Marco Ruben Sobral Monteiro

Licenciado em Ciências da Engenharia

Desenvolvimento de testes automatizados para *frontend*

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: David Ribeiro,
Head Of Products Development,
Thales Group

Co-orientador: Miguel Carlos Pacheco Afonso Goulão,
Professor Associado,
Faculdade de Ciências e Tecnologia
da Universidade Nova de Lisboa

Júri

Presidente:	Hervé Miguel Cordeiro Paulino, FCT-UNL
Arguente:	Vítor Manuel Alves Duarte, FCT-UNL
Vogal:	David Ribeiro, Thales Group

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2020

Desenvolvimento de testes automatizados para *frontend*

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

Dedico a elaboração desta tese à minha família, sem eles este percurso não teria sido possível.

Agradeço aos meus pais por todo o apoio incondicional que me deram não só nestes cinco anos, mas ao longo de toda a minha vida, incentivando-me sem nunca duvidarem do potencial das minhas capacidades. A eles devo tudo.

Obrigado à Shanti que tornou cada dia desta jornada em dias melhores.

Obrigado às minhas avós que apesar das suas dificuldades em perceber o meu percurso, conseguiram sempre estar presentes com o desejo de me verem conseguir concluir esta etapa da minha vida.

O meu agradecimento à minha colega e grande amiga Adriana que me acompanhou de perto durante toda a fase de elaboração da tese, com ela partilhei este percurso em que ambos desenvolvemos as nossas teses. Apesar das dificuldades, juntos conseguimos.

Obrigado a todos os meus amigos que me acompanharam até hoje, principalmente ao João e ao Duda, a quem agradeço o apoio e o companheirismo nestes cinco anos. As memórias que guardo impedem-me de imaginar este percurso sem a sua presença. Levo-os para a vida e sou agradecido por a vida académica me ter permitido conhecê-los.

O meu agradecimento ao David Ribeiro, por ter aceite ser meu orientador e pela sua colaboração.

Agradeço ao meu co-orientador professor Miguel Goulão, pelo seu apoio e pela sua contribuição através dos seus conselhos e sugestões para este trabalho.

Resumo

A qualidade dos sistemas de software é uma exigência nos dias de hoje. Assim, a fase de testes representa, cada vez mais, um papel importante no desenvolvimento ou melhoramento de um projeto. A fase de testes é uma área de estudo que tem crescido significativamente nos últimos tempos, em especial a sua automatização.

Na Thales surgiu a necessidade de planear uma estratégia de forma a automatizar validações para o *frontend* de um dos seus produtos, o APIS 8. A constante necessidade de garantir a sua qualidade, devido à utilização diária por inúmeros clientes a nível mundial, leva à exigência de um suporte contínuo do sistema. A criação de testes procura evitar a regressão do sistema, permitindo um desenvolvimento contínuo do produto, beneficiando de um *feedback* mais rápido da qualidade e consequente deteção de erros.

A presente tese propõe o desenvolvimento de casos de teste automatizados para *frontend* com base na documentação já existente do APIS 8 e por conseguinte, uma reformulação desta documentação para uma linguagem de fácil interpretação universal. A sua escrita vai beneficiar da sintaxe *Gherkin*, permitindo implementar elementos da metodologia *BDD*. Serão ainda adotadas duas técnicas de geração de casos de teste, *Cause-Effect Graphing* e *Decision Table Testing*. Estas visam combater a falta de uma metodologia na criação dos mesmos, contribuindo para uma maior qualidade do software produzido.

Com a aplicação das técnicas foi possível gerar novos casos de teste que aumentam a cobertura dos mesmos. A automatização dos testes e a consequente análise dos tempos de execução manual e automatizada permitiu concluir que esta é uma opção viável para garantir que falhas no sistema são detetadas rapidamente.

A área da automatização é uma que ainda não ganhou o destaque que realmente necessita, uma vez que os resultados comprovam que é um bom investimento devido à rapidez com que os testes são executados. Adicionalmente, a aplicação de técnicas de geração de casos de teste é algo que ainda não se encontra diretamente associado à automatização, no entanto, este é um passo crucial para criar uma bateria de testes relevante. Esta tese demonstra que existem benefícios ao investir na automatização, sendo um deles a libertação dos trabalhadores que executam de forma repetitiva testes manuais, permitindo que estes alcancem o seu potencial e desenvolvam novas e revolucionárias funcionalidades.

Palavras chave: Automatização de testes, *Robot Framework*, casos de teste, *frontend*,

keyword-driven testing, *Gherkin*, BDD, integração contínua, testes de aceitação, técnicas de geração de casos de teste, *Decision Table Testing*, *Cause-Effect Graphing*.

Abstract

The quality of software systems is a requirement these days, so the testing phase increasingly plays an important role in the development or improvement of a project. The testing phase is an area of study that has grown significantly in recent times, especially its automation.

At Thales, the need arose to plan a strategy in order to automate validations for the frontend of one of its products, the APIS 8. The constant need to guarantee its quality due to the daily use by countless customers worldwide leads to the requirement for continuous support of the system. The creation of tests seeks to avoid the regression of the system, allowing a continuous development of the product, benefiting from a faster feedback of the quality and consequent error detection.

The present thesis proposes the development of automated test cases for frontend based on the existing documentation of APIS 8 and therefore, a reformulation of this documentation into a language that is easy to interpret universally. The writing of the tests will benefit from the Gherkin syntax, allowing to implement the BDD methodology. Two techniques for generating test cases will also be adopted, Cause-Effect Graphing and Decision Table Testing. These aim to combat the lack of a methodology in their creation, contributing to a higher quality of the software produced.

With the application of the techniques it was possible to generate new test cases that increase their coverage. The automation of the tests and the consequent analysis of the manual and automated execution times allowed us to conclude that this is a viable option to ensure that failures in the system are detected quickly.

The automation area is one that has not yet gained the prominence it really needs, since the results prove that it is a good investment due to the speed with which the tests are performed. Additionally, the application of test case generation techniques is something that is not yet directly associated with automation, however, this is a crucial step in creating a relevant battery of tests. This thesis demonstrates that there are benefits to investing in automation, one of which is the liberation of workers who perform repetitive manual tests, allowing them to reach their potential and develop new and revolutionary features.

Keywords: Test automation, Robot Framework, test cases, frontend, keyword-driven testing, Gherkin, BDD, continuous integration, acceptance testing, test case generation

techniques, Decision Table Testing, Cause-Effect Graphing.

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Siglas

API *Application Programming Interface*.

APIS *Advanced Passenger Information System*.

ATDD *Acceptance Test-Driven Development*.

BDD *Behaviour Driven Development*.

DDQS *Design, Develop and Qualify the Solution*.

DSL *Domain Specific Language*.

GDP *GTS Digital Platform*.

gTAA *general Test Automation Arquitecture*.

GTS *Ground Transportation Systems*.

GUI *Graphical User Interface*.

HMI *Human-Machine Interface*.

HTML *Hypertext Markup Language*.

IT *Information Technology*.

IVVQ *Integration, Verification, Validation and Qualification*.

IVVQP *Integration, Verification, Validation and Qualification Plan*.

MUT *Model UI-Driver Tests*.

MVC *Model View Controller*.

POM *Page Object Model*.

QA *Quality Assurance*.

STD *Software Test Description*.

STP *Software Test Plan*.

STR *Software Test Report*.

SUT *System Under Test*.

TAA *Test Automation Arquitecture*.

TAF *Test Automation Framework*.

TAS *Test Automation Solution*.

TDD *Test-Driven Development*.

UI *User Interface*.

URL *Uniform Resource Locator*.

VPN *Virtual Private Network*.

XPath *XML Path Language*.

Capítulo 1

Introdução

Este capítulo foca-se na introdução da tese, começando com a descrição e contextualização do problema acompanhado pelas motivações. Seguem-se os objetivos e as contribuições com a elaboração desta dissertação. Por fim, o capítulo é fechado com uma visão geral da estrutura do restante documento.

1.1 Contexto e descrição

No mundo atual, praticamente todos os sistemas e dispositivos que nos rodeiam possuem software. Este é responsável pelo controlo de milhares de componentes, dos quais os seres humanos desenvolveram uma grande dependência ao longo dos anos. Portanto, quando um erro ocorre existem perdas, de tempo, monetárias e em casos extremos, de vidas. Logo é imprescindível que quando um software é lançado para o mercado este esteja intensamente testado e a sua qualidade e funcionamento sejam assegurados.

Esta tese enquadra-se na área de *software testing*. E o que é *software testing*? É a técnica mais usada para verificar e validar a qualidade do software através da execução de um programa ou sistema com o objetivo de encontrar falhas, garantindo consistência e evitando acontecimentos inesperados [46, 48].

De acordo com um relatório realizado pelo *American National Institute of Standards and Technology* em 2002, os impactos económicos negativos causados pela falta de infraestruturas de *software testing* custam 62 mil milhões de dólares por ano, apenas nos Estados Unidos [24]. É um processo intensivo, uma vez que é calculado que mais de 50% do custo de desenvolvimento do software é para testar o mesmo, podendo mesmo chegar aos 70% [57, 76]. Este processo encontra-se entrelaçado com o ciclo de vida de desenvolvimento, englobando a validação e verificação do sistema. A validação assegura que um produto, serviço ou sistema vai ao encontro às necessidades dos clientes [34]. A

verificação avalia se um produto, serviço ou sistema está de acordo com regulamentos, requisitos, especificações ou condições impostas [34].

Apesar de existir uma tendência crescente para a automatização, os testes manuais mantêm a sua importância, pois é necessário que *testers* e clientes tenham interação com o sistema. No entanto, com os testes automatizados, o benefício de conseguir replicar exatamente os mesmos *inputs*, na mesma sequência temporal é alcançado. Manualmente esta é uma tarefa com uma elevada probabilidade de fracassar, podendo categorizar-se como impraticável [22].

Face a esta tendência, surgiu a necessidade e o interesse por parte da Thales em planear uma estratégia para um dos seus produtos, o *Advanced Passenger Information System* (APIS) 8, de forma a automatizar validações para o *frontend*. O APIS é uma plataforma multifuncional de mensagens informativas aos passageiros, que permite controlar e conceder diversas informações por dispositivos áudio e visuais distribuídos geograficamente.

Foram, portanto, selecionadas diversas tecnologias a serem utilizadas para atingir estes objetivos, nomeadamente o *Robot Framework* e o *SeleniumLibrary* [58, 63].

O desenvolvimento de testes para o *frontend* revela-se um processo complexo devido às múltiplas fontes de *input* assíncronas, ou seja, à possibilidade de interagir com o sistema por diferentes ordens e obter resultados semelhantes. Além disto, a natureza gráfica dos objetos da interface acrescenta um grande número aos resultados que requerem a sua análise [32].

Apesar de a Thales visionar para futuros projetos, desenvolver primeiro os testes de *backend*, automatizá-los e depois de garantido o funcionamento e a estabilidade do *backend* dar seguimento para o *frontend*, o mesmo não se verifica neste projeto. O APIS já está no mercado e só agora é que está a decorrer a transição para a automatização dos testes. Portanto, a prioridade é garantir que o cliente tenha uma interação fluída e sem falhas ao utilizar o sistema.

1.2 Motivação

Ao sujeitar o software produzido a testes é possível detetar defeitos introduzidos no sistema devido a erros humanos, sendo que estes defeitos podem despoletar falhas no software. Estas falhas trazem consequências negativas, como erros no código e falhas nas expetativas dos clientes. A deteção de defeitos no software pode ter diferentes impactos conforme a fase em que é detetado esse defeito. Quanto mais cedo o mesmo é detetado, menor serão os custos, e menor será o tempo necessário para o corrigir.

Através da observação do figura 1.1, é possível concluir que a pior fase para corrigir falhas é após o produto já ter sido disponibilizado ao cliente. No entanto, é nesta fase que se encontra o projeto onde a presente tese se insere. O software para o qual estão a ser desenvolvidos os testes automatizados encontra-se em fase de manutenção, logo, qualquer falha que seja descoberta vai ter um elevado custo ao ser corrigida. O próprio processo de descobrir falhas só por si já é dispendioso, quer financeiramente, quer em tempo, uma

vez que o software não foi testado exaustivamente antes de ser lançado. É importante que para versões futuras, os testes automatizados consigam oferecer uma maior análise de potenciais erros do produto antes deste ser lançado, minimizando os custos.

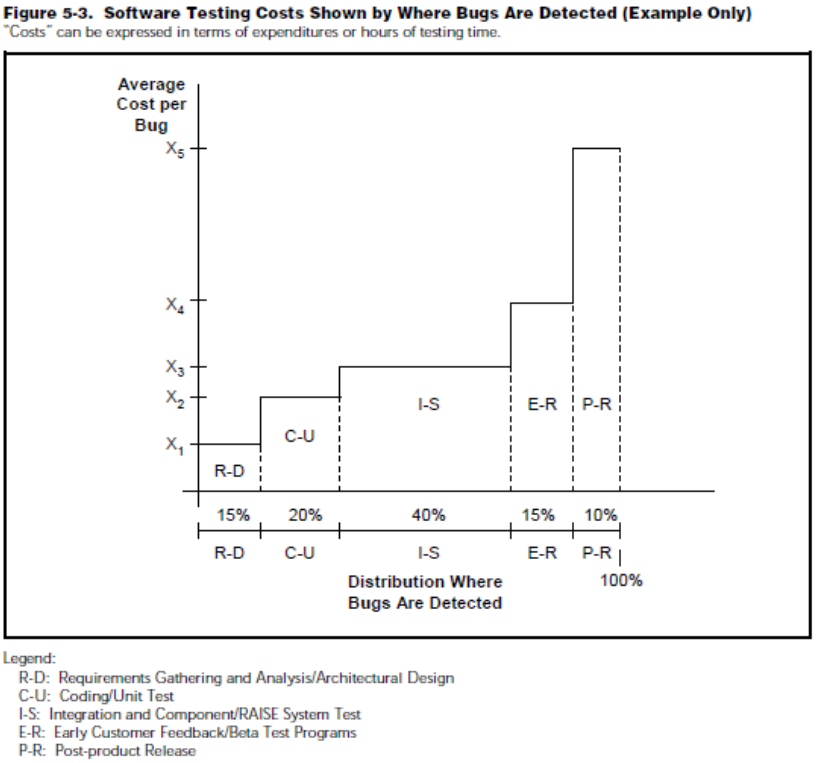


Figura 1.1: Custo de *bug fixes* nas diferentes fases de desenvolvimento [57]

Com o desenvolvimento destes testes e com as novas metodologias *Agile* implementadas na Thales, é esperado que produtos em desenvolvimento, como por exemplo, a versão 9 do APIS, transitem mais facilmente para uma metodologia de integração contínua. Ou seja, um mundo onde se obtém *feedback* o mais rapidamente possível quando ocorrem erros no código, de forma a serem imediatamente identificados e corrigidos [39]. Estes testes estão a ser desenvolvidos com inspiração nas metodologias *Behaviour Driven Development* (BDD), de forma a ser criado um conjunto de testes que partilhem uma linguagem universal para técnicos e não técnicos, desenvolvedores, *testers* e clientes [39]. O resultado final será um sistema interpretável por todas as partes e que responde facilmente às mudanças a que é sujeito.

1.3 Objetivos

A Thales possui uma extensa quantidade de cenários de testes criados e, a cada nova iteração na aplicação é necessário executar manualmente todos esses cenários de teste. É fulcral minimizar o esforço de executar esta quantidade extensa de testes manuais e passar a realizar uma execução automática. Muitas vezes não é possível realizar uma

conversão total dos testes manuais para automatizados. Existem limitações e, como tal, os testes manuais continuam a estar presentes mesmo em ambientes automatizados.

Um dos requisitos da Thales é que os testes automatizados sejam escritos numa linguagem com uma sintaxe em que a sua interpretação seja rápida e eficaz. Como consequência, os casos de teste desenvolvidos vão ser a nova documentação, isto porque foi decidido que é preciso reformular e simplificar o conjunto de cenários de teste do APIS, permitindo que qualquer pessoa dentro e fora da empresa possa facilmente compreender quais os passos necessários para executar uma determinada ação.

O produto APIS 8 está em fase de manutenção, estando no mercado há alguns anos, e continua a ser utilizado por múltiplos clientes espalhados pelo mundo. Consequentemente, o relato de falhas e a necessidade de atualizações é constante. Como tal, continua a existir um intenso suporte e lançamento de novas versões do produto (entregas para o ambiente de produção são realizadas a cada mês e meio). Dito isto, o trabalho desta tese foca-se ainda no desenvolvimento de testes que vão garantir a estabilidade do funcionamento do APIS após alguma atualização. De forma a garantir o maior nível possível de cobertura dos casos de teste, é necessário antes da escrita dos mesmos existir uma análise de quais os testes que são necessários escrever e automatizar. O projeto do APIS 8 não tem presente na elaboração dos seus casos de teste nenhuma técnica de geração de casos de teste, o que levou ao estudo e consequente aplicação destas técnicas no projeto. Está ainda a ser desenvolvida uma nova versão do APIS, onde alguns destes testes automatizados vão ser reutilizados para desenvolver novos testes para esta nova versão.

1.4 Principais contribuições

O desenvolvimento dos casos de teste automatizados foi um processo que necessitou de esforço e tempo, especialmente por não existirem antecedentes de automatização no APIS.

Com a automatização, é esperado que a produtividade da equipa do APIS cresça, permitindo:

- Obter *feedback* da qualidade do produto de uma forma mais eficiente, permitindo uma correção mais rápida e menos dispendiosa, poupando tempo que pode ser investido noutras tarefas;
- Maior foco na execução dos testes manuais;
- Continuar a elaborar mais casos de testes automatizados, de forma a obter uma maior cobertura de todos os casos de teste do APIS 8;
- Elaborar novos tipos de teste, como por exemplo, testes de segurança e desempenho.

Com o desenvolvimento da presente tese, foram produzidos testes automatizados de *frontend* que tiveram por base a documentação original do APIS. Para além disso, sendo

estes testes escritos com inspiração em elementos da metodologia BDD, através da sintaxe do *Gherkin*, é pretendido substituir a documentação original por esta nova originada pelos testes, mais intuitiva e mais fácil de interpretar e executar.

Não foi possível aos *testers* fazerem parte da fase de desenvolvimento do APIS 8 mas, com a conclusão desta transição para a automatização, o desenvolvimento de futuras funcionalidades do APIS já poderá ser elaborada com o acompanhamento dos *testers*. Isto será possível através do desenvolvimento de um ambiente onde novos e antigos *testers* possam passar também a ser integrados no processo de desenvolvimento, uma vez que estes têm um papel fundamental também nesta fase. Devido a eles consegue-se:

- Rever os requisitos pedidos pelo cliente, analisar as *user stories*, concluir quais os requisitos a serem desenvolvidos e quais podem conter defeitos, reduzindo o risco de criar funcionalidades instáveis [38];

- Garantir que, ao trabalhar em conjunto com os desenvolvedores durante a fase de desenvolvimento, os *testers* vão ter um melhor conhecimento de como o sistema funciona e quais as suas limitações e capacidades, facilitando a compreensão do sistema e consequentemente o desenvolvimento de testes para o mesmo [38].

Os testes automatizados foram desenvolvidos para o APIS 8, que já se encontra em funcionamento por exemplo em Doha, no Qatar. No entanto, qualquer outro projeto do APIS, também em funcionamento, vai beneficiar com os testes automatizados, promovendo ainda o interesse e motivando as outras equipas a adotarem os testes já desenvolvidos, e elaborarem eles mesmos, novos testes automatizados para outras funcionalidades.

No início da elaboração da dissertação, a equipa já tinha desenvolvido alguns testes automatizados, no entanto estes eram em pouca quantidade. O desenvolvimento desta tese permitiu acelerar a transição para a automatização.

Foi realizada uma análise, quer dos testes já automatizados, quer dos testes que ainda não tinham sido automatizados. Dos testes já automatizados, foi necessário avaliar se existia a necessidade de serem repartidos em testes mais pequenos. Dos testes não automatizados, foi crucial avaliar se era necessário de todo fazer a transição para a automatização. Em ambos os casos foi necessário verificar se a cobertura dos testes era suficiente e se os mesmos não necessitavam de ser estendidos. Esta reestruturação e reescrita dos casos de teste não seguia nenhuma técnica respetivamente à geração de casos de teste (previamente à elaboração desta tese). Como consequência, foi realizado um estudo das diversas técnicas de geração de casos de teste e da aplicabilidade das mesmas ao projeto do APIS. A aplicação de uma proposta de umas destas técnicas teria certamente um impacto benéfico em futuros desenvolvimentos de casos de testes para outras versões do APIS e outros projetos da Thales, uma vez que de momento os casos de teste são escritos apenas com base nos requisitos levantados pelo cliente.

Externamente à Thales, é expectável que esta tese permita funcionar como um caso de estudo para ajudar outras organizações nas suas transições para a automatização.

1.5 Estrutura

O restante documento encontra-se estruturado da seguinte forma:

- Capítulo 2 - *Background*: este capítulo contextualiza o ambiente em que esta tese foi elaborada. Inicia-se com a apresentação de alguns dados relativos a *software testing* em ambiente empresarial. É dado a conhecer um pouco sobre a Thales e o produto onde a automatização se insere. De seguida, são debatidas metodologias e técnicas adotadas na empresa, assim como as utilizadas na elaboração de testes;

- Capítulo 3 - Trabalho relacionado: neste capítulo são abordados quatro projetos dentro do tema desta dissertação. Três externos à Thales e um interno. O projeto interno é o maior caso de motivação para a automatização dos testes no APIS, uma vez que é uma transição para a automatização de outro produto da Thales;

- Capítulo 4 - Solução proposta e protótipo de automatização: neste capítulo é explicado em detalhe a finalidade do desenvolvimento desta tese. Seguem-se algumas comparações entre as várias tecnologias disponíveis para a automatização e a razão da escolha das mesmas. É por fim apresentado um protótipo, de forma a exemplificar como são automatizados testes usando as ferramentas selecionadas.

