highsmith et al. "The agile manifesto". Em: *Software Development* 9.8 (2001), pp. 28–35.

[7] B. Haugset e G. K. Hanssen. "Automated acceptance testing: A literature review and an industrial case study". Em: *Agile 2008 Conference*. IEEE. 2008, pp. 27–38.

[8] *Introducing BDD*. https://https://dannorth.net/introducing-bdd/. Accessed: 10/02/2020.

[9] *Introducing BDD*. https://dannorth.net/introducing-bdd/. Accessed: 21/11/2020.

[10] ISTQB. "[Agile] Foundation Level Extension Syllabus – Agile Tester". Em: *International Software Testing Qualifications Board* (2014).

[11] ISTQB. "[Specialist] Advanced Level Syllabus – Test Automation Engineer Version 2016". Em: *International Software Testing Qualifications Board* (2016).

[12] ISTQB. "[Core] Foundation level Syllabus version 2018 v3.1". Em: *International Software Testing Qualifications Board* (2018).

[13] ISTQB. "[Agile] Advanced Level Syllabus – Agile Technical Tester Version 1.0". Em: *International Software Testing Qualifications Board* (2019).

[14] *IVVQ Lead Carreer*. https://jobs.thalesgroup.com/job/crawley/ivvq-lead/1766/14558301. Accessed: 05/02/2020.

[15] *Jenkins oficial site*. https://www.jenkins.io/. Accessed: 21/11/2020.

[16] J. A. Livermore. "Factors that Significantly Impact the Implementation of an Agile Software Development Methodology." Em: *JSW* 3.4 (2008), pp. 31–36.

[17] *Login Tests Test Log – Robot Framework*. https://robotframework.org/robotframework/latest/images/log_failed.html#s1-s1-t5. Accessed: 30/11/2020.

[18] *Login Tests Test Report – Robot Framework*. https://robotframework.org/robotframework/latest/images/report_failed.html#suite_s1-s1. Accessed: 30/11/2020.

[19] *Manifesto for Agile Software Development*. https://agilemanifesto.org/. Accessed: 05/02/2020.

[20] A. Martin-Lopez, S. Segura e A. Ruiz-Cortés. "Test coverage criteria for RESTful web APIs". Em: *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 2019, pp. 15–21.

[21] R. Miller e C. T. Collins. "Acceptance testing". Em: *Proc. XPUniverse* 238 (2001).

[22] *Programmatic Ponderings – Continuous Integration and Delivery of Microservices using Jenkins CI, Maven, and Docker Compose*. https://programmaticponderings.com/2015/06/22/continuous-integration-and-delivery-of-microservices-using-jenkins-ci-maven-and-docker-compose/. Accessed: 30/11/2020.

[23] *Robot Framework introduction*. https://robotframework.org/#introduction. Accessed: 31/01/2020.

[24] *Robot Framework oficial site – Robot Framework Foundation*. https://robotframework.org/foundation/. Accessed: 23/11/2020.

[25] D. Saff e M. D. Ernst. "An experimental evaluation of continuous testing during development". Em: *ACM SIGSOFT Software Engineering Notes* 29.4 (2004), pp. 76–85.

[26] C. Solis e X. Wang. "A study of the characteristics of behaviour driven development". Em: *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE. 2011, pp. 383–387.

[27] S. Stresnjak e Z. Hocenski. "Usage of robot framework in automation of functional test regression". Em: *Proc. 6th Int. Conf. Softw. Eng. Adv.(ICSEA)*. 2011, pp. 30–34.

[28] *Teamlead – Products*. https://www.teamlead.ru/display/EN/Bamboo. Accessed: 30/11/2020.

[29] *Thales em Portugal*. https://www.thalesgroup.com/pt-pt/portugal/global-presence-europe/portugal. Accessed: 31/01/2020.

[30] *Thales Group official page*. https://www.thalesgroup.com/en. Accessed: 31/01/2020.

[31] S. Tyagi, R. Sibal e B. Suri. "Adopting Test Automation on Agile Development Projects: A Grounded Theory Study of Indian Software Organizations". Em: *International Conference on Agile Software Development*. Springer, Cham. 2017, pp. 184–198.

[32] M. Wynne, A. Hellesoy e S. Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[33] P. W. Young. *Advanced Passenger Information System – User manual*. RELEASE 8.9.7. Thales Portugal. 2018.