- Capítulo 5 - Processo de automatização: neste capítulo são apresentadas as duas técnicas de geração de casos de teste, como estas funcionam e as suas vantagens. Neste capítulo encontra-se ainda descrito o processo de automatização dos testes. Ambos os temas, automatização dos casos de teste e técnicas de geração de casos de teste, são acompanhados de exemplos de como ambos foram aplicados no ambiente do APIS 8.

- Capítulo 6 - Avaliação: neste capítulo são apresentados os resultados obtidos com a elaboração da dissertação acompanhados pela sua respetiva análise.

- Capítulo 7 - Conclusão: esta dissertação encerra neste capítulo, onde é feita uma visão geral do trabalho realizado.

Capítulo 2

Background

Este capítulo faz uma contextualização do ambiente onde a tese foi desenvolvida, abordando vários temas como a empresa e a equipa onde o projeto foi elaborado e o produto para o qual foram desenvolvidos os testes automatizados. Segue-se um conjunto de metodologias e técnicas, algumas já aplicadas na empresa, e outras que se encontram em fase de implementação.

2.1 Importância de *software testing* nas empresas

A área de *software testing* não é nova e, como tal, é importante ter uma perspetiva da necessidade de garantir a qualidade de software.

Para avaliar a qualidade do software nas empresas, os seguintes dados foram analisados [24]:

- 97% dos membros envolvidos em projetos IT têm problemas em detetar erros no software;
- 9 em 10 do dos membros envolvidos em projetos IT sofreram perdas de receita.

Existe uma série de relatórios publicados regularmente, denominado por *CHAOS*. O mesmo é publicado pelo *Standish Group* [75], uma empresa dedicada à pesquisa no mercado da área de *Information Technology* (IT). Tendo como base dois relatórios realizados, um em 2001 e outro em 2009 a diversos projetos IT, é possível retirar alguns dados relevantes [24]. Nomeadamente, o facto que de 2001 para 2009 a situação piorou uma vez que houve uma maior percentagem de projetos cancelados/falhados. Estes valores são apresentados na figura 2.1.

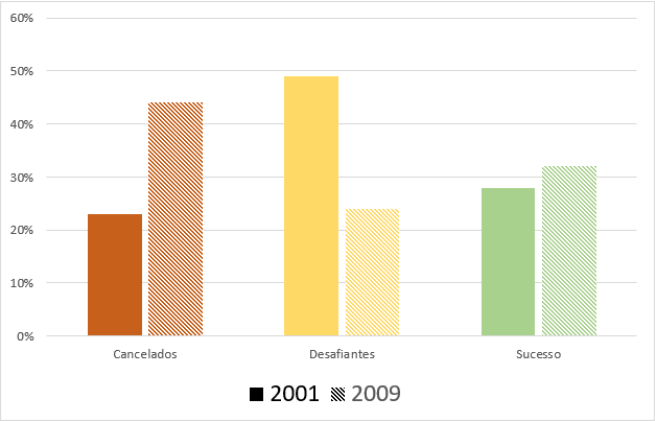


Figura 2.1: Relatório projetos IT 2001 VS 2009

Sucesso significa que os projetos foram entregues a tempo, dentro do orçamento e com as funcionalidades requisitadas a funcionar. Desafiantes no sentido em que alguns excederam o orçamento e outros atrasaram-se, por exemplo.

Devido a resultados como estes, a procura pelo aumento da qualidade de software levou a um maior investimento na área de *software testing*, e como tal, é essencial perceber em que ponto se encontram atualmente as empresas nesta área. A figura 2.2 apresenta os dados do relatório *CHAOS* de 2018. Estes resultados provêm de um estudo a múltiplos projetos ao longo de 5 anos, especificamente entre o período de 2013 e 2017 [8].

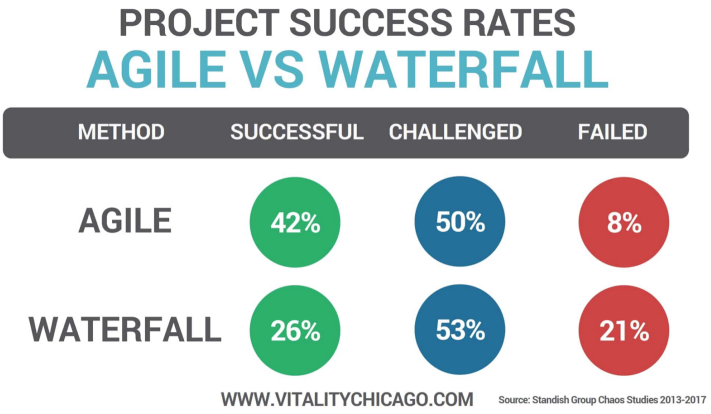


Figura 2.2: Relatório projetos IT 2018 [8]

É possível retirar da figura 2.2 que os projetos *Agile* tiveram mais sucesso que os de *waterfall*, quase o dobro. De acordo com o *Standish Group*, os resultados para todos os projetos demonstraram que os projetos *Agile* têm 60% mais probabilidade de sucesso em comparação a projetos não *Agile*. A acrescentar, os projetos *waterfall* têm 3 vezes mais probabilidade de falhar comparativamente com os projetos *Agile* [8]. *Waterfall* é um ciclo de vida de software onde a sua evolução decorre através de uma sequência de transições ordenadas de uma fase para a outra, por ordem [77]. O processo *Agile* é um ciclo iterativo, este é apresentado na secção 2.3.

Segue-se a análise dos resultados levantados do relatório das práticas de *software testing* realizado pelo ISTQB no período de 2017-2018 [36].

Este relatório teve a participação de mais de 2000 pessoas de 92 países e foi realizado no período de 2015-2016. Houve um aumento na automatização de testes e um aumento nas expectativas de incrementar o orçamento para *software testing* para os 12 meses seguintes. Os dois relatórios revelam que a automatização de testes continua a ser o maior desafio para testes em projetos *Agile*. Este desafio resulta das complexidades da natureza do software em projetos *Agile*, que experienciam uma evolução contínua.

Alguns dos dados mais relevantes retirados do relatório são apresentados de seguida:

- Para um projeto IT, a média de orçamento alocado para testes é de 11-25%;
- O ciclo de desenvolvimento *Agile* é o mais utilizado, com uma adesão de 79%;
- As atividade de teste têm maior foco na deteção de falhas, seguindo-se a necessidade de mostrar o funcionamento correto do sistema e a necessidade de ganhar confiança no sistema produzido;
- O tipo de testes considerados mais importante são os testes funcionais, com um valor de 83%. Este resultado era expectável, uma vez que sem funcionalidades todos os aspetos não-funcionais do sistema tornam-se irrelevantes. Com a crescente valorização pelo cliente, o tópico *user acceptance testing* apresenta-se como o mais relevante;
- A área com maior evolução nas atividade de *testing* é a automatização de testes com 64,4%. A manutenção dos casos de testes também é importante com 41,7%;
- É esperado que um *tester* possua as seguintes capacidades: analisar, planear, desenhar, executar e automatizar testes, reportar falhas e definir estratégias de *testing*;
- A automatização de testes (70.1%), análise de requisitos e *design* de testes são algumas das ferramentas mais utilizadas pelas equipas de *testers*;
- O tópico que apresenta uma maior tendência, relativamente a profissões de *software testing* no futuro, é a automatização de testes.

Através da análise deste relatório é possível observar que a área onde esta tese se insere está em crescimento. É uma área onde se investe cada vez mais por ser altamente valorizável para o ciclo de desenvolvimento do software. As empresas cada vez mais valorizam a importância da qualidade do software e a necessidade de testar o mesmo.

2.2 Thales, a empresa, a equipa e o produto

2.2.1 Grupo Thales

O Grupo Thales é uma companhia multinacional focada no desenvolvimento de sistemas eletrónicos para cinco setores: aeroespacial, espaço, defesa e segurança, identidade digital e transporte terrestre. A Thales surgiu em Portugal em 1990 e destacou-se como um líder de soluções integradas para transportes. Hoje permanece como um dos grandes nomes no segmento dos transportes, proporcionando produtos e soluções para sistemas de segurança e informação ao passageiro [71–73].

2.2.2 Produto APIS

A Thales oferece uma plataforma multifuncional de mensagens informativas aos passageiros denominada por *Advanced Passenger Information System* (APIS). Tem o objetivo de controlar e oferecer diversas informações aos passageiros por dispositivos áudio e visuais distribuídos geograficamente. É um produto utilizado internacionalmente, em concreto, a versão 8 para a qual foram desenvolvidos os testes automatizados, está a ser utilizado em Portugal, Qatar, Dubai, Polónia, Taiwan, Senegal e Austrália. A *Human-Machine Interface* (HMI) do APIS 8, na qual o trabalho desta tese incide, foi escrita nas linguagens *JavaScript*, *CSS* e *HTML5*. Algumas das funcionalidades de destaque que o APIS oferece são apresentadas em baixo:

- *Automatic messaging*: as mensagens são enviadas automaticamente através de um conjunto de pré-condições definidas. Estas pré-condições controlam quando e onde as mensagens são geradas e qual o conteúdo a ser transmitido;

- *Multi-lingual text*, *image and audio messaging*: o APIS tem suporte de mensagens áudio, texto e imagem em diversas línguas. As mensagens de texto podem ser automaticamente convertidas em áudio através de um software de conversão. As mensagens de áudio podem ser criadas através de uma funcionalidade de gravação incluída no APIS ou através da importação de ficheiros de áudio já existentes. As imagens são manipuladas externamente;

- *Scheduled and repetitive messaging*: as mensagens podem ser enviadas de acordo com alarmes e padrões de repetição. Cada repetição de uma mensagem pode ser controlada como uma entidade separada sem afetar a mensagem original.

O APIS é um sistema modular e escalável conectado através de uma rede TCP/IP. Habitualmente o sistema encontra-se nas seguintes localizações:

- *Operational Control Centre* (OCC): o OCC é composto pelos operadores e servidores principais que controlam todo o sistema;

- *Manned Stations*: a estas estações é fornecido o *APIS Station Server*. Este assume a responsabilidade no caso da comunicação entre a estação e o OCC se encontrar em baixo;

- Veículos: neles está incorporado o *APIS OnBoard*, que gera mensagens de trânsito da viagem autonomamente. Os operadores dos veículos usam o *APIS OnBoard* para comunicarem com os passageiros.

A informação ao passageiro é transmitida através de uma interface áudio/visual entre o operador do transporte e os passageiros, oferecendo informação fidedigna da viagem e contribuindo para um maior conforto e segurança aos clientes.

O APIS é uma solução que usa uma plataforma de software, permitindo a criação, distribuição e apresentação de informação através de um sistema em tempo real, que integra e sincroniza anúncios públicos e informações visuais e áudio. Estas informações são passadas ao público por canais de distribuição como painéis nas estações e *OnBoard*. A arquitetura do sistema permite simplificar imenso a informação, gerindo a mesma por diversas redes de transportes através da unificação numa única plataforma e interface, funcionalidades vindas de todos os modos de transporte. Esta é desenhada como uma plataforma distribuída, transacional, segura e portável. Fornece todos os serviços para controlar informação proveniente de *inputs* heterogéneos (encaminhamento e controlo do comboio, tabelas de horários, etc) e coordena a entrega de informação por vários meios (visual e áudio) e *media* (*passenger displays*, altifalantes, etc) [6, 45].

2.2.3 Equipa de testes IVVQ

A equipa de testes onde a dissertação foi desenvolvida é denominada por *Integration, Verification, Validation and Qualification* (IVVQ). A estratégia do IVVQ adotada para o APIS segue as seguintes fases: integração, verificação e validação, opcionalmente existe ainda a qualificação. Todas estas fases são sujeitas a iterações, com escopos diferentes, completude de atividades diferentes assim como a documentação. Este processo resulta da natureza do *design*, desenvolvimento e *testing* do sistema do APIS [49].

Sendo o APIS um sistema de natureza iterativa, é necessário a equipa realizar também um plano iterativo que acompanhe as diferentes fases de desenvolvimento do sistema. A figura 2.3 descreve as atividades a serem executadas como parte do plano iterativo.

Esta equipa guia-se por um plano denominado por *Integration, Verification, Validation and Qualification Plan* (IVVQP). É uma equipa multidisciplinar, sendo esperado da parte da mesma: a elaboração de casos de teste e sua respetiva execução. Trabalham com o propósito de oferecer ao cliente um software com o máximo de qualidade, fortificando a satisfação dos utilizadores [74].

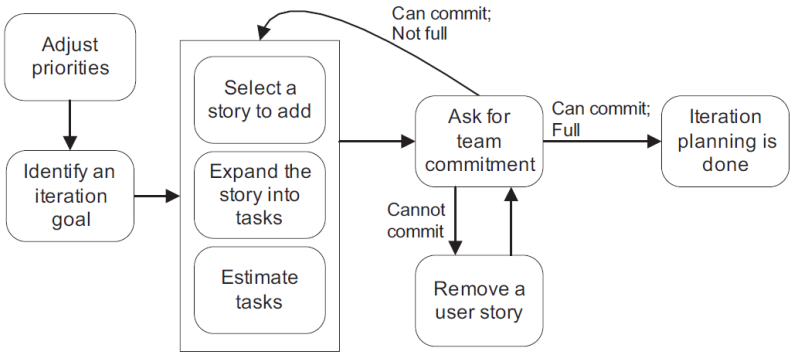


Figura 2.3: Atividades do *Commitment-driven iteration planning** [66]

Além do IVVQP existem outros documentos mais específicos do ponto de vista dos procedimentos e resultados. O *Software Test Plan* (STP) define qual a aproximação a ter para testar as diversas atividades em diferentes ambientes, níveis e classes. Ainda presente, existe o *Software Test Description* (STD), que surge com base nos requisitos e nas *user stories*. Este documento especifica as *suites* de testes e os seus procedimentos e metodologias correspondentes de forma a executar estes testes. Após cada execução dos testes é gerado o *Software Test Report* (STR), que indica quais as *suites* que passaram, quais falharam, permitindo tirar as conclusões desejadas [45].

2.2.4 Validação do APIS 8

Em seguida é demonstrado como é feito o processo de construção de casos de teste e como o sistema é verificado e validado na Thales. A figura 2.4 representa as etapas que compõem o processo de desenho, desenvolvimento e qualificação da solução proposta pela Thales (*Design, Develop and Qualify the Solution* (DDQS)), no âmbito deste projeto, a solução é o APIS 8.

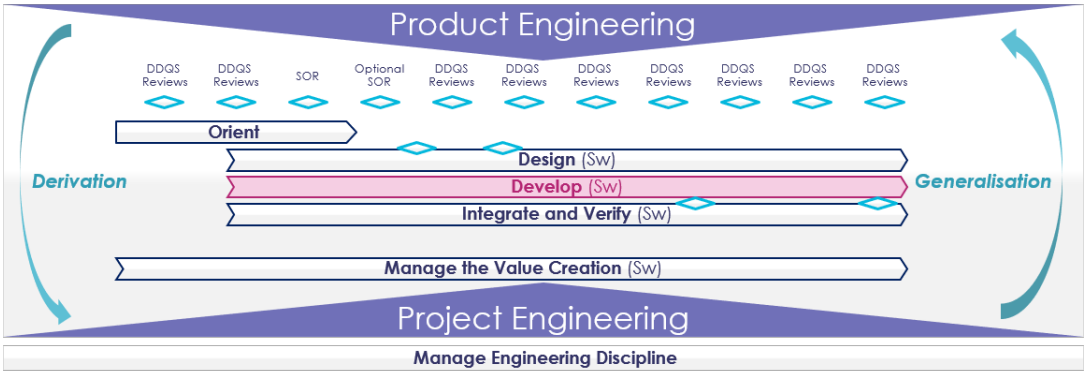


Figura 2.4: *Design, Develop and Qualify the Solution* (DDQS)* [70]

É possível verificar na figura 2.4 que o processo de validação não se encontra presente. Este fator é um dos objetivos que esta tese visa mitigar oferecendo técnicas de geração de

casos de teste que passem a ser integradas na fase de validação do produto final. A Thales reconhece a necessidade de implementar a fase de validação nos seus produtos e, como é possível verificar na figura 2.5, a mesma já pretende vir a fazer parte do DDQS. Esta figura foi retirada de um documento da Thales onde são descritas as fases que a equipa IVVQ que vai estar envolvida no desenvolvimento do APIS 9 deve realizar. Uma dessas fases é a validação.

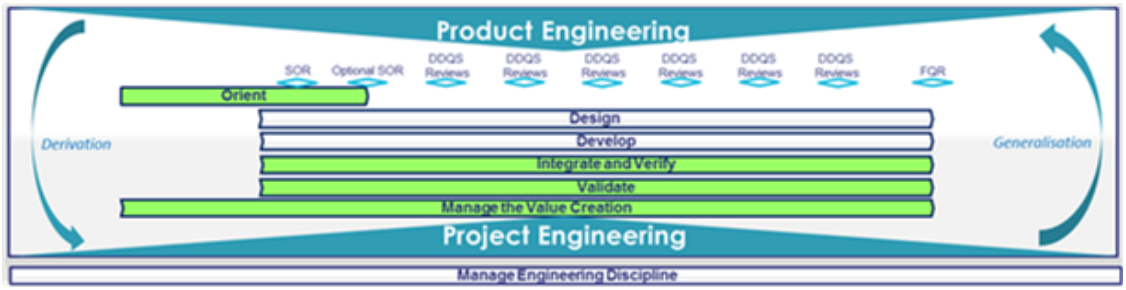


Figura 2.5: *Design, Develop and Qualify the Solution* (DDQS) do APIS 9* [49]

A fase de validação visa a elaboração de um conjunto de testes que sejam relevantes para garantir o bom funcionamento do sistema produzido. Como tal, segue-se o estudo de como é realizado o processo de desenvolvimento destes testes através da análise do artigo intitulado "*Software Test Process, Testing Types and Techniques*" [31]. É através deste processo que os requisitos do sistema e as suas componentes são exercitadas e avaliadas manualmente, ou através de ferramentas de automatização. Desta forma descobre-se se o sistema vai de encontro aos requisitos especificados e quais as diferenças entre o expectado e os resultados atuais [31].

Existem três fases fundamentais neste processo, são elas: análise dos testes, planeamento e preparação dos testes e, por fim, a execução dos mesmos.

A figura 2.6 é o mapeamento das diferentes fases de desenvolvimento, assim como dos diferentes tipos de testes que as acompanham. De momento, na Thales não estão a ser elaborados testes de desempenho (*performance testing*) nem de segurança (*security testing*). Estes últimos estão de momento a ser planeados. A figura 2.7 representa, portanto, uma melhor aproximação às fases e tipos de teste presentes na Thales.

A fase de preparação dos testes (*test preparation*) podia ficar omissa nesta figura, uma vez que, de momento, os casos de teste apenas são gerados através da aplicação da técnica *exploratory testing* (secção 2.14). É precisamente nesta fase que a elaboração desta tese visou melhorar o método de como os testes são preparados e criados, através da aplicação de duas técnicas de geração de casos de teste. As mesmas são apresentadas na secção 5.1.

*© *All rights reserved. Passing on and copying of this document or use and communication of its contents are not permitted without written authorization from Thales Portugal, S.A*

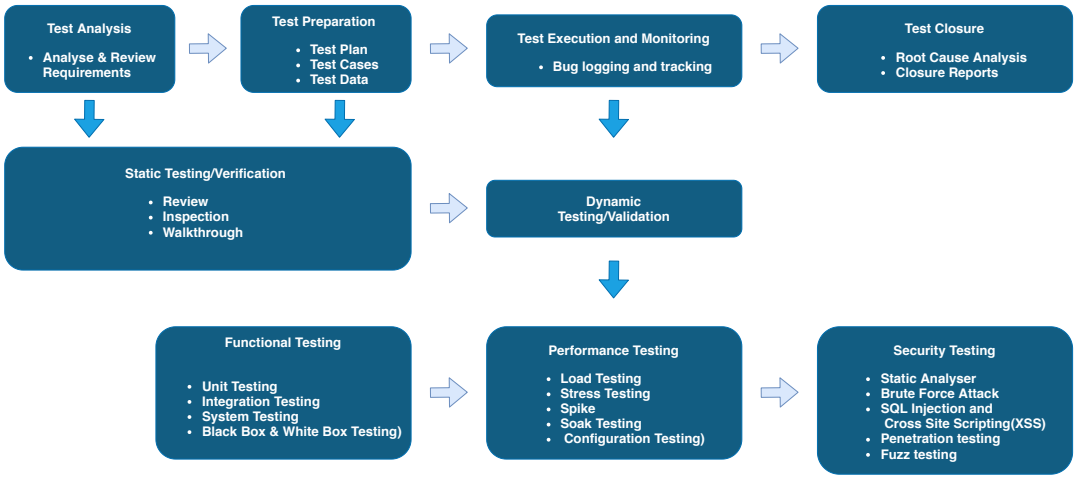


Figura 2.6: Mapeamento dos tipos e das fases de desenvolvimento dos testes [31]

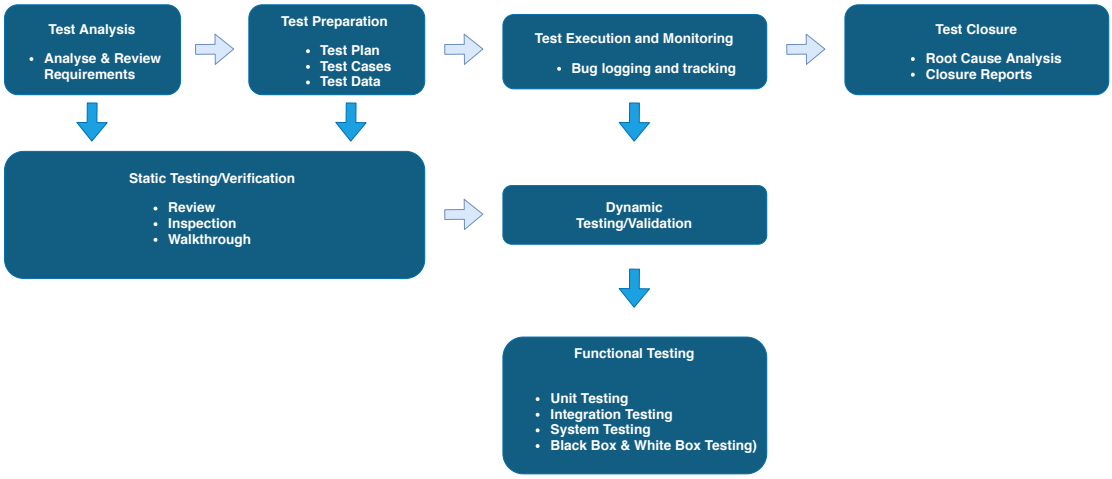


Figura 2.7: Mapeamento das fases de desenvolvimento dos testes e dos tipos de testes (modificado)

2.3 *Agile*

O *Agile* é um conjunto de metodologias de *project management* e de desenvolvimento de software baseada em iterações. Ou seja, o objetivo é produzir pequenos incrementos de trabalho, avaliar de forma contínua os requisitos e os resultados, criando um sistema que responde facilmente e rapidamente a mudanças [7] .

Através da aplicação de técnicas de desenvolvimento *Agile*, usando conceitos como o *Test-Driven Development* (TDD), *Behaviour Driven Development* (BDD) e *Acceptance Test-Driven Development* (ATDD) é possível produzir um software com menos erros e que vai mais de encontro com as necessidades do cliente.

As pessoas integradas neste ambiente de desenvolvimento *Agile* que executam os testes são chamados de *agile testers*. Devem compreender diferentes pontos de vista, tanto dos desenvolvedores, como do cliente, assegurando a qualidade e o valor de mercado do

software produzido [27].

No caso do projeto do APIS, este não foi desenvolvido de acordo com a metodologia *Agile*, resultando num modelo de testes *waterfall*. As equipas que usam este modelo acabam por verificar que a quantidade de testes começa a crescer exponencialmente, dificultando o acompanhamento dos testes por parte do *Quality Assurance* (QA) [21]. Com a transição para o modelo *Agile*, a automatização de testes faz sentido ser aplicada neste projeto, uma vez que os testes automatizados são um dos pilares do modelo *Agile* [18].

A abordagem *Agile* foi posta em prática na Thales Portugal há cerca de um ano. Esta prática ainda não teve tempo de amadurecer e muitas áreas na empresa ainda não aplicam a mesma. Isto deve-se ao facto dos produtos da Thales serem bastante robustos, com um nível de risco entre o médio e o alto e, como tal, nem todos os processos foram adaptados para a prática *Agile*. As questões de segurança são a razão fundamental do processo de transição ser mais lento, sendo que ainda estão a decorrer estudos e experiências, de forma a compreender melhor onde é ou não possível utilizar a prática *Agile*.

2.4 Metodologias

2.4.1 TDD

O *Test-Driven Development* (TDD) é um método de desenvolvimento de software baseado nos seguintes passos:

- Automatizar casos de teste em ciclos de desenvolvimento;
- Construir e integrar pequenos pedaços de código;
- Executar os testes;
- Corrigir erros;
- Reestruturar código;
- Correr novamente o teste para verificar se os mesmos passam.

Esta metodologia decorre em pequenas iterações até uma componente estar completamente construída e todos os testes desta componente gerarem um *output* satisfatório. Este é um exemplo de uma abordagem *test-first* [38]. Este método vê o *design* do software como um processo contínuo, onde cada iteração é constituída por pequenas mudanças na produção do código, oferecendo desta maneira oportunidades aos desenvolvedores para irem aprimorando o *design*.

2.4.2 ATDD

O *Acceptance Test-Driven Development* (ATDD) tem como objetivo coordenar projetos de software de forma a que o produto final esteja de acordo com as necessidades do

utilizador final. Para alcançar este objetivo, a colaboração de vários membros envolvidos no projeto incluindo clientes, desenvolvedores e *testers*, é essencial para criar testes de aceitação previamente à implementação das funcionalidades do sistema. Estes testes são representativos do ponto de vista dos utilizadores finais e servem como forma de descrever quais os requisitos funcionais esperados do sistema. Estes são um meio de confirmar que o sistema se comporta como esperado, por norma são automatizados [2].

Em suma, os pilares do ATDD são o *feedback* dos clientes e a comunicação entre a equipa.

BDD e ATDD são focados no cliente enquanto que o TDD tem foco nos desenvolvedores [39].

2.4.3 BDD

O *Behaviour Driven Development* (BDD) surgiu como uma evolução das duas metodologias anteriores, o TDD e o ATDD.

Os problemas do TDD surgiram quando programadores começaram a relatar que não sabiam por onde começar, o que era necessário testar e o que não era, quantos testes deviam ser executados de cada vez e como perceber o porquê dos testes falharem. Como resposta a estes problemas, North criou o BDD [35]. Com o BDD, programadores em equipas *Agile* passaram a conseguir aplicar boas técnicas de como produzir testes e código, devido a esta metodologia ser mais acessível, eficiente e independente do tipo de equipa, graças à sua linguagem universal.

A diferença entre o ATDD e o BDD é o facto do último promover uma linguagem utilizada pelos desenvolvedores durante o desenvolvimento do software, que por usa vez, também é facilmente interpretável pelos clientes. Esta linguagem é baseada em exemplos reais e comportamentos, ao invés de testes e critérios de aceitação [9].

Com a evolução do BDD, a análise e a automatização de testes de aceitação assumiram-se como os seus pontos fulcrais [28].

Consultando o ISTQB [39], o BDD é uma metodologia na qual desenvolvedores, *testers* e clientes trabalham em conjunto de forma a analisarem os requisitos do software do sistema. Formulam os mesmos usando uma linguagem que seja interpretável por todas as partes, técnicos e não técnicos. Esta linguagem omite os detalhes da implementação e as especificações do software. O BDD consiste na execução de múltiplos cenários permitindo uma análise de como se comporta o sistema [39].

Esta metodologia é constituída por um conjunto de passos, enumerados a seguir, que facilitam a sua aplicação:

1. Criar *user stories* em conjunto com todas as partes, clientes, desenvolvedores e *testers*;
2. Enunciar as *users stories* como cenários executáveis e comportamentos verificáveis;
3. Executar os cenários descritos de forma a verificar como se comportam.

Esta metodologia vai ser implementada através da sintaxe do *Gherkin* (ver secção 2.12.2).

O código apresentado na listagem 2.1 foi adaptado com base num artigo sobre BDD [14]. Através da análise das listagens é percetível a diferença de interpretação entre os dois testes. O teste de aceitação sem BDD (2.1) é muito mais difícil de ser interpretado por uma pessoa que não esteja familiarizada com a área da informática. Ao analisar o teste de aceitação com BDD (2.2), é fácil de observar que os passos são escritos numa linguagem mais próxima da língua humana. No exemplo apresentado, a frase referente a comprar uma televisão é muito diferente de uma listagem para a outra. Na listagem 2.1, a frase é *"TillScreen.addItem('Panatachi Television 3X')"*. Na listagem 2.2, a frase é *"Given that a customer purchased a Panatachi Television 3X"*.

Listagem 2.1: Teste de aceitação sem BDD [14]

```
1 TillScreen.addItem("Panatachi␣Television␣3X").pay("2000").
2 with(new CreditCard("2345123478902468")).done();
3 int originalNoOfTelevisionsInStock = StockScreen
4 .count("Panatachi␣Television␣3X");
5 Money originalBalance = FinanceScreen.getBalance();
```

Listagem 2.2: Teste de aceitação com BDD [14]

```
1 Given that a customer purchased a Panatachi Television 3X
2 and a Panatachi Television 3X costs $2000
3 and he paid with credit card number 2345123478902468
4 When I search for the most recent bill with credit card number 2345123478902468
5 and I refund the Panatachi Television 3X
6 Then the stock of Panatachi Television 3X should be unchanged
7 and we should have $2000 less.
```

2.5 *Continuous Integration*

Integração contínua é uma prática de desenvolvimento de software que assenta na integração regular do trabalho produzido por várias equipas. Com o desenvolvimento de um ambiente automatizado consegue-se testar cada integração e rapidamente detetar os erros que podem ocorrer em cada uma [23]. Na sua essência, o processo de integração contínua assenta na existência de um repositório comum onde as equipas realizam frequentemente *commits* para o mesmo. Posteriormente, uma ferramenta como o *Jenkins* vai compilar o código e, no caso da compilação ser concluída com sucesso, segue-se a execução dos testes automatizados. No âmbito desta tese, estes foram os testes desenvolvidos. No caso do resultado dos testes não apresentar nenhum erro, o produto revela-se como estável e pronto para, ou ser entregue ao cliente, ou dar seguimento ao seu desenvolvimento [68].

De momento, esta prática não se encontra presente no produto do APIS, mas com o investimento na automatização de testes a equipa espera conseguir integrar a mesma. O

objetivo é conseguir ter um *pipeline* de diversos tipos de teste, tais como testes à *Graphical User Interface* (GUI) e testes unitários, por exemplo, a serem corridos a cada integração que a equipa faz e consequentemente é feito o *deploy*. Desta forma, garante-se que toda a equipa está sempre a trabalhar com a última versão estável do produto.

2.6 *Continuous Testing*

Metodologia onde o objetivo é testar frequentemente, o mais cedo possível e automatizar os testes, obtendo assim um *feedback* dos riscos associados ao lançamento do sistema a ser desenvolvido. Permite que os desenvolvedores tenham conhecimento mais cedo dos erros e rapidamente os consigam resolver [39].

De momento o produto do APIS enquadra-se no processo de transição para o *continuous testing*. Previamente a equipa trabalhava na mesma versão que era lançada a cada mês e meio e no final desse tempo, eram executados os testes dessa versão. Agora já existem alguns testes automatizados e, como tal, estes podem ser executados diariamente.

2.7 *Black Box Testing*

Na técnica de *black box testing*, a pessoa que se encontra a testar o software não possui qualquer conhecimento/acesso ao código interno, sendo-lhe dispensada a necessidade do conhecimento dos mecanismos internos do sistema. O seu foco está em selecionar *inputs*, executar condições e analisar o *output* resultante. A escolha dos casos de teste é baseada totalmente nos requisitos ou especificações de *design* da entidade de software a ser testada [48].

Esta técnica apresenta algumas vantagens como por exemplo, o facto de ser não intrusiva, por não necessitar de acesso ao código fonte do sistema [3]. Ajuda ainda a expor quaisquer ambiguidades ou inconsistências nas especificações dos requisitos [48].

2.8 *Keyword-driven testing*

Uma *keyword* representa uma ação a ser executada em *keyword-driven testing*. Ou seja, é um conjunto de palavras escritas de uma forma universalmente compreensível e que indicam as operações desejadas a executar. Quando o ambiente a ser testado é a interface visual, estas *keywords* representam ações de utilizador, como por exemplo, carregar em botões, escrever texto, confirmar que certos elementos estão visíveis, etc. [54].

Ao utilizar uma aproximação a esta metodologia, os *testers* podem utilizar *keywords* que estejam relacionadas com os domínios de negócio ou de conhecimento onde estes se encontram [25].

Testes que têm por base *keywords* oferecem boa legibilidade, percetibilidade e manutenção dos *scripts* dos testes automatizados e manuais [37].

Através da utilização de *keywords* é possível abstrair a implementação dos testes através de ações de alto nível [54]. Por exemplo, a *keyword* de alto nível *"I modify output group 'Mixed Group'"* retirada de um caso de teste do APIS 8, é constituída por 5 *keywords* de baixo nível, como é possível observar na figura 2.8.

```
I modify output group "${group}"
    [Documentation]   Modifies output group.
        Click Element    ${current_page['edit_output_group']}
        I write "Mixed Output Group Modified" in "Output Group Name" field
        Click Button          ${current_page['add_all_button']}
        Click Element         ${current_page['save_group']}
        Logout APIS
```

Figura 2.8: *Keywords* de baixo nível*

Como é possível confirmar, as *keywords* apresentadas são bastante compreensíveis das ações que as mesmas representam. Através da utilização de *keyword-driven testing* é possível estabelecer uma ligação com a metodologia BDD uma vez que são construidos casos de teste facilmente compreensíveis.

Se for produzida uma sequência de *keywords* criam-se casos de teste. Dentro do escopo desta dissertação o objetivo em utilizar estas *keywords* é criar casos de teste que sejam facilmente interpretáveis por técnicos e não técnicos, como tal, os casos de teste possuem apenas *keywords* de alto nível.

Para o processo de criação de *keywords* e respetiva construção de casos de teste com as mesmas, é importante haver uma constante criação e manutenção das *keywords* criadas [37]. Consequentemente, o custo de adicionar novos testes é bastante reduzido. Como resultado, são gerados casos de teste que usufruem de uma fácil manutenção, boa legibilidade e desenvolvimento intuitivo.

2.9 Espetro de testes

No mundo dos testes existem diversos níveis e tipos onde estes se inserem, independentemente destes serem manuais ou automáticos.

Existem 4 níveis de testes [30]:

- *Acceptance testing*: visam obter a aprovação por parte dos clientes. Existem para conseguir atingir um determinado nível de confiança com o sistema. A procura por falhas não é a prioridade, o objetivo é perceber se o número de falhas que existe poderá comprometer a confiança do cliente para futuras entregas do sistema. Têm como foco a análise dos comportamentos e capacidades do sistema desenvolvido.

  Estes testes focam-se em três objetivos [38]: assegurar qualidade do sistema; validar a completude do sistema e que este funcione como esperado; verificar que tanto os comportamentos funcionais como os não funcionais do sistema estão implementados. Utilizam a técnica de *black box testing*;

- *System testing*: avaliação de como todo o sistema funciona. Processa transações *end-to-end*, desde a instalação do software à sua operação. Pretende detetar falhas no software produzido de forma a garantir que vai de encontro com os requisitos e especificações e que o mesmo esteja pronto para ser distribuído para os clientes. Tipicamente estes testes recorrem à técnica *black box testing*;

- *Integration testing*: foca-se nas interfaces entre as diferentes componentes e entre as diferentes partes do software e do sistema. Procura falhas nas interfaces e trocas entre componentes, ou seja, compreender se, ao haver interligação entre os diferentes elementos possam existir falhas;

- *Unit testing*: consiste em testar, por exemplo, componentes, funções, programas, bases de dados e verificar como estes elementos se comportam face aos requisitos e especificações. Procuram descobrir falhas nestes elementos. Tipicamente estes testes necessitam de acesso ao código.

O escopo da dissertação foca-se em dois níveis de testes, sendo estes *acceptance testing* e *system testing*.

Segue-se agora a apresentação dos diferentes tipos de teste existentes, são eles [30]:

- *Functional tests*: focam-se nas funções do software, o que estas fazem, os serviços que fornecem e os requisitos cobertos;
- *Non-Functional tests*: enquanto que os *functional tests* se focam nos serviços oferecidos pelo software produzido, estes testes focam-se na maneira como esses serviços são oferecidos. Avaliam a usabilidade, confiabilidade, eficiência e manutenção do sistema;
- *Structural tests*: permitem determinar a cobertura existente do sistema. Estes testes não foram desenvolvidos no âmbito desta tese;
- *Regression tests*: procuram expor potenciais erros que foram introduzidos aquando do desenvolvimento de novo código. Estes testes são essenciais durante a manutenção do sistema.

O produto do APIS 8 encontra-se em fase de manutenção, portanto, é necessário garantir que as mudanças ao sistema não provoquem falhas noutras funcionalidades, desestabilizando-o. Estas mudanças podem afetar a qualidade do software quer seja a nível funcional, como não funcional, afetando por exemplo a rapidez, compatibilidade e segurança do sistema. Surge então a necessidade de desenvolver *regression tests* de forma a garantir um bom comportamento funcional e não funcional [38].

2.10 Automatização de testes

De acordo com o ISTQB [38], o processo de testar não consiste apenas em executar testes e analisar os seus resultados. Este é um processo bastante mais complexo que envolve diversos passos como o planeamento dos testes, análise, *design*, implementação dos testes, a criação de *reports* do progresso dos testes e os respetivos resultados e analisar a qualidade de um objeto de teste. Envolve ainda a revisão dos requisitos e código. Para além de executar toda esta verificação, este processo engloba ainda a validação do sistema, garantindo que o sistema vai de encontro às necessidades do cliente [38].

De forma a realizar uma boa transição dos testes manuais para automatizados, é essencial fazer uma análise inicial, identificando quais os elementos a testar e se faz sentido estes serem automatizados. Definir quais as componentes que devem e não devem fazer

parte da automatização leva a uma diminuição da complexidade da tarefa de automatizar os testes [29].

O facto dos testes automatizados serem desenvolvidos sempre com a consciência da necessidade de preservação do sistema é essencial, especialmente na automatização do *frontend*, uma vez que as interfaces gráficas são compostas por diversos elementos complexos que são constantemente redesenhados durante o processo de desenvolvimento [56].

Após a conclusão de uma implementação totalmente automatizada de testes, o custo de correr estes testes automáticos é menor, em comparação com a execução manual, tornando esta uma solução mais económica. No entanto, para atingir esta vantagem económica, é necessário um maior investimento na fase inicial e na manutenção dos *scripts* de teste, em comparação com os testes manuais. A fase de manutenção é essencial ser levada em consideração por quem está a desenvolver os testes automatizados. Este profissional denomina-se por *test automator*, podendo ser um *tester* ou não [22].

De forma desenvolver testes automatizados com qualidade, é importante ter em mente os seguintes fatores [37]:

- *Test Automation Arquitecture (TAA)*: alinhada com a arquitetura do software. Deve ser claro quais os requisitos que o sistema deve e não deve suportar;
- *System Under Test (SUT) Testability*: quando é testada a *Graphical User Interface* (GUI), o SUT deve separar ao máximo as interações e os dados da aparência da interface gráfica;
- *Test automation strategy*: estratégia que permite garantir a manutenção e consistência do SUT, decidindo quais as zonas onde aplicar a automatização tendo em consideração os custos, benefícios e riscos existentes;
- *Test Automation Framework (TAF)*: de forma a desenvolver facilmente e rapidamente testes automatizados é essencial escolher uma ferramenta que seja intuitiva e bem documentada.

Uma *Test Automation Solution* (TAS) consiste em duas peças: ambiente de teste e *suites* de teste. A TAF é usada para criar TAS's através do uso das suas ferramentas e bibliotecas.

A *general Test Automation Arquitecture* (gTAA), arquitetura dos testes automatizados, é estruturada em 4 camadas [37]:

- *Test generation*: *design* de casos de teste manuais e automáticos. É o que providencia os meios para criar estes casos de teste;
- *Test definition*: definição e implementação das *suites* de teste e/ou casos de teste. Permite estabelecer testes de alto e baixo nível;
- *Test execution*: suporta a execução dos casos de teste e gera os respetivos *logs* através da utilização de uma ferramenta de execução de testes que permite executar

automaticamente estes casos de teste e que gera duas componentes de *logging* e *reporting*;

- *Test adaptation*: providencia o código que permite adaptar os testes automatizados a outros elementos ou interfaces do SUT.

2.10.1 Prós e contras da automatização de testes

2.10.1.1 Prós

Existem várias vantagens em automatizar os testes, tais como [22, 37]:

- A possibilidade de correr a pilha de testes previamente criada em novas versões do software. Uma vez que os testes já foram criados e automatizados, é possível saber se a nova versão do produto continua funcional através da execução dos mesmos;
- Ao automatizar os testes é diminuído o tempo de execução que estes iriam tomar se fossem feitos manualmente. Como resultado, é obtida uma velocidade de lançamento superior;
- Certos testes revelam-se inviáveis de realizar manualmente, por exemplo, se for necessário simular múltiplos *inputs*;
- Alivia os *testers*, uma vez que os dispensa de executarem testes simples e repetitivos, permitindo que estes tenham um maior foco nos testes que têm obrigatoriamente de ser executados manualmente;
- Garante consistência para testes que vão ser executados múltiplas vezes e onde é essencial garantir que o *input* é sempre o mesmo;
- *Feedback* mais rápido relativamente à qualidade do software.

2.10.1.2 Contras

Existem várias desvantagens em automatizar os testes, tais como [22, 37]:

- Más práticas de *testing*, como o mau planeamento dos casos de teste ou uma escolha inadequada da ferramenta de automatização, pode por exemplo, levar a prejuízos;

- Cada iteração produzida para o sistema vai levar à criação de mais testes, originando uma necessidade de reestruturação dos testes para tentar evitar que estes não aumentem em dimensão, mas sim que englobem mais casos. Consequentemente, é necessário mais tempo de desenvolvimento para concluir os testes e, como consequência, mais tempo para executar e verificar os resultados dos testes;

- Se os testes não forem elaborados com a mentalidade que estes têm de passar por vários processos de manutenção, então é uma questão de tempo até a automatização dos testes ter que ser abandonada e os custos de manutenção começarem a subir;

- Por vezes, como os testes automatizados requerem maior esforço na sua manutenção e elaboração comparativamente com os testes manuais, estes acabam por restringir as mudanças planeadas para o software devido a possíveis custos económicos. Isto se os testes não forem planeados com a manutenção do produto em mente, caso contrário, isto nunca será um problema pois a manutenção e a reestruturação dos testes é uma fase normal do ciclo de vida de desenvolvimento.

2.11 Como desenvolver bons casos de teste?

Um caso de teste é sempre composto por um conjunto de *inputs* que direcionam o sistema para um determinado estado, de onde o sistema vai gerar um *output*. Se esse *output* for o expectável, o teste é dado como passado, caso contrário, como chumbado [55]. Mas, quais são as componentes que compõem um bom caso de teste? Existem quatro atributos que descrevem a qualidade de um caso de teste e que devem ser levados sempre em conta na elaboração dos mesmos. São eles [22]:

- Eficácia: na deteção de erros;

- Exemplar: um teste exemplar deve testar mais do que uma coisa, de forma a reduzir a quantidade de casos de teste necessários;

- Economia: o quão económico um caso de teste é ao executar, analisar e fazer *debug*;

- Evolução: qual a quantidade de manutenção que o caso de teste vai necessitar conforme o lançamento de novas versões/funcionalidades.

2.12 Ferramentas de automatização

Uma *framework* de automatização de testes deve conter um conjunto de ferramentas de software e serviços que auxiliam os *testers* ao desenvolverem casos de teste e consequentemente na execução dos mesmos. Com uma *framework* destas, os *testers* podem focar-se inteiramente em testar o produto, ao invés de estarem a desenvolver uma infraestrutura com capacidade para suportar um ambiente onde se possam desenvolver testes.

2.12.1 *Robot Framework*

Como tal, a *framework* selecionada foi o *Robot Framework*. Esta é uma *framework* de *acceptance test automation*. Baseia-se na linguagem de programação *Python* e, como tal, preenche as necessidades de multiplataforma, uma vez que pode ser implementada em qualquer plataforma, independentemente da sua dimensão. É ainda uma *framework keyword-driven test automation* fazendo uso dos dois tipos de bibliotecas quando se está a trabalhar nesta *framework*, bibliotecas internas e externas. As internas vêm incorporadas com o *Robot Framework*, enquanto que as externas encontram-se em pacotes separados. Adicionalmente, é possível aos *testers* criarem bibliotecas. As bibliotecas externas são compostas por outro tipo de *keywords*, sendo estas denominadas de *user keywords*.

Uma das bibliotecas mais utilizadas em conjunto com a *framework* é o *Selenium Library*. Sendo uma *framework* orientada por *keywords*, esta vai de encontro com a metodologia BDD, uma vez que estas *keywords* são facilmente interpretáveis por qualquer elemento constituinte no desenvolvimento de testes (clientes, *testers* e desenvolvedores). Como esta *framework* se encontra interligada entre os testes desenvolvidos e o software que está a ser testado, a mesma é benéfica tanto para desenvolvedores como para *testers*.

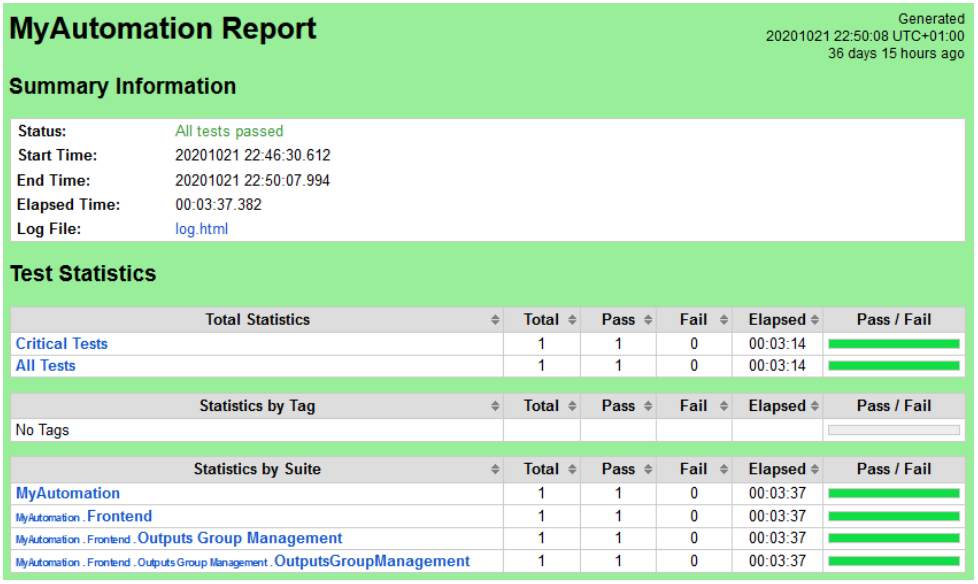
O seu funcionamento é bastante simples. Consiste em processar a informação que compõe os casos de teste, executar os casos de teste e gerar *logs* e *reports*.

A natureza flexível do *Robot Framework* permite uma fácil adaptação e, como é independente do *backend*, revela-se como uma *framework* essencial para a automatização.

Os casos de teste encontram-se organizados em *suites* de testes. *Suites* de testes são, na sua essência, coleções de casos de teste [65, 69].

Esta *framework* foi escolhida pela Thales pelo facto de oferecer um elevado nível de personalização através da criação de bibliotecas. Foi uma decisão a longo prazo, permitindo no futuro elaborar um nível mais complexo de testes, uma vez que também o próprio software pode gerar a necessidade de desenvolver testes com diversos níveis de complexidade.

A figura 2.9 apresenta o relatório gerado pelo *Robot Framework* após a execução com sucesso de uma *suite* de testes. Neste relatório é também possível visualizar o tempo de execução dos testes automatizados.



MyAutomation Report

Generated
20201021 22:50:08 UTC+01:00
36 days 15 hours ago

Summary Information

Status:	All tests passed
Start Time:	20201021 22:46:30.612
End Time:	20201021 22:50:07.994
Elapsed Time:	00:03:37.382
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:03:14	
All Tests	1	1	0	00:03:14	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
MyAutomation	1	1	0	00:03:37	
MyAutomation . Frontend	1	1	0	00:03:37	
MyAutomation . Frontend . Outputs Group Management	1	1	0	00:03:37	
MyAutomation . Frontend . Outputs Group Management . OutputsGroupManagement	1	1	0	00:03:37	

Figura 2.9: Relatório *Robot Framework* [15]

2.12.1.1 Arquitetura

A arquitetura apresentada na figura 2.10 é composta pelas seguintes componentes [15]:

- *Test* e *Test Data*: Conjunto de testes, *data files* e pastas, assim como os conteúdos que vão reger a execução dos testes;
- *Test results*: *Output* de correr os testes;
- *Robot Framework*: *Framework* utilizada;
- *Test tool driver*: Executa a comunicação entre a *framework* e as ferramentas em uso;
- *Testing tool*: Software utilizado para executar os testes aceitação;
- *End App*: Onde é testada a usabilidade do software para a sua aceitação pelo cliente e pelos utilizadores finais.

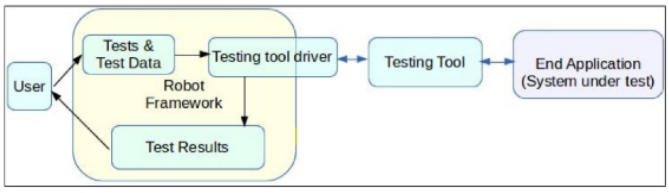


Figura 2.10: Arquitetura *Robot Framework* [15]

2.12.2 *Gherkin*

O *Gherkin* é uma *Domain Specific Language* (DSL) utilizada neste projeto devido ao seu foco direcionado para a documentação, automatização de testes e interligação com o BDD.

É possível descrever o comportamento do sistema sem o *tester*, por exemplo, ter qualquer conhecimento da estrutura interna do sistema.

Esta sintaxe facilita a comunicação com o cliente, pois permite elaborar exemplos do mundo real que facilitam a interpretação das funcionalidades do sistema, dispensando todos os pormenores que estão por trás da funcionalidade [27].

O *Gherkin* é escrito em ficheiros que representam funcionalidades. A palavra *"Feature"* oferece uma descrição de alto nível de uma funcionalidade do software e agrupa os cenários de teste associados a esta funcionalidade. Cada *feature file* é composta por um ou mais cenários que por sua vez são descritos por vários passos(*"steps"*) [20, 27]. Um *"step"* pode começar a ser escrito de acordo com uma das seguintes *keywords*:

- *Given*: descreve o estado inicial do sistema, indica as pré condições;
- *When*: descreve um evento ou uma ação, pode ser uma interação de uma pessoa com o sistema ou um evento acionado por outro sistema;
- *Then*: descreve o *output* que é esperado/desejado;
- *And*, *But*: se ocorrer uma sequência das três *keywords* acima, é possível usar o *"And"* e o *"But"* para criar uma estrutura mais fluída.

De seguida é apresentado um pequeno exemplo da sintaxe do *Gherkin*:

Listagem 2.3: Exemplo da sintaxe *Gherkin*

```
Feature
  Scenario:
    Given some precondition
      And some other precondition
    When some action by the role
      And some other action by the role
    Then some expected testable outcome
      And some other expected testable outcome
      But some expected testable outcome
```

Na *"Feature"* é possível inserir um título, um texto descritivo daquilo que é desejado que esta funcionalidade faça. Identifica a importância e benefício desta funcionalidade. No *"Scenario"* é possível inserir um pequeno título descritivo ou uma descrição do ambiente.

2.12.3 *SeleniumLibrary*

O *SeleniumLibrary* é uma biblioteca de testes da *web* para o *Robot Framework* que utiliza os módulos provenientes do *Selenium Web Driver*, permitindo manipular o *browser* [63]. Esta biblioteca é composta por várias *keywords* que permitem interagir com os vários elementos *Hypertext Markup Language* (HTML), que recebem um argumento, *"locator"*, onde é especificado como encontrar esse elemento. Estes elementos são acessíveis pelo seu *id*, *name*, *href* ou *XPath* [65]. Na figura 2.11 é possível observar três *keywords* fornecidas pela biblioteca *Selenium*.

Click Element	example	# Match based on `id` or `name`.
Click Link	example	# Match also based on link text and `href`.
Click Button	example	# Match based on `id`, `name` or `value`.

Figura 2.11: *Keywords* fornecidas pela biblioteca *Selenium* [62]

2.12.4 *Jenkins*

O *Jenkins* é um servidor *open source* que pode ser usado para automatizar quaisquer tarefas relacionadas com *building*, *testing*, *delivering* e *deploying* de software, suportando múltiplas linguagens. Possui algumas vantagens em comparação a outras ferramentas, como a sua baixa curva de aprendizagem e os múltiplos *plugins* para diferentes necessidades, entre elas, *version control*, *code quality metrics*, *build notifiers* [42]. O *Jenkins* é a plataforma onde os testes automatizados vão estar a correr. A cada integração que a equipa faça para o *Jenkins*, este vai executar todos os testes e produzir um relatório indicando quais os testes com sucesso e quais falharam.

2.13 Arquitetura da automatização

No projeto do APIS, a automatização de testes rege-se por dois padrões que definem a arquitetura dos testes automatizados. São eles: *Page Object Model* e *Model View Controller* (MVC).

2.13.1 *Page Object Model*

Devido à constante mudança das componentes que constituem a UI, a criação de testes que manipulam diretamente os elementos HTML (*page elements*) serão mais tarde descontinuados. Isto porque qualquer alteração a um *page element* vai levar a que todos os *scripts* de testes que estavam dependentes desse *page element* tenham que ser reescritos [51]. Como consequência, ocorrem despesas a nível financeiro e de tempo, podendo ainda levar a potenciais erros na aplicação se esta reescrita não for bem executada.

A solução é o *Page Object Model* (POM). Este consiste na existência de um repositório de elementos que constituem as páginas *web* da aplicação. Como tal, deve existir um ficheiro de classes, idealmente um para cada página *web* da aplicação, onde são armazenados todos os *page elements* e ações que podem ser executadas nos mesmos. Com esta implementação, a alteração de um *page element* não irá necessitar da reescrita dos *scripts*. Graças a este modelo os *testers* conseguem trabalhar num nível de abstração mais alto, uma vez que este modelo reduz a interligação entre as páginas *web* e os casos de teste [44].

Benefícios:

- Reaproveitamento de código [52];
- Fácil manutenção [52];

- Reduz complexidade e tamanho do código, tornando-o mais legível [52];
- O repositório de *page elements* é independente dos casos de teste. Consequentemente, é possível utilizar o mesmo repositório com outras ferramentas. Por exemplo, integrar o POM com, *automation runners* ou *wrappers*, como *TestNG* ou *JUnit*, para testes funcionais e ao mesmo tempo utilizar uma framework de automatização, como o *Cucumber* (secção 4.1.1.1), para testes de aceitação [53].

O POM é aplicado no protótipo apresentado na secção 4.2.

2.13.2 MVC

O *Model View Controller* (MVC) é um *design pattern* que visa organização de código, promovendo a sua fácil manipulação e interpretação. É composto por três camadas de código. Esta separação facilita a manutenção do sistema e permite desenvolvimento paralelo. Os dados encontram-se na camada *model*, a parte visual na *view* e a lógica da aplicação na *controller*.

A camada *Model* gere os dados da aplicação, fornecendo os mesmos ao *controller* e armazena dados provenientes do *controller*. Se os dados sofrerem alguma alteração, a única camada que precisa ser notificada é a *model*. A *view* gere a interação com a aplicação. Apresenta a informação ao utilizador e submete as interações do mesmo, juntamente com os dados providenciados pelo utilizador para a camada *controller*. Por fim, a camada *controller* faz a conexão entre a *view* e a *model* sendo responsável pela lógica da aplicação. É a camada onde se decide como as interações com o utilizador vão decorrer, por exemplo, quais os dados apresentados ao utilizador.

Com a adaptação deste modelo para a automatização surge o *Model UI-Driver Tests* (MUT). A camada *model* partilha as mesmas funções que no MVC, gere os objetos que devem ser atribuídos à nova camada *test*, permitindo a existência de dados relevantes para serem testados. Estes dados são por exemplo, nomes de utilizador e palavras-chave. A *UI-Driver* é a camada equivalente à *view* no MVC. Esta camada é composta, por exemplo, pelos objetos constituintes das páginas HTML a ser testadas. A camada *test* é a camada equivalente à *controller* [4].

2.14 *Exploratory testing*

Existem múltiplas técnicas de geração de casos de teste documentadas e que podem ser adotadas, dependendo das necessidades, de forma a criar casos de teste, sendo que podem consequentemente sofrer uma transição para a automatização. A equipa IVVQ optou por adotar a técnica de testes exploratórios que possui a seguinte definição: *"Exploratory testing is simultaneous learning, test design, and test execution"* [10]. Esta definição pode ser explicada da seguinte maneira: *exploratory testing* é qualquer teste onde o *tester* controla o *design* dos casos de teste, enquanto esses testes vão sendo executados. Por sua vez, o *tester*

usa a informação adquirida enquanto testa a aplicação para conseguir produzir testes com maior cobertura [10].

Este tipo de teste não apresenta nenhuns casos de teste pré-especificados sendo, portanto, uma aproximação à fase de *testing* onde o desenho dos testes é executado como parte da execução dos testes, ao invés de possuir uma fase onde os casos de teste são desenhados *a priori* da execução dos mesmos [40].

Uma das grandes desvantagens desta técnica é o facto de assentar bastante na experiência e habilidades do profissional que a está a aplicar e, por vezes, as qualidades do mesmo podem não ser suficientes. Como tal, a aplicação desta técnica está mais sujeita à possibilidade de não conseguir criar um caso de teste que descubra uma falha, comparativamente à aplicação de técnicas sistemáticas. No entanto, nenhuma técnica está livre deste cenário e, portanto, um *designer* de casos de teste que até aplique técnicas sistemáticas, pode não conseguir descobrir uma falha que seria descoberta usando *exploratory testing* [47]. Dito isto, existe uma clara vantagem em aplicar lado a lado os dois métodos de criar casos de teste, uma vez que assim é possível assegurar um maior nível de cobertura. Uma outra desvantagem da utilização de *exploratory testing* é a falta de capacidade para perceber quanta cobertura real do sistema já foi realizada, este problema resulta da falta de planeamento e seleção dos casos de teste. Por sua vez, esta última desvantagem gera uma maior dificuldade em conseguir priorizar os testes a serem executados [47].

A equipa que trabalha com o APIS 8 e está no processo de automatização de todos os casos de teste descritos no STD, sentiu que era necessário aumentar a cobertura do sistema. De acordo com a equipa, existiam vários testes por criar que necessitavam de validar campos vazios e certos valores de *inputs*, por exemplo. Devido às necessidades que a equipa expressou e após a análise do STD ficou claro que era essencial implementar algum método sistemático. Desta forma seria possível aumentar a cobertura das funcionalidades do sistema, e consequentemente aumentar a qualidade do produto apresentado ao cliente. Foi levado a cabo um estudo durante a dissertação das várias técnicas de geração de casos de teste, do qual se selecionou duas técnicas, *Cause-Effect Graphing* e *Decision Table Testing*, para aplicar no APIS 8. A aplicação destas técnicas no sistema e o seu respetivo funcionamento estão descritas na secção 5.1.

Capítulo 3

Trabalho relacionado

Neste capítulo são abordados quatro projetos. Três externos à Thales e o outro interno, sendo que este é o maior caso de motivação para a automatização dos testes no APIS, uma vez que é uma transição a decorrer, para a automatização, de outro produto da Thales. Dois projetos externos exploram as transições de duas equipas para a automatização, onde numa delas foi utilizado, à semelhança do APIS 8, Robot Framework acompanhado de keyword-driven testing. Por fim, é apresentado um projeto relacionado com o retorno de investimento da automatização.

3.1 Transição para a automatização

Nesta secção é detalhado um projeto proveniente do livro *"Experiences of Test Automation: Case Studies of Software Test Automation"*[26], onde estão descritos um conjunto de implementações de automatização de testes em diversos projetos e onde podem ser estudadas as experiências positivas e negativas desta automatização assim como novas aproximações da implementação da mesma [26](pág.30).

O projeto estudado [26] descreve um processo de transição de um projeto, que inicialmente não tinha testes automatizados, para um estado em que passado um ano, todos os testes de regressão passaram a ser automatizados. O nome da aplicação para a qual foram desenvolvidos os testes automatizados não é revelado no documento. Este projeto enquadra-se no domínio financeiro e foi desenvolvido para uma companhia dos Estados Unidos da América.

Uma das pessoas envolvidas neste processo e que elaborou toda a descrição da transição para a automatização presente no documento referenciado foi *Lisa Crispin*, que ativamente providencia apoio a comunidades de teste para que estas aprendam sobre *devOps* e *continuous delivery* e ajuda comunidades *Agile* e *devOps* a conhecerem o processo

de teste. Foi ainda premiada com o título *"Most Influential Agile Testing Professional Person at Agile Testing Days"* em 2012 [1]. *DevOps* é um conjunto de práticas destinadas a reduzir o tempo entre a confirmação de uma mudança em um sistema e a mudança sendo realizada em produção normal, garantindo alta qualidade [13].

Esta equipa era *Agile* e a necessidade de fazer entregas frequentes e a elevada velocidade de produção levaram a equipa a concluir que a automatização de testes de regressão iria permitir aumentar a velocidade com que a equipa trabalhava. Como consequência, a integração contínua seria também algo desejado a ser implementado no produto que estava a ser desenvolvido. Previamente à transição, os *scripts* eram executados manualmente, num período entre 1 a 2 dias a cada 2 semanas, uma vez que a cada 2 semanas era lançada uma nova atualização. Consequentemente, o tempo para corrigir falhas detetadas nesta fase não era suficiente. A qualidade do software não estava a ser assegurada. A equipa calculou ainda que cerca de 20% do tempo de cada iteração do software era dedicada a consertar erros de desenvolvimento.

Foi realizado um estudo interno da empresa, não relatado neste documento, e concluiu-se que com a automatização, haveria um retorno de 40% de tempo para investir no desenvolvimento.

Um aspeto que a equipa achou relevante foi o facto do resultado dos testes ser visível, sendo uma forma de publicidade. A importância da automatização dos testes vai ganhando relevo e a quantidade de testes desenvolvidos vai aumentando, assim como a sua variedade. Existe mais tempo para desenvolver novas funcionalidades e assegurar o bom funcionamento das já existentes. A compreensão por parte dos clientes quando é pedido um adiamento de uma funcionalidade é melhor aceite devido a esta visibilidade oriunda da automatização.

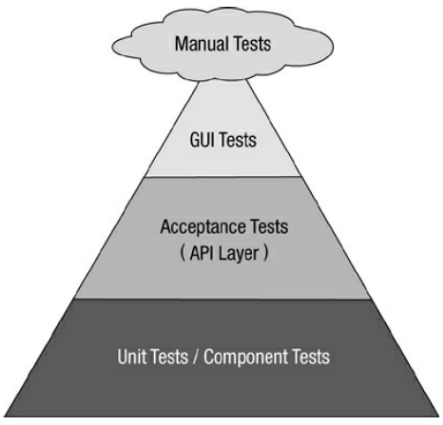


Figura 3.1: Pirâmide de testes automatizados [26]

O produto desenvolvido por esta equipa já possuía várias camadas como, por exemplo, UI, bases de dados, lógica do negócio. Como todas as componentes estavam juntas era difícil isolá-las. Para facilitar este estudo, o *manager* da equipa desenhou uma pirâmide ilustrada na figura 3.1. Com o conhecimento de que os testes unitários são os testes com

maior retorno de investimento, fáceis de manter e escrever, estes tornaram-se a base da automatização dos testes de regressão [26].

A necessidade de testar a interface continuava a ser uma realidade preocupante devido à sua fragilidade, mais dispendiosa a nível de manutenção e mais lenta de executar os seus testes correspondentes. Mas foi por aqui, como no projeto onde esta dissertação se insere, que a equipa começou por automatizar os testes. Esta decisão veio da necessidade de cobrir alguns casos de teste e garantir o funcionamento do sistema, protegendo o código. Com o desenvolvimento dos testes de interface foi necessário passar os testes para uma *build* diferente e deixar a correr durante a noite, devido ao tempo de execução.

Devido à elevada quantidade de testes já criados pela equipa, o processo de os executar começou a ser muito árduo. Como tal, um ponto essencial capturado pela equipa é a necessidade de rever os testes automatizados produzidos, refatorizá-los e garantir que todos os testes que estão a ser executados são realmente necessários. Assim, é possível manter os custos de manutenção baixos. Um ponto também importante, relatado pela equipa, foi o facto desta possuir profissionais com boas capacidade de programação, o que permitiu um melhor *design* dos testes automatizados em todos os níveis.

O empenho da equipa permitiu elaborar um sistema onde é possível obter rápido *feedback*, levando a uma fácil correção dos erros que sejam detetados. Após a correção, são criados mais testes para garantir um maior nível de estabilidade do sistema. Outro benefício adquirido foi uma maior quantidade de tempo, permitindo desenvolver mais funcionalidades e cumprir os prazos de entrega. Concluindo, a equipa conseguiu ir de encontro aos os objetivos da empresa, proporcionando um produto com a melhor qualidade possível e, ao mesmo tempo, acrescentando valor de mercado ao produto.

Este projeto assemelha-se muito às necessidades da Thales e, como tal, a sua presença nesta dissertação é relevante. O processo pelo qual esta equipa passou para melhorar a qualidade do produto e conseguir oferecer maior valor económico para a empresa e para o produto, vai ser idêntico ao processo que a Thales enfrenta de momento com o APIS.

3.2 Automatização do GDP

O trabalho apresentado de seguida é o caso mais próximo e do qual vem a maior inspiração para o desenvolvimento da automatização de testes para o APIS 8.

A Thales, para além do APIS, possui outro produto, o *GTS Digital Platform* (GDP), também este já disponibilizado ao cliente, distribuído globalmente. O "GTS" dentro do acrónimo GDP significa *Ground Transportation Systems*. O GDP é uma plataforma aplicacional que oferece componentes para construir aplicações escaláveis, modernas, robustas e seguras. Permite especificar metodologias e tecnologias modernas de desenvolvimento, boas práticas, guias de "*how-to*" e mecanismos *devOps* para controlar aplicações já lançadas.

Este produto, tal como o APIS, está num processo de transição dos seus testes manuais

para automatizados. Mas, ao contrário do APIS, que em fevereiro de 2020 ainda se encontrava numa fase muito rudimentar de automatização, na mesma data o GDP já contava com 2 meses de desenvolvimento de testes automatizados. Apesar de ser muito pouco tempo, o resultado foi significativo. São apresentados de seguida dados que comprovam os benefícios desta automatização.

O GDP possuía 843 testes, todos estes executados de forma manual previamente à automatização. Até fevereiro de 2020 foram desenvolvidos 233 testes automatizados, deixando de fora 610 testes ainda por automatizar. Calculando os valores, já existia uma cobertura automatizada de 28% de casos de teste.

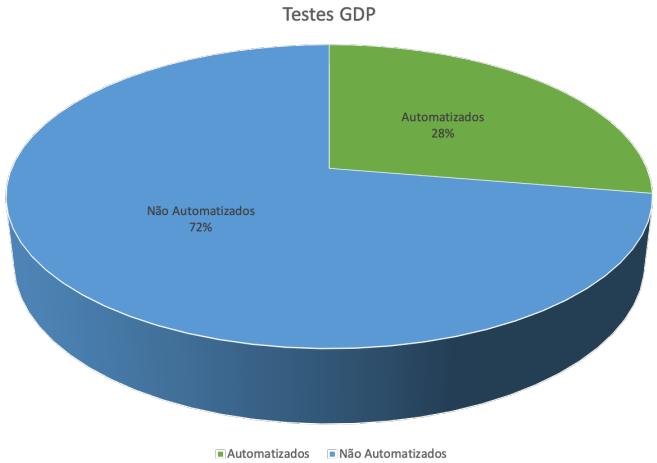


Figura 3.2: Testes automatizados e por automatizar em percentagens (fevereiro 2020)

De forma a tornar visível o impacto no tempo de execução dos testes, são apresentados agora os seguintes resultados. Os 233 testes que se encontravam automatizados foram colocados no *Jenkins* a executar. A execução dos 233 casos de teste levou 38 minutos. Um *tester*, que estivesse a *full time* a executar esta pilha de testes, iria demorar 6 dias para concluir a execução dos mesmos. Ou seja, em média, cada teste executado manualmente, iria levar 35 minutos. Um teste executado manualmente do GDP levaria quase o mesmo tempo a executar 233 testes automatizados. Na tabela 3.1 são apresentados os resultados enumerados anteriormente.

Tabela 3.1: Tabela com tempos

Tipo de teste	**Tempo de execução em minutos**
Manual - Todos os testes	8640 (6 dias)
Manual - Um teste	35
Automatizado - Todos os testes	38

Este progresso que a equipa conseguiu fazer, para além do impacto direto na libertação

de recursos valiosos, teve um impacto psicológico bastante positivo e serve também como moralizador para a equipa do APIS.

Em novembro de 2020, 8 meses após a recolha dos dados anteriores, foram apresentados os novos resultados. No decorrer da escrita desta dissertação, o GDP já conta com um total de 1784 casos de teste. São executados manualmente 962 testes e automaticamente 822. A figura 3.3 representa em percentagens os novos valores produzidos por esta equipa.

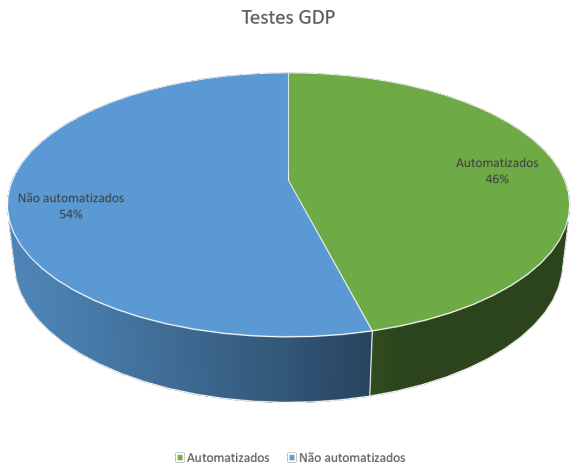


Figura 3.3: Testes automatizados e por automatizar em percentagens (novembro 2020)

O GDP utiliza as mesmas metodologias e ferramentas planeadas para a automatização do APIS. É esperado que estes dois projetos justifiquem o investimento futuro da Thales em iniciativas de alargamento do âmbito deste esforço de automatização.

3.3 Adoção de *keyword-driven testing* com auxilio de *Robot Framework*

Segue-se a análise ao caso de estudo intitulado *"Adopting Keyword-driven Testing Framework into Jenkins Continuous Integration Tool: iProperty Group Case Study"* [43]. O trabalho apresentado em seguida é um caso de estudo muito próximo do projeto desenvolvido durante a dissertação e cuja situação inicial se assemelha muito à Thales. O desejo de transitar para a automatização, a implementação de *keyword-driven testing* de forma a facilitar a compreensão por técnicos e não técnicos e o facto dos membros da equipa não possuírem avançados conhecimentos de linguagens de programação são alguns dos factos que se assemelham à situação vivenciada na Thales.

O caso de estudo tem como base a experiência da *iProperty Group*. Esta organização tal como a Thales, é uma que ainda não amadureceu a metodologia *Agile* e, como tal, a fase de testar o sistema é feita por profissionais sem experiência na área e que possuem poucos conhecimentos de linguagens de programação mas que contribuem para a especificação de casos de teste. Através da utilização de uma linguagem natural, mais descritiva e

compreensível, o problema enunciado anteriormente podia ser colmatado. Para tal, é necessária uma ferramenta que suporta *keyword-driven testing*, esta é o *Robot Framework*. O *Jenkins* foi a ferramenta integrada juntamente com o *Robot*, proporcionando integração contínua.

Em seguida, é apresentada a figura 3.4 onde é exposta a prática da equipa *iProperty.com Singapore IT* previamente à implementação das ferramentas de automatização.

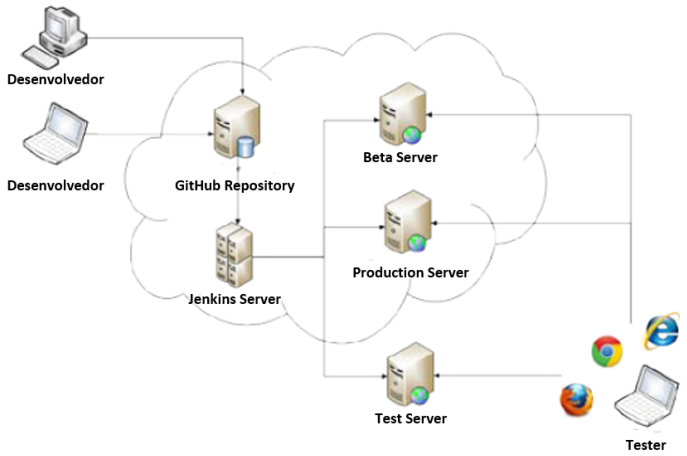


Figura 3.4: Modelo antigo [43]

No modelo da figura 3.4, por cada nova mudança, quer seja uma *user story*, quer seja a correção de uma falha, a integração das mesma no código base presente é feito em etapas. Existem três ambientes onde são executados os testes:

- *Test*: servidor local com uma base de dados teste;
- *Beta*: servidor com acesso restrito com a base de dados de produção;
- *Production*: servidor publicamente acessível.

O *Jenkins* compila o código fonte e faz *deploy* para os diferentes ambientes. O código é armazenado no *Git*. Neste modelo, para cada mudança ao código este é *deployed* e testado no ambiente *Test* primeiramente. Os testes são feitos manualmente utilizando 3 *browsers*. Após se confirmar que o comportamento da mudança é o esperado e os efeitos adversos não têm impacto nas outras funcionalidades, a mudança é *deployed* para o próximo ambiente, *Beta*, onde todo o processo de teste se repete até chegar ao ambiente *Production*.

Testes exploratórios manuais são feitos e, como tal, documentações mínimas são produzidas. Consequentemente, vai ser necessário esforço por parte do *Quality Analyst* para testar e re-testar todos os ambientes para todas as mudanças, com pouca documentação especificando o funcionamento correto das funcionalidades.

Face a estes problemas, foi criado um novo modelo onde as novas ferramentas estão integradas. Foi implementado o *Robot Framework*, juntamente com a biblioteca *Selenium2Library*. Ao usar a biblioteca é expectável que o produto seja testado ao nível da UI, sendo que está mais próxima da experiência do utilizador final.

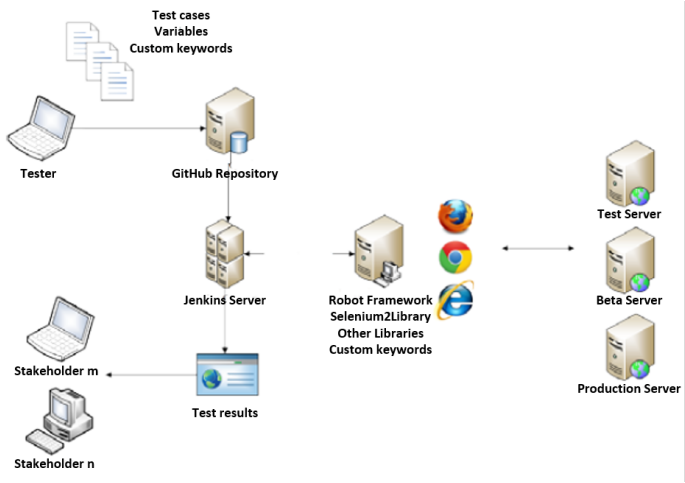


Figura 3.5: Novo modelo [43]

O novo modelo, representado na figura 3.5, permite que para qualquer funcionalidade adicionada ou correção de uma falha, após qualquer *deploy* para os ambientes de teste, que a execução dos testes seja notificada pelo *Jenkins*. O *Robot Framework* irá executar os testes e utilizará a biblioteca do *Selenium* para enviar instruções e obter resultados do *browser*. Terminada a execução do *Robot*, este compila os resultados e envia para o *Jenkins* onde os mesmos poderão ser consultados pelos *stakeholders*. "A utilização de *keyword-driven testing* pode necessitar de um grande investimento para arranque e integrar as novas práticas na infraestrutura e fluxo de trabalho existente. No entanto, os benefícios de manutenção, reutilização, visibilidade, execuções rápidas e automáticas de teste em múltiplos *browsers* e várias plataformas permitem rapidamente compensar os investimentos iniciais" [43].

A adoção da estrutura de teste *keyword-driven testing* usando *Robot Framework*, comprovou oferecer efeitos positivos para este caso de estudo. Com este caso de estudo existem duas contribuições principais para esta organização, que são:

1. Configuração da *framework keyword-driven testing* para visibilidade de status de teste;

2. Desenvolvimento de testes reutilizáveis para melhorar a capacidade de manutenção.

3.4 Retorno de investimento da automatização

Por último, é analisado um estudo relativo a métricas sobre o retorno de investimento com a automatização intitulado *"Understanding roi metrics for software test automation"* [41].

Este projeto tinha como objetivo descobrir recomendações em como adotar a automatização com um retorno de investimento positivo.

As métricas utilizadas são:

- Custo;
- Tempo;
- Eficácia.

Foi realizada uma experiência de forma a obter valores para as métricas indicadas durante 5 semanas. A mesma contou com 24 participantes, todos trabalharam independentemente em dois projetos, usando métodos de testes manuais num e noutro automatização de testes. Dos 24 participantes, 15 não tinham qualquer experiência em testar software. Ao realizar a execução manual dos testes e, em seguida, passar para a automatização, leva os dados a não serem tão fidedignos uma vez que após a execução manual os participantes passam a ter familiaridade com o sistema. Para tal, foram criados dois projetos diferentes, um para execução manual e outro para a automatização.

A ferramenta para automatizar os testes foi a *Rational Robot* [33]. A eficácia da automatização dos testes foi medida através do número de defeitos e falsos positivos reportados. Todos os participantes integraram em sessões de treino durante duas semanas para se familiarizarem com as técnicas de automatização.

De forma a repartir as diferentes fases do ciclo de teste de software, foram criadas as seguintes fases:

- Fase 1: Análise;
- Fase 2: Documentação;
- Fase 3: Treino com as ferramentas;
- Fase 4: Implementação;
- Fase 5: Execução.

As fases 3 e 4 apenas ocorrem no projeto de automatização. Consequentemente é possível perceber-se que a automatização requer um esforço extra. Em média foram necessários 30 minutos para executar 10 testes de forma manual, incluindo o tempo para ler e interpretar o caso de teste. Em modo automático, cada teste demora cerca de 1 minuto, consumindo 10 minutos de execução automatizada. São poupados 20 minutos, contudo, no primeiro ciclo de vida de testes o processo de automatização é muito mais demorado que o manual

devido às fases 3 e 4. Quando existe novo código a ser adicionado ao sistema, a velocidade de crescimento do tempo cumulativo do ciclo de testes, aquando execução automatizada, começa a decrescer em comparação à execução manual. Se a fase 3 e 4 não existissem, os testes automatizados já tinham um ciclo de vida menor. Contudo, no caso de estudo seriam necessárias 147 execuções automatizadas até haver retorno de investimento. Porém, quando o presente cenário é apenas repetir os mesmos casos de teste "n" vezes, o retorno de investimento ocorre mais cedo, necessitando apenas de 15 execuções.

A capacidade de determinar se a automatização é um recurso que compensa, assenta na comparação de dois valores. O tempo necessário de implementação da automatização, juntamente com o tempo de execução dos testes, contra o tempo de execução manual. A fase que mais vai variar é a de implementação, nela existem várias variáveis, como a frequência com que o software é alterado e a complexidade dos *scripts*. Portanto, se o tempo de execução manual for inferior à soma dos outros dois valores, então faz sentido automatizar.

Outra informação relevante foi o número de defeitos e falsos positivos detetados nas duas aproximações (manual e automática). Um total de 14 defeitos foram detetados no método manual e apenas 5 no automatizado.

Após a realização de um questionário aos participantes sobre a eficácia atingida com os testes, verificou-se que 44% dos participantes confirmaram sentir que com a automatização conseguiam testar em maior pormenor o sistema. A automatização assegurou que estes não falhassem nenhum passo na execução do teste. A mesma permitiu executar o mesmo caso de teste sempre da mesma maneira, um humano ao executar a mesma tarefa repetidamente pode tender a saltar algum passo na execução do teste devido a este processo se tornar mundano.

Concluindo, os resultados apontam para a necessidade de um grande investimento inicial em termos de custos, mas que acaba por ser amortizado ao fim de consequentes ciclos de teste ou mesmo ao longo de desenvolvimento de novos projetos.

Capítulo 4

Solução proposta e protótipo de automatização

Este capítulo apresenta a solução que a presente dissertação pretende produzir para o problema em questão. Seguidamente, é efetuada uma comparação entre as ferramentas selecionadas para a automatização de testes e as restantes no mercado, assim como a razão para terem sido escolhidas estas ferramentas. É apresentado um protótipo realizado com base no site dos CTT, de forma a exibir em prática o uso das ferramentas.

4.1 Solução Proposta

Uma vez que o processo de renovação do *Software Test Description* (STD) está em curso, esta é a altura ideal para implementar a metodologia BDD e construir um conjunto de *suites* de testes, compostas por múltiplos cenários, com o intuito de permitir que os *stakeholders* os consigam compreender. Consequentemente, como estes testes se tornam mais compreensíveis, a automatização dos mesmos torna-se mais fácil, uma vez que os *testers* conseguem ter uma interpretação mais clara dos requisitos do sistema.

De forma tornar este objetivo real, é necessário recorrer à utilização de uma ferramenta de automatização, concretamente, foi utilizado o *Robot Framework* que, através do seu alto foco em automatização, se revela como a ferramenta mais indicada para elaborar o novo cenário de testes (4.1.1.1). O *Gherkin* é a sintaxe selecionada que permite aplicar nos testes que estão a ser desenvolvidos, a metodologia BDD desejada. Esta automatização visa ainda acelerar e facilitar o processo de teste para o produto do APIS, facilitando a transição da equipa para uma prática de integração contínua. Foram, portanto, desenvolvidos testes de regressão para a versão 8 do APIS, testes estes que visam a não-regressão do software produzido e, ao mesmo tempo, construir os alicerces para um sistema que rapidamente

deteta potenciais erros em cada integração feita pela equipa. Estes testes foram postos na plataforma do *Jenkins* (4.1.1.2), que permite integração contínua, onde os testes irão ser executados a cada *commit*.

Após a análise dos casos de teste descritos no documento do STD, ficou claro que muitos possíveis cenários de teste não estavam a ser explorados e onde poderiam estar erros ainda não detetados do sistema. Isto devia-se à falta da presença de técnicas com métodos sistemáticos que permitissem a seleção de conjuntos de casos de teste que explorem em detalhe todas as funcionalidades que compõem este sistema. Como tal, foram estudadas técnicas de geração de casos de teste de forma a aumentar a cobertura do sistema e consequentemente garantir uma maior qualidade do produto. As técnicas selecionadas foram: *Cause-Effect Graphing* e *Decision Table Testing*.

4.1.1 Comparação de ferramentas

4.1.1.1 Ferramenta de automatização

A transição da execução manual dos casos de teste para a sua execução automatizada só foi possível através de uma ferramenta que permite elaborar essa automatização. Para tal, foi necessário selecionar uma ferramenta que disponha as seguintes funcionalidades:

- Agrupar múltiplos casos de teste, resultando numa *suite* de testes;
- Ferramenta dedicada à automatização;
- Possibilidade de adicionar múltiplas bibliotecas, como por exemplo o *Selenium2Library*;
- Elaborar e executar testes que avaliam o comportamento do lado do servidor (*backend*) e do cliente (*frontend*). É necessário que a ferramenta permita também avaliar o *backend* uma vez que durante o decorrer da elaboração desta dissertação houve também um projeto paralelo onde foram automatizados casos de teste para *backend*;
- Escrita dos testes numa sintaxe que seja facilmente compreensível por todas as partes envolventes do projeto do APIS 8.

Após identificadas as funcionalidades fundamentais, é possível fazer uma comparação entre as diferentes ferramentas existentes e tomar a decisão de qual se adapta melhor às necessidades de quem realiza a automatização. De entre as diversas ferramentas existentes no mercado, foram selecionadas três para efetuar esta comparação de funcionalidades. São elas: *Robot Framework*, *Cucumber* e *Selenium Web Driver*.

Na tabela 4.1 é possível observar uma comparação entre o *Robot* e o *Cucumber* e analisar o que cada um oferece.

Tabela 4.1: Comparação entre *Robot Framework* e *Cucumber* [19, 58, 59]

Funcionalidade	**Robot Framework**	**Cucumber**
Agrupar casos de teste	Sim	Não
Dedicada à automatização	Sim	Não
Adicionar bibliotecas	Sim	Sim
Configurações e bibliotecas personalizadas	Sim	Não
Sintaxe compreensível	Sim	Sim
Testar *frontend* e *backend*	Sim	Sim

Posto em suma, o *Robot Framework* é mais flexível que o *Cucumber*, devido a uma maior oferta de bibliotecas, linguagens, ferramentas, e da possibilidade de implementar por parte dos profissionais, configurações e bibliotecas mais personalizadas. É com esta *framework* que se consegue desenvolver testes automatizados para *frontend*, *backend* e de desempenho [58]. O *Cucumber*, por sua vez, é uma *framework* que se foca essencialmente na elaboração de especificações, de acordo com a metodologia do BDD, através da sintaxe do *Gherkin*. Gera relatórios dos resultados obtidos ao executar os testes destas especificações [19]. No entanto, a sintaxe do *Gherkin* pode também ser implementada no *Robot Framework*.

A ferramenta que falta debater é o *Selenium Web Driver*, no entanto, de acordo com a empresa *Expound Digital*, esta comparação não é válida, apesar da mesma ser debatida online [60]. O *Selenium Web Driver* é uma biblioteca e o *Robot* é uma *framework* de automatização de testes que, por sua vez, utiliza bibliotecas. No *Robot* são desenvolvidos testes automatizados para testar *interface* e *Application Programming Interface* (API) que, após a sua elaboração, podem ser executados nesta *framework*. Tudo isto pode ser realizado com a assistência das bibliotecas do *Selenium Web Driver*. O *Selenium Web Driver*, por sua vez permite a criação de testes funcionais de interface, mas para os executar é necessário um *automation runner* ou *wrapper*, como por exemplo o *TestNG* e *Junit*.

Concluindo, o *Robot* é uma *framework* muito mais abrangente a nível de funcionalidades e é totalmente focado para a automatização [61]. Dito isto e após a comparação feita com algumas ferramentas, o *Robot Framework* revela-se como a ferramenta que se adapta melhor às necessidades da empresa e foi, portanto, a ferramenta utilizada durante a elaboração desta dissertação.

4.1.1.2 *Jenkins VS Bamboo*

A Thales, previamente ao início da elaboração desta tese, já trabalhava com o *Jenkins*. Esta ferramenta foi escolhida devido a ser gratuita, ter um servidor que permita a execução automatizada de testes e, consequentemente, gere relatórios sobre a execução dos mesmos. No entanto, após o início da fase de automatização, não houve uma reavaliação da capacidade desta ferramenta.

Contudo, outras tecnologias foram analisadas de forma a compreender se existem grandes desvantagens em utilizar o *Jenkins*. Uma tecnologia que se revelou bastante interessante devido às suas funcionalidades é o *Bamboo*. Na tabela 4.2 é possível fazer uma análise das funcionalidades adicionais que o *Bamboo* oferece.

Tabela 4.2: Comparação entre *Jenkins* e *Bamboo* [12]

Caraterísticas	***Jenkins***	***Bamboo***
Grátis	Sim	Não
REST APIs	Sim	Sim
Automatização de testes	Necessita *plugins* (grátis)	Sim
Deploy de projetos integrado	Não	Sim
Bitbucket Server integrado	Não	Sim

O *Bamboo* simplifica o processo de fazer *merges* em *branches* do *Git*, devido à funcionalidade de *automated merging*. Automaticamente deteta, faz *build*, testa, e faz *merge* das *branches*, garantindo uma atualização contínua do código. Oferece uma visibilidade *end-to-end* da implementação, qualidade e *status* com integração do *Jira* e *Bitbucket Server* [12]. Apesar do *Bamboo* ser uma oferta interessante e a considerar, esta é paga. No *Bamboo* por exemplo, se houver a necessidade ter agentes remotos, os preços começam nos 1200 USD [11].

Esta ferramenta aparenta ser mais apta para a automatização dos testes e mais completa, no entanto, como já referido, a Thales já utilizava o *Jenkins* e não houve uma reavaliação das tecnologias para esta nova fase da automatização.

4.2 Protótipo

Previamente à apresentação do protótipo em si, é essencial apresentar a arquitetura onde este foi desenvolvido. Todo o projeto de automatização explorado durante esta dissertação seguiu a arquitetura presente na figura 4.1.

A figura 4.1 representa as ligações entre os ficheiros existentes no projeto da automatização. Segue-se agora a explicação desta arquitetura:

- De forma a explicar o significado das setas segue-se o exemplo: "*page_elements.py*" fornece variáveis/recursos às caixas verdes, por exemplo, "*gherkin_kw.txt*";

- As caixas amarelas representam ficheiros que fornecem variáveis e funções ao ficheiro "*common_kw.robot*". Os ficheiros "*common_resources.py*" e "*db_mongo_resources.py*" fornecem as funções, sendo o último provedor de funções de consulta e alteração à base de dados. O ficheiro "*configurations.py*" fornece variáveis, como por exemplo, o endereço URL onde os testes vão correr;

- As caixas verdes representam os ficheiros que aglomeram todas as *keywords* criadas pelos *testers*;
- A caixa vermelha, "*page_elements.py*", representa o ficheiro constituído pelas variáveis *XPath*, *id's*, etc., que compõe os elementos das páginas *web*;
- A caixa azul, "CTT.robot", representa o ficheiro que possui todos os casos de teste escritos, ou seja, é a *suite* de testes para as funcionalidades que estão a ser testadas.

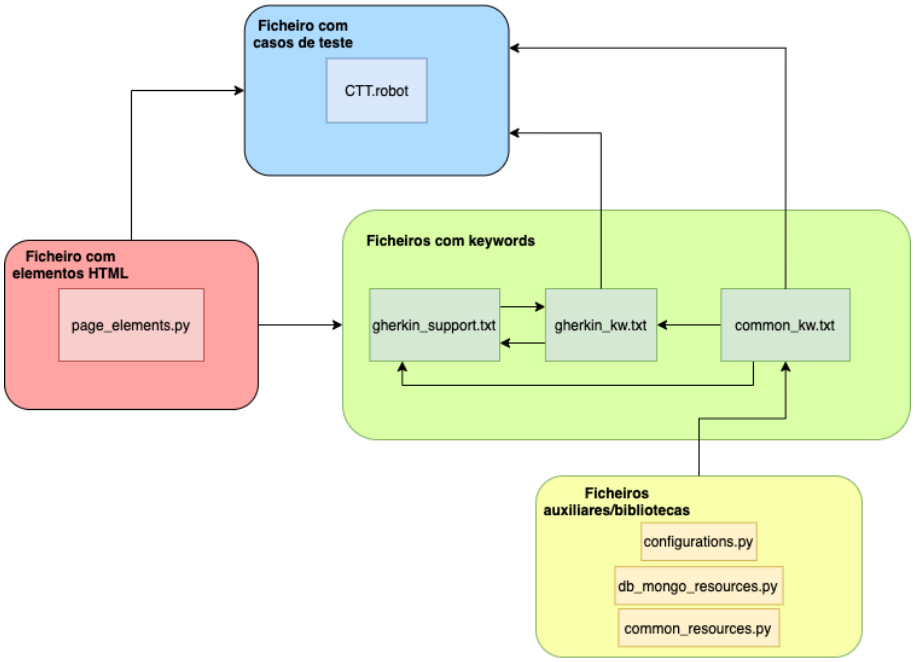


Figura 4.1: Arquitetura do processo de automatização utilizando o *Robot Framework*

No capítulo 5, na figura 5.3 está representado um diagrama de atividades onde está descrito o processo de automatização utilizando os elementos da arquitetura apresentada.

Dada como finalizada a explicação da arquitetura que foi adotada para o desenvolvimento deste projeto, o foco vira-se agora para o protótipo desenvolvido que teve como base o site dos CTT 4.2.

Para este protótipo foram desenvolvidos 2 casos de teste que consistem na navegação e seleção de vários elementos HTML na página dos CTT.

A análise ao protótipo começa pelas configurações inicias e finais das *suites* e dos casos de teste. Existe a "*suite setup*" e a "*suite teardown*". A "*suite setup*" serve para indicar quais passos a serem executados aquando o início da execução da pilha de testes, neste caso, existe por exemplo a *keyword* "*Open Web Browser*", em português "abrir explorador", que executa exatamente o que o nome indica. No final de correr todos os testes existe a "*suite teardown*". Neste caso, é pretendido fechar o *browser*. Existe ainda a "*test setup*" e "*test teardown*". Os passos descritos nestas configurações são executados no início e fim de cada caso de teste.

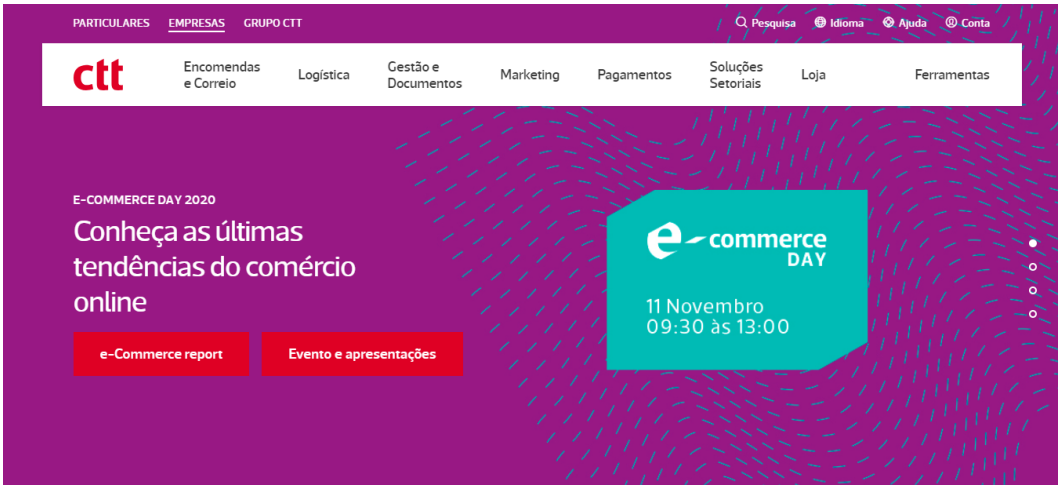


Figura 4.2: Página *web* dos CTT [64]

```
*** Settings ***
Suite Setup       Run Keywords    Set Selenium Timeout    20s    AND    Open Web Browser
Suite Teardown    Close All Browsers
Test Setup        Delete All Cookies
Test Teardown     Close Browser
Resource    gherkin_kw.txt

*** Test Cases ***

T01-Vantagens empresas
    Given I am in the "CTT Page"
    When I hoover over "Gestao e Documentos"
    And I click when available on "Recibos Online"
    Then I see "Vantagens Empresas" on the page

T02-Encontrar loja em Corroios
    Given I am in the "CTT Page"
    When I hoover over "Encomendas e Correio"
    And I click when available on "Correio"
    And I click when available on "Encontrar Lojas"
    And I search store in Corroios, Seixal
    Then I see "Corroios Store" on the page
```

Figura 4.3: Casos de teste com base no site dos CTT

Uma análise aos casos de teste, apresentados na figura 4.3, permite concluir que a sua interpretação é bastante fácil e intuitiva. Isto é possível graças ao *Gherkin*. Através da sua sintaxe, *Given When Then*, foi possível descrever em passos muito simples o que era pretendido ser executado em cada caso de teste.

Por exemplo, o primeiro caso de teste, "T01-Vantagens empresas", tem como objetivo abrir a página dos CTT e navegar até à página onde é possível consultar informações sobre vantagens das empresas. O processo para atingir este objetivo está bastante explícito. É essencial garantir que, quando o caso de teste começa a página aberta é a página inicial dos CTT. Consequentemente, são descritas as ações pretendidas que o *Robot Framework* execute, neste caso, navegar para outra página fazendo duas ações: colocar o cursor sob "Gestao e Documentos" e consequentemente carregar no botão "Recibos Online".

Relembrando a arquitetura apresentada na figura 4.1, estes casos de teste encontram-se descritos no ficheiro "CTT.robot".

Tal como foi apresentado na arquitetura desenhada, o ficheiro "CTT.robot" recebia

variáveis e ficheiros de duas fontes diferentes, os ficheiros com as *keywords* e o ficheiro com os *page elements*. De seguida, são apresentadas algumas das *keywords* necessárias para conseguir elaborar estes casos de teste de forma a interagir com a página dos CTT. Como tal, foram criadas as *keywords* presentes na figura 4.4.

```
*** Settings ***
Library           SeleniumLibrary
Variables         pageElements.py

*** Keywords ***

Open Web Browser
    [Documentation]  The keyword open, enters url and resizes browser
        Open Browser                          https://www.ctt.pt/empresas/index    chrome
        Maximize Browser Window

I am in the "${name_page}"
    [Documentation]  Waits until an element is visibe which confirms the page is loaded
        Wait Until Element Is Visible         ${initialPage['${name_page.lower().replace(" ", "_")}']}

I 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(" ", "_")}']}

I hoover over "${element}"
    [Documentation]  Hooves over an element
        Mouse Over    ${initialPage['${element.lower().replace(" ", "_")}']}

Done loading
    [Documentation]  Waits for loading to complete
        Wait Until Page Does Not Contain Element    xpath=//div[@class="blockUI"]

I see "${element}" on the page
    [Documentation]  Checks that current page contains an $element
        Page Should Contain Element           ${initialPage['${element.lower().replace(" ", "_")}']}

I search store in Corroios, Seixal
    [Documentation]  Finds a CTT store located in Corroios
    Done Loading
    I click when available on "Distrito Button"
    I click when available on "Setubal District"
    Done Loading
    I click when available on "Concelho Button"
    I click when available on "Seixal Concelho"
    Done Loading
    I click when available on "Freguesia Button"
    I click when available on "Corroios Freguesia"
    Done loading
    I click when available on "Search Store Button"
    Done loading
```

Figura 4.4: *Keywords*

As *keywords* construídas são de alto nível, permitindo uma leitura universal. É importante relembrar que a construção dos casos de teste e das suas *keywords* constituintes têm inspiração nas metodologias BDD. Estas *keywords* são construídas por passos mais pequenos, também eles escritos com *keywords* do *SeleniumLibrary*, por exemplo, *"Open Browser"* ou *"Click Element"*. Em suma, a criação destas *keywords* permite que os casos de testes sejam mais curtos e legíveis, uma vez que os passos mais complexos são descritos neste ficheiro e aglomerados numa simples *keyword*. Um bom exemplo disso é a *keyword "I search store in Corroios, Seixal"*, onde estão descritos 12 passos.

Por fim, falta indicar os *page elements* que fizeram parte destes casos de teste. Cada variável tem como valor um *XPath* ou *id*, por exemplo, que identificam unicamente o

elemento selecionado.

```
initialPage = {
    'ctt_page'              : '//h1[@class="brand"]//a//img',
    'recibos_online'        : 'xpath://*[@id="main-nav-mobile"]/nav/ul/li[3]/div/div/div[1]/ul/li[6]/a',
    'vantagens_empresas'    : 'xpath://*[@id="page-content"]/section[1]/div/div/h2',
    'correio'               : 'xpath://*[@id="main-nav-mobile"]/nav/ul/li[1]/div/div/div[1]/ul/li[3]/a',
    'encomendas_e_correio'  : 'xpath://*[@id="main-nav-mobile"]/nav/ul/li[1]/a',
    'encontrar_lojas'       : 'xpath://*[@id="page-content"]/div/ul/li[1]/a',
    'distrito_button'       : 'xpath://*[@id="districts1"]',
    'setubal_district'      : 'xpath://*[@id="districts1"]/option[27]',
    'concelho_button'       : 'xpath://*[@id="municipalities1"]',
    'seixal_concelho'       : 'xpath://*[@id="municipalities1"]/option[11]',
    'freguesia_button'      : 'xpath://*[@id="parishes1"]',
    'corroios_freguesia'    : 'xpath://*[@id="parishes1"]/option[3]',
    'gestao_e_documentos'   : 'xpath://*[@id="main-nav-mobile"]/nav/ul/li[3]',
    'search_store_button'   : 'xpath://*[@id="stationSearchForm"]/div[7]/input[1]',
    'corroios_store'        : 'xpath://*[@id="stationSearchResultForm"]/ul/li[1]/article/div/h3'
    }
```

Figura 4.5: *Page elements*

De forma a compreender como todos os elementos apresentados estão interligados, é demonstrado o seguinte exemplo: Um dos passos que se pretende efetuar durante a execução deste teste é aceder à página principal dos CTT. Dito isto, é necessário extrair o elemento que indique que a página inicial dos CTT está carregada aquando a execução do teste. Como tal, é extraído o *XPath* do elemento e guardado na variável "*ctt_page*", visível na figura 4.5. Em seguida é criada a *keyword* "*I am in the name_page*" presente na figura 4.4. Esta *keyword* é constituída por uma *keyword* de baixo nível, "*Wait Until Element is Visible*", essa *keyword* recebe um argumento, sendo ele o *XPath* do "*ctt_page*". Desta forma, a *keyword* anterior vai funcionar da seguinte forma, aguardar que o elemento selecionado esteja visível. Para concluir o exemplo, na figura 4.3, é possível confirmar-se a utilização dessa *keyword* em todos os casos de teste.

A elaboração deste protótipo revelou-se bastante útil, uma vez que permitiu pôr em prática as ferramentas e metodologias estudadas anteriormente. Ao executar este protótipo, foi possível observar a rapidez com que os casos de teste eram executados e ao mesmo tempo, confirmar que a construção dos casos de teste com a sintaxe do *Gherkin* acaba por facilitar imenso a interpretação dos mesmos.

Contudo, a complexidade deste protótipo é extremamente rudimentar quando comparado com a complexidade dos testes necessários a automatizar para o APIS. Como tal, este protótipo serve apenas como uma pequena demonstração das capacidades da automatização.

Capítulo 5

Processo de automatização

Este capítulo inicia-se com a apresentação das duas técnicas de geração de casos de teste selecionadas e a sua respetiva aplicação no sistema do APIS 8. Segue-se a explicação de como é feita a automatização dos casos de teste e as diferentes fases pelas quais esta é composta.

5.1 Técnicas de geração de casos de teste

A geração de casos de teste é um dos pilares de qualquer processo de teste de um sistema e da sua automatização, sendo que permite poupar tempo e esforço, reduzindo a possibilidade de ocorrerem erros e falhas durante a utilização do sistema [16].

Um sistema é sempre composto por múltiplas funcionalidades e usado por múltiplas pessoas, logo a geração de diversos fluxos de uso é inevitável devido às diferentes combinações de *inputs* possíveis e consequentes *outputs* gerados. Por exemplo, quando um utilizador se depara com o preenchimento de um campo onde é suposto apenas aceitar valores numéricos, é essencial validar que o utilizador não pode inserir letras nem deixar os campos a vazio e, se os deixar, o utilizador tem de ser alertado para a ausência dos dados. Outro exemplo é simplesmente a ordem pela que uma determinada funcionalidade é executada. Uma aplicação como o APIS apresenta diferentes possibilidades de realizar a mesma ação e é necessário validar todas essas interações do utilizador com o sistema. Como tal é complicado analisar todas as combinações possíveis de *inputs* e os seus resultantes *outputs*. Sem a existência de um método sistemático que permita escolher um subconjunto de *inputs* a testar, é difícil de saber se o subconjunto selecionado não vai levar à execução de testes redundantes, consequentemente originado um investimento ineficiente de recursos [46].

Para contornar este problema, existem várias técnicas de geração de casos de teste.

Estas técnicas são diversas mas todas visam a criação de casos de teste que sejam úteis e cruciais para garantir a qualidade do software produzido.

Nesta dissertação são apresentadas duas técnicas que se enquadram com o tipo de sistema que é o APIS 8 e com a necessidade da empresa em definir um método sistemático de geração de testes, assim como complementam o método informal *exploratory testing*.

5.1.1 *Cause-Effect Graphing*

O *cause-effect graphing* é uma técnica de *black box testing* focando-se apenas no comportamento externo do sistema [67]. A técnica de geração de casos de testes *cause-effect graphing* apresenta-se como um método sistemático que permite selecionar uma grande quantidade de casos de teste possíveis e, ao mesmo tempo, identificar ambiguidades e componentes incompletos que ainda não foram descobertos aquando da especificação da aplicação. Tem como objetivo aumentar a cobertura das diferentes funcionalidades da aplicação, de forma a garantir a qualidade da mesma [17, 46].

É uma técnica que permite identificar os possíveis *inputs* e os consequentes *outputs*. Tal é possível através da construção de um grafo que representa as relações lógicas entre *inputs* e *outputs*, que podem ser expressas através de expressões booleanas [67].

Existem três componentes fundamentais quando esta técnica é aplicada: causas, efeitos e restrições. As causas são os *inputs* e os efeitos os *outputs*, ou seja, as causas são condições que afetam o *output* do sistema e os efeitos são as respostas do sistema face às múltiplas combinações de *inputs*. As restrições são limitações embutidas no sistema [5, 67].

O processo inicia-se com a identificação das causas, efeitos e restrições do sistema. Segue-se a construção dos grafos que representam uma combinação de conectores lógicos compostos por nós, sendo estes as causas ou os efeitos, e arestas, que estabelecem a conexão entre os nós. Sob as arestas são utilizados os operadores booleanos, *and*, *or* e *not*. A última etapa da aplicação desta técnica consiste numa análise a todos os fluxos de utilização, representados nos grafos e levando à construção de uma tabela de decisão. Cada coluna da tabela representa um caso de teste que, potencialmente, pode ser convertido em código, de forma a testar uma funcionalidade [5, 67].

O *cause-effect graphing* é um avanço face a especificações de funcionalidades elaboradas pelo meio informal *exploratory testing*.

Para ilustrar a aplicação da técnica *cause-effect graphing*, é exibido o cenário relativo às mensagens aos passageiros, extraído do STD do projeto APIS 8. No total, são 26 casos de teste que compõem este cenário e onde são exploradas as diversas funcionalidades. Em concreto, para este exemplo, a funcionalidade selecionada é a de criar mensagens de texto via texto livre ou *template*. A figura 5.1 é retirada diretamente do sistema do APIS 8 e representa o cenário associado à criação de mensagens aos passageiros.

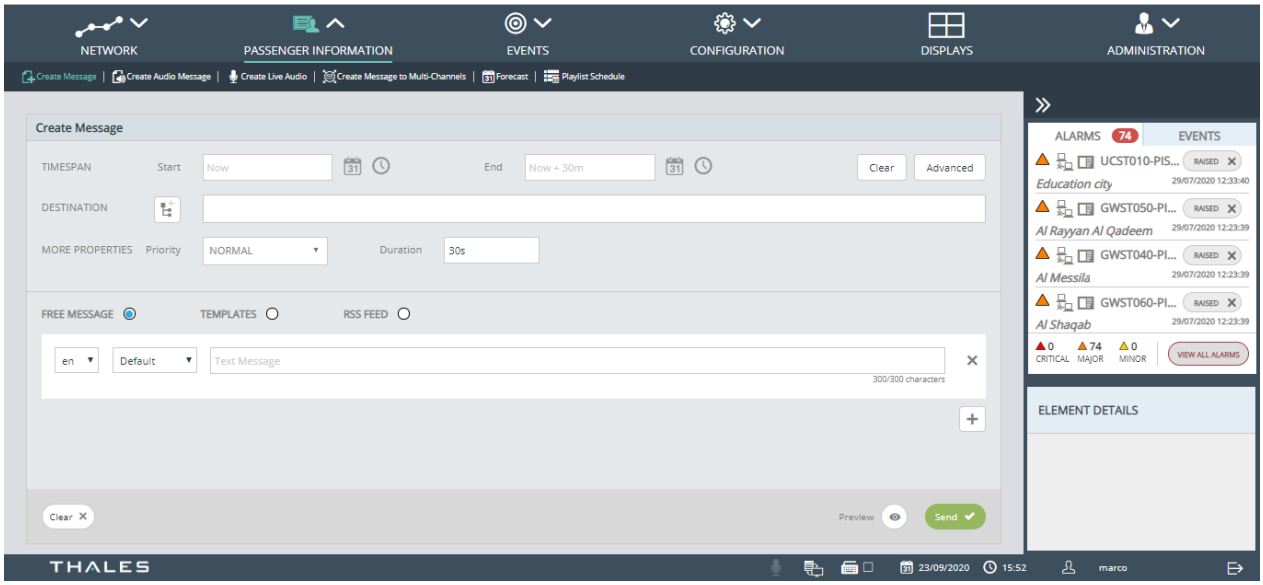


Figura 5.1: Ambiente do APIS 8*

Antes de iniciar a aplicação da técnica é essencial identificar quais os elementos presentes no *Human-Machine Interface* (HMI) que podem ser manipulados pelos utilizadores, são eles:

- *Timespan start/end*: campos que recebem uma data de início e fim como *input*. É possível inserir manualmente uma data através da seleção no calendário, ou então deixar os campos a vazio, sendo que o sistema automaticamente coloca a data de início para a data presente e a data de fim 30 minutos após a data de início;
- *Destination*: campo onde é inserido o destino pretendido para as mensagens serem disponibilizadas. Este campo não pode ficar vazio, uma vez que a aplicação notifica o utilizador dessa impossibilidade e não permite continuar aquando da tentativa de criar uma mensagem.
- *Message type*: existem dois tipos de mensagens de texto possíveis, através da escrita livre de uma mensagem de texto ou através de um *template* já existente;
- *Add extra message*: quando é selecionada uma mensagem de texto livre, é possível adicionar mais do que uma mensagem. É útil para, por exemplo, adicionar a mesma mensagem mas numa outra língua;
- *Language*: é possível emitir uma mensagem em inglês ou árabe.

Sendo que o passo inicial da aplicação desta técnica é identificar as causas, efeitos e restrições, então, as causas representadas por "C", são:

- C1: *Timespan start = empty*;
- C2: *Timespan start < current date*;
- C3: *Timespan start > current date*;
- C4: *Timespan end = empty*;
- C5: *Timespan end < current date*;
- C6: *Timespan end > current date*;
- C7: *Destination = filled*;
- C8: *Destination = empty*;
- C9: *Message type = free message*;
- C10: *Message type = templates*;
- C11: *Message type = rss feed*;
- C12: *Language = english*;
- C13: *Language = arabic*;
- C14: *Text message = filled*;
- C15: *Text message = empty*;
- C16: *Template = selected*;
- C17: *Template = not selected*;
- C18: *Timespan start = same as timespan start and time*;
- C19: *Add extra message = yes*;
- C20: *Add extra message = no*.

Os efeitos, representados por "E", são:

- E1: *Message created*;
- E2: *Empty fields*;
- E3: *Failed to create message*.

As restrições são:

- O *(Exactly one must be true)*;
- E *(At most one)*.

Em seguida foram desenhados os três grafos, um para cada efeito. Nesta demonstração, é apresentado o efeito onde a mensagem é criada com sucesso.

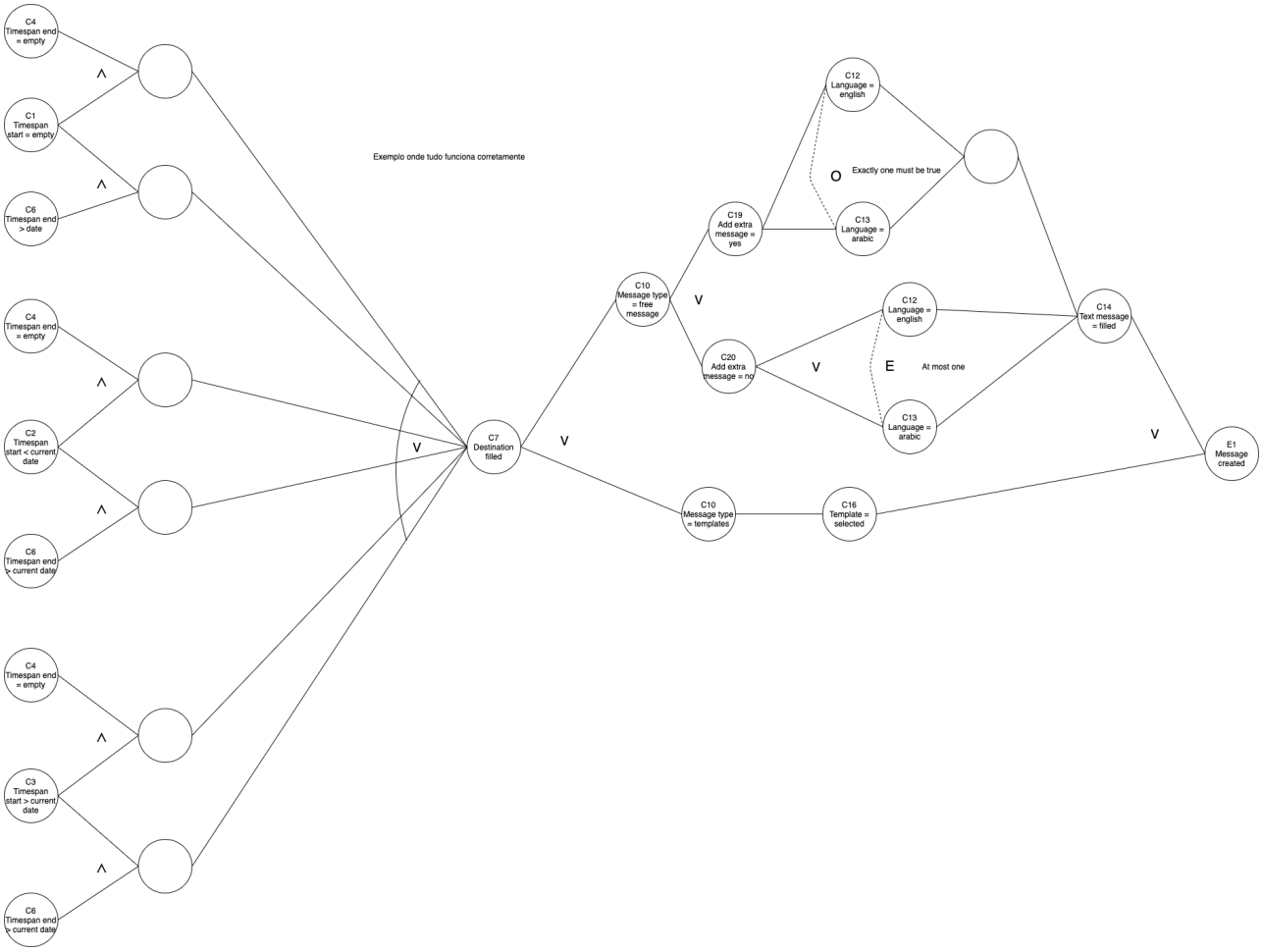


Figura 5.2: *Cause-Effect graph* para o efeito de mensagem criada

É em seguida apresentada uma análise ao grafo da figura 5.2. É possível observar do lado esquerdo as múltiplas combinações possíveis entre datas de início e fim. Para o efeito pretendido, qualquer combinação possível das datas presentes é válido. O destino tem que estar obrigatoriamente preenchido para a mensagem ser criada com sucesso. Após a causa 7, "*destination filled*", o grafo diverge em duas possibilidades, ou é enviada uma mensagem do tipo livre, ou *template*.

No caso de ser selecionada a opção *template*, é obrigatório selecionar um dos múltiplos *templates* disponibilizados e, seguidamente, a mensagem será criada com sucesso.

No caso de ser uma mensagem de texto livre, existe a possibilidade de adicionar uma ou duas mensagens de texto. Se for apenas uma mensagem, é possível escolher que esta

seja escrita em árabe ou em inglês mas no máximo, apenas umas delas é que pode ser selecionada, daí a restrição E (*at most one*). Se forem duas mensagens de texto, existe a possibilidade de criar uma mensagem em inglês e a outra em árabe, ou criar as duas na mesma língua, daí a restrição O (*exactly one must be true*), uma vez que pelo menos uma das línguas tem que ser selecionada. Antes de criar a mensagem é essencial garantir que os campos de texto estão de facto preenchidos, se estiverem, então a mensagem é criada com sucesso.

Segue-se a construção da tabela de decisão com todas as ramificações identificadas através da análise do grafo desenhado anteriormente. Cada coluna da tabela é um caso de teste. A tabela 5.1 é apenas um excerto da tabela mãe, onde estão presentes as restantes ramificações possíveis para o efeito de mensagem criada, assim como para os outros efeitos. No total, foram calculados 61 casos de teste para o cenário estudado (consultar anexo I.1).

Tabela 5.1: Tabela de decisão

Caso de teste	**1**	**2**	**3**
Causes:			
1(Timespan start = empty)	1	1	1
2(Timespan start < current date)	0	0	0
3(Timespan start > current date)	0	0	0
4(Timespan end = empty)	1	1	1
5(Timespan end < current date)	0	0	0
6(Timespan end > current date)	0	0	0
7(Destination = filled)	1	1	1
8(Destination = empty)	0	0	0
9(Message type = Free message)	1	1	1
10(Message type = Templates)	0	0	0
11(Message type = RSS Feed)	0	0	0
12(Language = English)	1	0	1
13(Language = Arabic)	0	1	0
14(Text message = filled)	1	1	1
15(Text message = empty)	0	0	0
16(Template = Selected)	0	0	0
17(Template = Not selected)	0	0	0
18(Timespan end = same as timespan start and time)	0	0	0
19(Add extra message = yes)	0	0	1
20(Add extra message = no)	1	1	0
Effects:			
1(Message created)	1	1	1
2(Empty fields)	0	0	0
3(Failed to create message)	0	0	0

Esta tabela é composta pelas causas a verde, e os efeitos a laranja. A mesma é construída da seguinte maneira: - analisar cada ramificação possível do grafo; - colocar o valor "1" se a causa e efeito ocorreram nessa ramificação e "0" se não ocorreram.

Para efeitos de melhoria de interpretação da tabela, a mesma foi convertida na tabela 5.2.

Tabela 5.2: Tabela de decisão com interpretação simplificada

Test case	Timespan start	Timespan end	Destination	Message type	Language	Text message	Template	Add extra message	Result
1	Empty	Empty	Filled	Free Message	English	Filled		No	Message created
2	Empty	Empty	Filled	Free Message	Arabic	Filled		No	Message created
3	Empty	Empty	Filled	Free Message	English	Filled		Yes	Message created
4	Empty	Empty	Filled	Free Message	Arabic	Filled		Yes	Message created
5	Empty	Empty	Filled	Free Message	Both	Filled		Yes	Message created
6	Empty	Empty	Filled	Templates			Selected		Message created
7	Empty	>Current Date	Filled	Free Message	English	Filled		No	Message created
8	Empty	>Current Date	Filled	Free Message	Arabic	Filled		No	Message created
9	Empty	>Current Date	Filled	Free Message	English	Filled		Yes	Message created
10	Empty	>Current Date	Filled	Free Message	Arabic	Filled		Yes	Message created
11	Empty	>Current Date	Filled	Free Message	Both	Filled		Yes	Message created
12	Empty	>Current Date	Filled	Templates			Selected		Message created
13	<Current Date	Empty	Filled	Free message	English	Filled		No	Message created
14	<Current Date	Empty	Filled	Free message	Arabic	Filled		No	Message created
15	<Current Date	Empty	Filled	Free message	English	Filled		Yes	Message created
16	<Current Date	Empty	Filled	Free message	Arabic	Filled		Yes	Message created

Esta conversão consiste em múltiplos passos, são eles:

1. Analisar a tabela de decisão original;

2. Para cada causa e efeito, simplificar a sua apresentação, ou seja, nesta tabela simplificada é possível observar que, por exemplo, uma das causas é simplesmente "*Destination*" e em cada célula é preenchido se o valor é "*Filled*" ou "*Empty*" de acordo se foi colocado "1" ou "0" na causa respetiva. Ao trocar "1"'s e "0"'s por texto torna a tabela mais compreensível e mais fácil de extrair os valores para cada caso de teste;

3. Identificar quais as causas que não fazem parte de determinadas ramificações. Por exemplo, no caso de teste 6, como foi selecionado um *template*, a opção de adicionar uma mensagem de texto livre extra não existe, logo, é omitido o valor da célula.

A tabela apresentada é um excerto da tabela mãe que, tal como referido anteriormente, possui 61 casos de teste. Para esta demonstração são apresentados os 16 casos de teste para o efeito de mensagem criada com sucesso.

Este foi o último passo da aplicação da técnica *Cause-Effect Graphing*. Contudo, a aplicação desta técnica gerou 61 casos de teste que, por sua vez, não é crucial serem executados, porque muitos deles são redundantes. Isto porque certos valores de *input* não vão afetar o *output*, logo, esses valores podem ser suprimidos, reduzindo a necessidade de validar o valor desses *inputs*. Assim, é importante aplicar uma técnica de geração de casos de teste que permita descobrir quais os testes cruciais a serem testados na aplicação,

garantindo assim o funcionamento correto do sistema. A técnica selecionada para esse feito é a *Decision Table Testing*.

5.1.2 *Decision Table Testing*

A técnica *decision table testing* pode ser aplicada de forma independente ou em conjunto com a técnica *cause-effect graphing*. Como explicado na técnica de *cause-effect*, a fase final da mesma é a geração de uma tabela de decisão. No entanto, é possível reduzir o número de casos de teste presentes nesta tabela de decisão, reduzindo os testes apenas aos mais cruciais. Isto porque existem vários casos de teste nos quais certos *inputs* não têm qualquer impacto no *output* do sistema, esses *inputs* podem ser suprimidos.

Esta técnica permite construir uma tabela que apresenta de uma forma bastante clara, quais os casos de teste cruciais a serem criados e quais os valores de *input*, tal como os seus efeitos, de uma forma facilmente interpretável.

A demonstração exibida em seguida tem por base a tabela de decisão gerada aquando da finalização da aplicação da técnica *cause-effect*. Antes de prosseguir com a aplicação da técnica é importante explicar que houve uma transformação da tabela gerada pelo *cause-effect* para a tabela 5.3. As mudanças são:

- As causas adotam o estilo de interrogações, de forma a que os valores inseridos nas células das tabelas sejam valores booleanos de verdade ou falso;

- Na última linha da tabela está indicado o efeito a ser estudado, se a mensagem é criada ou não. De forma a simplificar esta demonstração, assumiu-se que apenas interessa se a mensagem é criada ou não, sendo que a mensagem pode não ser criada devido a campos vazios ou a erros nos dados de inserção;

- As células com "-" significam que esses valores são irrelevantes para o caso de teste que está a ser testado, ou seja, não é um passo executado durante a execução do teste.

A tabela 5.3 é um fragmento da tabela mãe onde estão a ser avaliados 6 casos de teste.

Tabela 5.3: Tabela de decisão com secção relativa ao teste de destino vazio

Conditions	Timespan start empty?	T	T	F	F	F	F
	Timespan start < current date?	F	F	T	T	F	F
	Timespan start > current date?	F	F	F	F	T	T
	Timespan end empty?	T	F	T	F	T	F
	Timespan end < current date?	F	F	F	F	F	F
	Timespan end > current date?	F	T	F	T	F	T
	Same as?	F	F	F	F	F	F
	Destination filled?	F	F	F	F	F	F
	Free message?	-	-	-	-	-	-
	Template Selected?	-	-	-	-	-	-
	Language English?	-	-	-	-	-	-
	Text message filled?	-	-	-	-	-	-
	Extra message?	-	-	-	-	-	-
Actions	Message created?	F	F	F	F	F	F

É possível observar que os valores de *timespan start* e *end* não têm qualquer impacto no efeito final, uma vez que este é sempre falso. Como tal, é assinalado a amarelo as células que possuem valores de *timespan*.

Tabela 5.4: Tabela de decisão com células a suprimir

Conditions	Timespan start empty?	T	T	F	F	F	F
	Timespan start < current date?	F	F	T	T	F	F
	Timespan start > current date?	F	F	F	F	T	T
	Timespan end empty?	T	F	T	F	T	F
	Timespan end < current date?	F	F	F	F	F	F
	Timespan end > current date?	F	T	F	T	F	T
	Same as?	F	F	F	F	F	F
	Destination filled?	F	F	F	F	F	F
	Free message?	-	-	-	-	-	-
	Template Selected?	-	-	-	-	-	-
	Language English?	-	-	-	-	-	-
	Text message filled?	-	-	-	-	-	-
	Extra message?	-	-	-	-	-	-
Actions	Message created?	F	F	F	F	F	F

Após a identificação das células irrelevantes, as mesmas são suprimidas e os valores de "V" e "F" são substituídos por "-", ou seja, o valor daquele *input* não afeta o *output* que pretende ser testado. Ao executar esta ação também é possível observar que a linha que possui a condição/causa "*Destination filled?*" tem sempre o valor falso, logo, suprimem-se as seis colunas, resultando em apenas um caso de teste.

Tabela 5.5: Tabela de decisão com as células suprimidas

Conditions	Timespan start empty?	-
	Timespan start < current date?	-
	Timespan start > current date?	-
	Timespan end empty?	-
	Timespan end < current date?	-
	Timespan end > current date?	-
	Same as?	-
	Destination filled?	F
	Free message?	-
	Template Selected?	-
	Language English?	-
	Text message filled?	-
	Extra message?	-
Actions	Message created?	F

O resultado final é apenas um caso de teste, onde o importante é testar que a mensagem não é criada quando se insere um destino vazio.

Todo este processo é iterativo e, como tal, toda a tabela vai ser analisada com o intuito de reduzir o número de casos de teste para apenas aqueles que são fundamentais testar. O resultado final é a tabela 5.6. As células com "-" e a amarelo são as células onde é indiferente o valor colocado no *input*. As que apenas têm "-" são as células onde é possível não inserir nenhum valor de *input*.

Em suma, dos 61 potenciais casos de teste, foi possível reduzir para apenas 6 que são essenciais testar.

Tabela 5.6: Tabela de decisão final

Conditions	Timespan start empty?	-	-	-	-	-	-
	Timespan start < current date?	-	-	-	-	-	-
	Timespan start > current date?	-	-	-	T	-	-
	Timespan end empty?	-	-	-	-	-	-
	Timespan end < current date?	-	-	-	-	-	-
	Timespan end > current date?	-	-	-	-	-	-
	Same as?	F	F	F	T	F	F
	Destination filled?	F	-	-	-	T	T
	Free message?	-	F	T	-	T	F
	Template Selected?	-	F	-	-	-	T
	Language English?	-	-	-	-	-	-
	Text message filled?	-	-	F	-	T	-
	Extra message?	-	-	-	-	-	-
Actions	Message created?	F	F	F	F	T	T

Foram ainda estudadas outras funcionalidades, nas quais também foram aplicadas as técnicas, resultando em grafos, tabelas de decisões e, consequentemente, quais os casos de teste mais importantes a realizar para aquelas funcionalidades. Os testes resultantes da aplicação das técnicas foram automatizados.

5.2 Processo de automatização

Ao contrário do cenário das mensagens aos passageiros apresentado na secção 5.1, que já apresentava casos de teste automatizados, previamente à elaboração desta dissertação, muitas das *suites* descritas no STD ainda não tinham sofrido qualquer transição para a automatização. Em concreto, em janeiro de 2020, o volume de testes automatizados representava menos de 40% da cobertura.

Uma vez que estas *suites* de testes não possuíam precedentes de automatização, foi necessário realizar uma fase de estudo e de experiências de forma a compreender qual o comportamento do sistema aquando da execução dos testes de determinados cenários.

Após este estudo ter sido concluído e ser criada alguma familiaridade com os cenários presentes no sistema do APIS 8, foi necessário dar início à fase concreta da automatização dos testes.

O processo de automatização dentro da equipa do APIS 8 inicia-se com a criação de uma *issue* na ferramenta *Jira*. Esta proporciona instrumentos que permitem agilizar a coordenação de projetos, por exemplo, através da representação dos *agile kanban boards*, sendo estes bastante utilizados por desenvolvedores e, em concreto, pela própria equipa onde a dissertação ocorreu. O *Jira* permite ainda a possibilidade de fazer um acompanhamento do progresso da equipa, controlar *backlogs* e planear *sprints* [50]. Após a criação dessa *issue* é criada uma nova *branch* que fica associada à mesma. Quando é criada uma *issue* é necessário indicar algumas informações como por exemplo:

- Versão afetada;
- Prioridade;
- Resultados esperados.

Estas *issues* são criadas também para relatar falhas [49].

Para cada uma das *suites* de testes, ou seja, para cada funcionalidade, vai ser criada a respetiva *branch*, possibilitando cada elemento da equipa trabalhar em diversas funcionalidades sem afetar as restantes.

Segue-se a escrita dos casos de teste e a sua respetiva execução através do uso do *Robot Framework*. Na figura 5.3 é possível observar o diagrama de atividades onde é descrito o processo de escrita dos casos de teste.

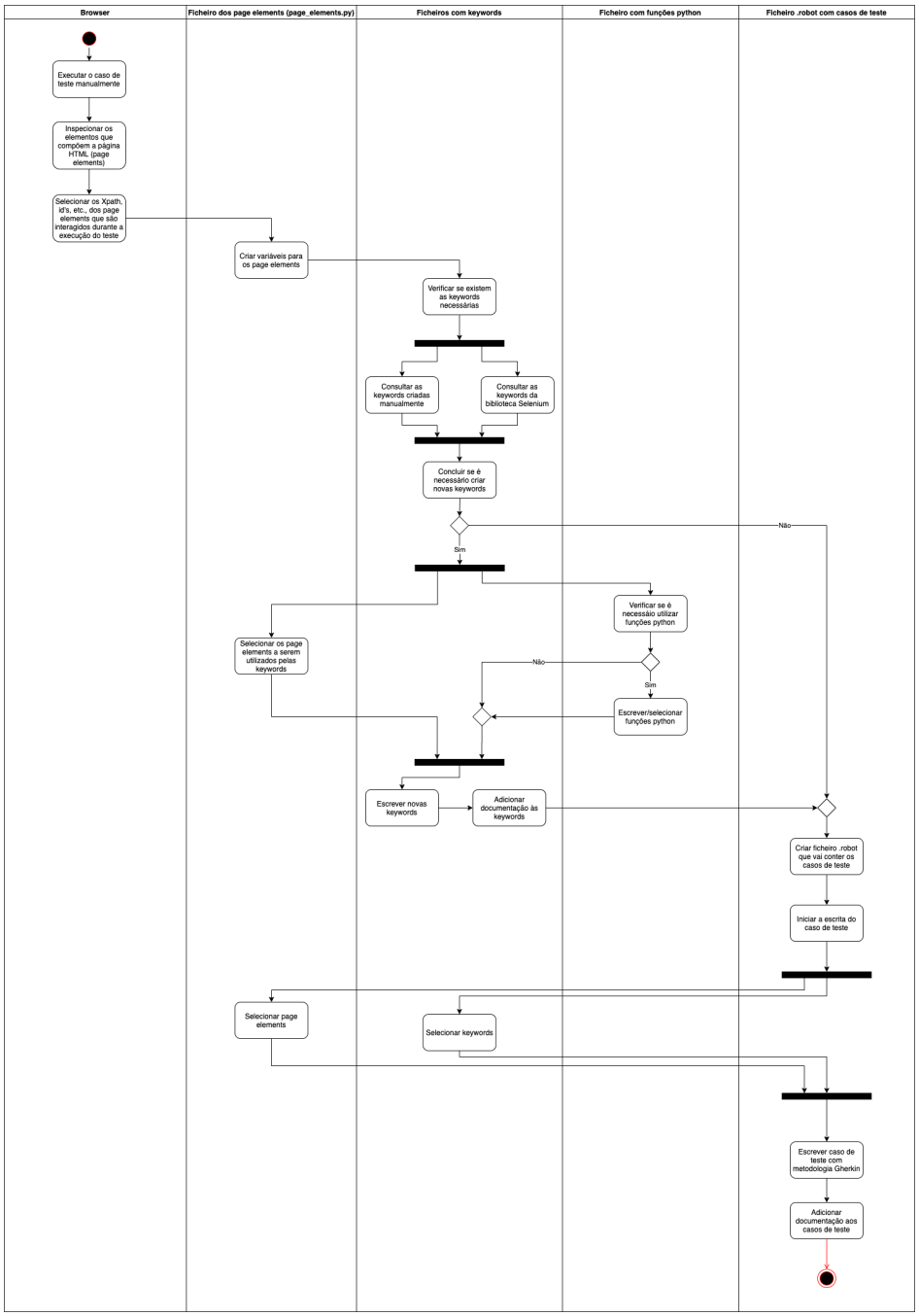


Figura 5.3: Representação do processo de escrita de casos de teste num diagrama de atividades

Após a apresentação de como se processa a escrita dos casos de teste, é relevante apresentar o que resulta da mesma. É recorrido o exemplo da figura 5.1 apresentado no subcapítulo 5.1 onde foi apresentado o cenário de criação de mensagens aos passageiros. Assumindo que vai ser criado um caso de teste para este cenário onde são executados os seguintes passos:

1. *Login* no APIS;
2. Aceder ao menu de mensagens aos passageiros;
3. Criar uma mensagem de texto com destino a *Doha*, com duração de 20 segundos e cuja mensagem contenha o texto: "*First text message*";
4. Verificar se a mensagem foi criada.

Visualmente, a criação da mensagem de texto com os passos indicados anteriormente tem o aspeto apresentado na figura 5.4.

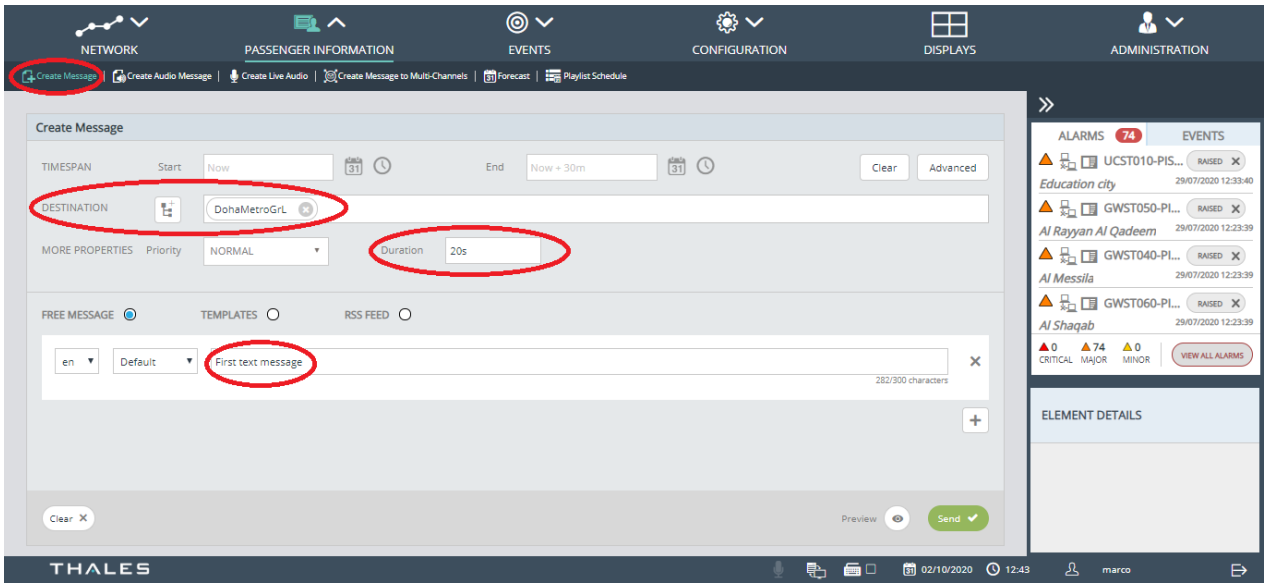


Figura 5.4: Ambiente do APIS 8 aquando a criação de uma mensagem de texto*

O respetivo caso de teste é traduzido para *Robot Framework* na figura 5.5.

```
TEST1 - SEND TEXT MESSAGES ENGLISH
    [Documentation]  Create a new free message
        Given I login to APIS
        And I enter in the Message menu
        When I write "Doha" in destination field
        And I set "Message Duration" to "20s"
        And I write "First text message" in "Text Input Box English" field
        And I click on "Send Button"
        Then The message "First text message" is in the "visual" forecast
```

Figura 5.5: Caso de teste escrito com metodologia Gherkin*

Sendo que o caso de teste é escrito numa linguagem de mais alto nível, as *keywords* que o compõem são de mais baixo nível. A figura 5.6 apresenta uma *keyword* cuja função visa preencher o campo de destino.

```
I write "${text}" in destination field
    [Documentation]  Writes the desired destination input.
        Wait Until Element Is Visible          ${current_page['destination_input']}
        Press Keys                             ${current_page['destination_input']}    ${text}+RETURN
        Element Should Be Visible              ${current_page['remove_destination_button']}
        Press Keys                             ${current_page['destination_input']}    RETURN
```

Figura 5.6: *Keyword* composta por vários passos*

Esta *keyword* recebe como argumento o nome do destino que se pretende inserir na criação da mensagem. Como já referido anteriormente, existem *keywords* que podem ser compostas por outras *keywords*. Neste caso a *keyword "I write "Doha" in destination field"* é composta por outras *keywords* que provêm da biblioteca *Selenium*.

Terminada a automatização das *suites* de testes, é necessário colocar o código produzido no repositório do *BitBucket* onde se encontram os restantes testes automatizados. É nesta fase que se insere mais um passo importante na validação da qualidade do software produzido. Após o *commit* dos novos testes, membros da equipa de testes vão validar e rever o código produzido. Neste processo é analisada a qualidade do código e se o mesmo segue as regras e estrutura dos outros testes já realizados e submetidos para a *branch* principal. Esta *branch* agrupa todos os testes automatizados até ao momento. Se tudo se encontrar de acordo com o esperado, a equipa realiza o *commit* dos novos testes para a *branch master*. O processo de revisão de código é explicado ao pormenor em 5.2.2.

De forma a visualizar como todo o processo é realizado e como todas as componentes estão interligadas entre si, é apresentado na figura 5.7 um diagrama de atividades onde estão presentes todas as fases da automatização.

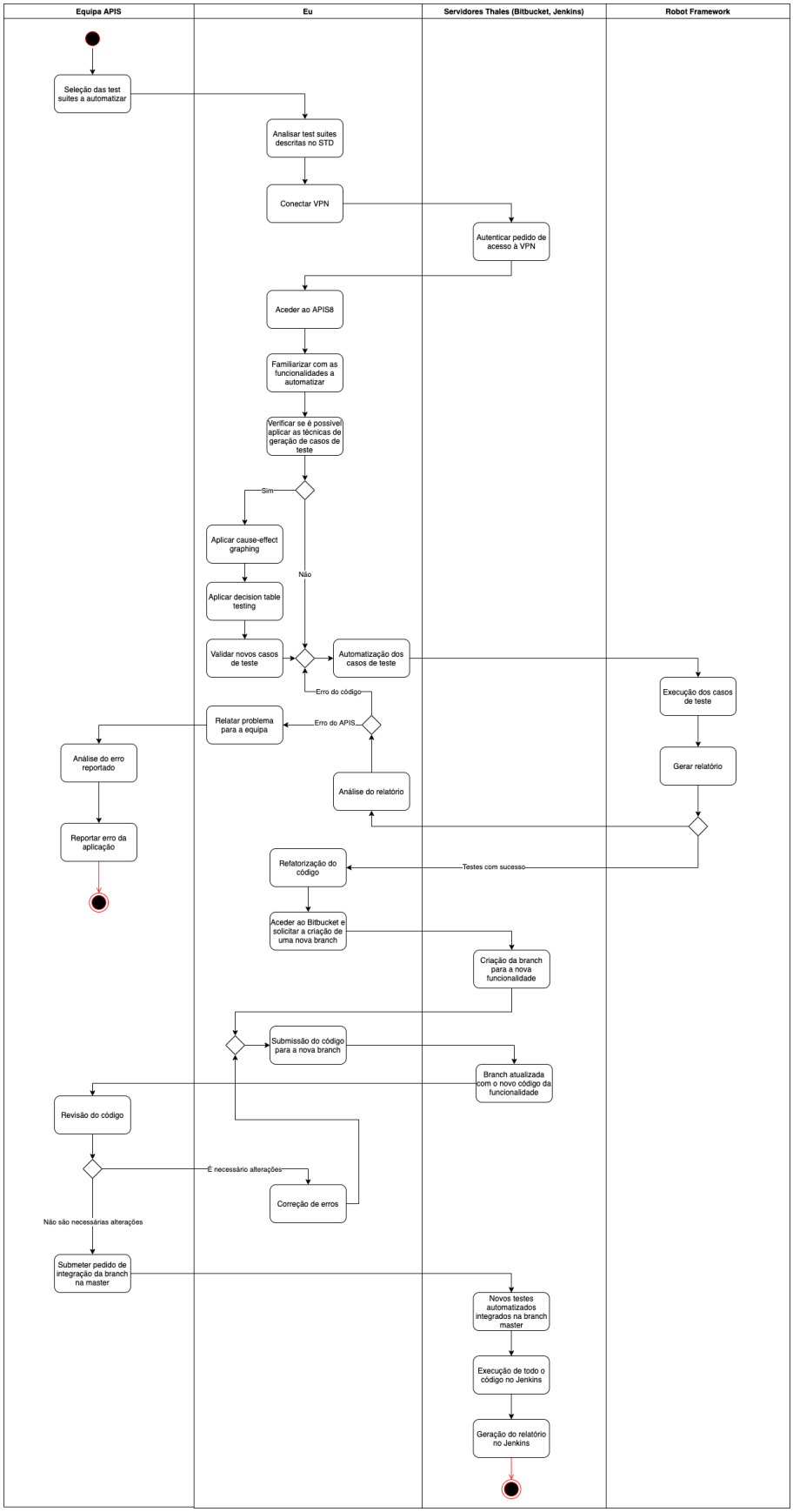


Figura 5.7: Representação do processo de automatização num diagrama de atividades

5.2.1 Testes pouco viáveis de automatização total

Existem determinados testes descritos no STD cuja total automatização não é possível devido, por exemplo, a certos casos de teste conterem etapas em que é necessário visualizar um elemento criado durante a execução do teste num painel que apenas se encontra disponível no edifício da *Thales*. Na figura 5.8 é possível verificar o resultado esperado, descrito no documento STD, o passo 3, marcado a vermelho, indica onde é necessário fazer a validação visual no painel.

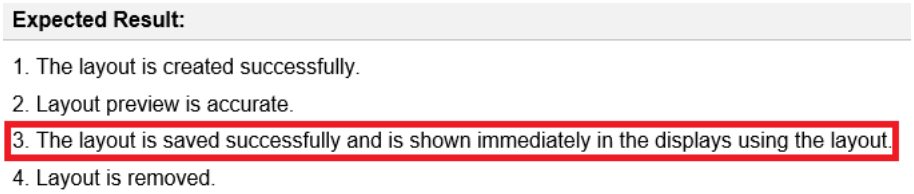


Figura 5.8: Teste impossibilitado de automatização total descrito no STD*

Consequentemente, na figura 5.9 é possível observar que o último passo do caso de teste não está concluído, sendo deixado em comentário que é necessário visualizar os painéis para concluir o teste (*TODO*).

```
APIS-STD-1005 VISUAL LAYOUT - CHANGE IMMEDIATELY
    [Documentation]  Creates a new layout and sets him to be in service immediately
        Given I login to APIS
        And I delete previous Layout Test
        When I enter in the Layout Manager menu
        And I click on "Create Layout"
        And I create a basic new layout
        Then I create a wayside triger using the Layout Test
        # TODO Check panels and then remove trigger
```

Figura 5.9: Teste impossibilitado de automatização total escrito no *Robot Framework**

Outro exemplo são os testes que envolvem a gravação de voz através de um microfone. Nestas situações apenas é possível automatizar os passos anteriores até ao momento em que é necessário utilizar o microfone, comentando no código quais os passos que necessitam de execução manual. Estes tipos de teste não são apropriados para serem submetidos às técnicas de geração de casos de teste, uma vez que existem diversas possibilidades que não podem ser exploradas devido às limitações da automatização. Existem ainda funcionalidades com determinados cenários que não apresentam benefícios em serem submetidos às técnicas de geração de casos de teste, isto porque certos testes cuja automatização foi concluída, possuem um grau de especificação elevado nos *inputs* que devem ser tomados para avaliar uma determinada funcionalidade, criando um *workflow* linear sem qualquer ramificação possível, uma vez que só existe um único valor possível para cada *input*. Como tal, a exploração de alternativas de *inputs* para estes casos de teste não é justificável. Estes testes mais específicos, por norma, são compostos por *inputs* que

constituem outros casos de teste, cuja exploração e submissão às técnicas de geração de casos de teste já ocorreu.

5.2.2 *Code review*

O processo de *code review* foca-se apenas na revisão do código que compõe os casos de teste automatizados. Existe ainda a revisão do código do sistema do APIS mas esse não se encontra no escopo desta dissertação. A revisão do código permite fazer a identificação e eliminação, o mais cedo possível de todos os potenciais defeitos do produto, resultando num produto com maior qualidade. Esta revisão consiste em confirmar, por evidências objetivas, que a forma, o conteúdo e o processo de criação dos casos de teste atendem ao objetivo e à política de gestão dos testes [66].

O processo de *code review* inicia-se com a criação de um *pull request* para a *branch* que contém os novos testes automatizados, remetendo a revisão do código a um colega que está integrado na equipa de testes do APIS 8.

Durante a análise, o *reviewer* indica quais as alterações necessárias em forma de comentário na *branch* do novo código. Estas alterações podem incluir:

- Renomeação de *keywords*;
- Alteração de *keywords* estáticas para dinâmicas;
- Criação de novos testes e consequente automatização, entre outras alterações e sugestões.

Segue-se a correção e implementação das alterações necessárias ao código. Quando concluídas, o código modificado volta a ser colocado no repositório e o processo reinicia-se, até não existirem mais alterações a realizar, terminando assim na criação de uma *suite* de testes que abrange diversos cenários possíveis.

Capítulo 6

Avaliação

Neste capítulo são apresentados os resultados provenientes do desenvolvimento deste projeto.

A elaboração da presente dissertação teve um alto foco na construção dos casos de teste e consequente automatização, como tal, em seguida são analisados todos os valores resultantes do trabalhado realizado.

O capítulo é iniciado com a análise dos ciclos de vida das *suites* de teste. É importante ter uma noção do tempo que é necessário para ser construído um conjunto de casos de teste que consiga abranger um diverso conjunto de cenários e, consequentemente, prevenir potenciais erros na utilização do sistema do APIS 8. A tabela 6.1 encontra-se ordenada de forma descendente de *commits*.

Tabela 6.1: Ciclo de vida das *suites* de teste

Suite	Data de criação	Última atualização	*Commits*
Output Groups Management	29 maio	2 outubro	26
Alarms Management	20 maio	27 julho	10
Priorities	20 maio	14 julho	5
Media Presentation	20 maio	14 julho	1
Content Templates	29 julho	24 agosto	1
Messages	12 maio	maio	1

Durante a elaboração da tese, foram 6 as *suites* de testes que transitaram para a automatização. Como é possível observar na tabela 6.1, existem *suites* que possuem um ciclo de vida maior que outras. A *suite Output Groups Management* foi a *suite* com o maior ciclo

de vida, tendo uma duração de 4 meses. Esta *suite* foi bastante trabalhada e reestruturada e, no total conta com 26 *commits*. Era uma *suite* composta por apenas um caso de teste descrito no STD, sendo que o mesmo foi automatizado, mas devido à sua elevada complexidade sofreu bastantes alterações graças ao *code review*. Em seguida, esta *suite* foi sujeita às técnicas de geração de casos de teste, resultando num total de 14 casos de teste adicionais, perfazendo então 15 casos de teste no final da construção desta *suite*. Estes 14 casos de teste, resultantes das técnicas, foram também automatizados. Por sua vez, existem *suites*, como a *suite* de *Media Presentation*, que tem um ciclo de vida mais curto, aproximadamente dois meses. Isto deve-se ao facto da *suite* apenas ter sofrido uma transição para a automatização e a mesma nunca foi revista por ninguém, contando apenas com um *commit*. O processo de automatização ocupa bastante tempo, não só na automatização dos casos de teste, mas com a consequente execução e reconstrução dos mesmos.

Segue-se a análise do volume de casos de teste automatizados, incluindo os provenientes das técnicas de geração de casos de teste.

Tabela 6.2: Análise dos casos de teste originais e gerados por técnicas

Suite	Casos de teste	Casos de teste técnicas
Output Groups Management	15	14
Alarms Management	9	1
Priorities	1	0
Media Presentation	6	0
Content Templates	12	0
Messages	10	10
Total	53	25

É possível retirar da tabela 6.2 que foram automatizados um total de 53 casos de teste. De salientar que, aquando do início da elaboração da tese, a equipa de testes do APIS 8 era constituída por quatro elementos e esta equipa elaborava entre 1 a 2 casos de teste por semana. Com a elaboração da tese foi possível contribuir com uma média de 2 casos de teste por semana. Existiram por sua vez algumas limitações durante a elaboração do projeto que não permitiram aumentar este número, nomeadamente, algumas *suites* necessitaram de grandes alterações após o *code review*. Isto deve-se à tentativa inicial de abstrair ao máximo as *keywords*, ou seja, garantir que os passos descritos nos casos de teste (futuramente utilizados como um manual) contivessem apenas *keywords* de alto nível, para aumentar a legibilidade e compreensibilidade. A criação de *keywords* mais abstratas é um processo demorado, assim como a dinamização das mesmas, de forma a fazer um maior reaproveitamento das *keywords*. No entanto, por decisão da empresa certos passos dos casos de teste não devem ser tão abstratos, então foi necessário reestruturar as *suites* de novo e mudar a forma como estas estão escritas. Por fim, o *frontend* do APIS 8 não facilitou a sua automatização, dado que a versão onde foram realizados os casos de

teste sofrer de uma latência significativa e também devido à construção do *frontend* do sistema não ser apropriado para a automatização.

Dos 53 casos de teste automatizados, 25 foram originados pelas técnicas *cause-effect graphing* e *decision table testing* (47,2%). Desta forma, foi possível demonstrar à Thales que, através da aplicação das técnicas, é possível aumentar a cobertura do sistema e, consequentemente entregar uma maior qualidade de software ao cliente final. No entanto, o volume de casos de teste gerados pelas técnicas acabou por não ser maior devido à empresa priorizar a automatização dos casos de teste já descritos no STD.

Recorrendo à *suite Output Groups Management*, sujeita às técnicas de geração de teste que resultou em 14 novos testes, é de destacar o facto de que a aplicação das mesmas e a descoberta dos testes realizou-se em menos de 6 horas. Por sua vez, é impossível calcular quanto tempo um profissional iria demorar a descobrir estes cenários através de *exploratory testing* mas, provavelmente, levaria um período de tempo maior. Isto porque, a técnica *exploratory testing* pode dar um falso sentimento de que a cobertura do sistema já está realizada, quando na verdade existem ainda testes por descobrir. As técnicas permitem assegurar de uma forma rápida que os testes criados contribuem, não só para o volume de testes, mas também para a cobertura das funcionalidades.

Segue-se a análise da dimensão das *suites* de teste.

Tabela 6.3: Composição das *suites* de teste

Suite	Passos	*Keywords* únicas	*Keywords* baixo nível	*Page Elements*
Output Groups Management	106	22	262	45
Alarms Management	50	14	88	37
Priorities	42	11	81	29
Media Presentation	43	18	279	37
Content Templates	148	26	105	35
Messages	79	17	140	37
Total	468	108	955	220

É possível extrair da tabela 6.3, que com a automatização das 6 *suites* foi necessário escrever 468 passos, utilizar e conhecer o funcionamento de 108 *keywords* únicas assim como 955 *keywords* de baixo nível e extrair e criar variáveis para 220 *page elements*.

É necessário perceber como os valores apresentados foram alcançados. Para tal, é demonstrado o caso de teste presente na figura 6.1.

```
APIS-STD-2103 OUTPUT GROUPS MANAGEMENT - DELETE OUTPUT GROUP
    [Documentation]    Creates an output group and then deletes it
        Given I login to APIS
        And I enter in the Network Groups menu
        And I click on "Create Output Group"
        And I create a new "Audio" Output Group named "Output Group To Be Deleted" with group "Fault controller" for station "Qatar National Library"
        And I see "Output Group To Be Deleted" in the list
        When Delete Output Group "Output Group To Be Deleted"
        Then I don't see "Output Group To Be Deleted" in the list
        And Logout APIS
```

Figura 6.1: Caso de teste da *suite Output Groups Management**

O número de passos neste exemplo são 8, sendo que é contada cada linha que constitui o caso de teste. Na figura é possível observar uma caixa vermelha que sinaliza um passo, assim como uma *keyword* única. Existem 8 *keywords* únicas, ou seja, não existe repetição de *keywords* neste caso de teste. No entanto, ao longo da *suite* de testes recorre-se à reutilização de *keywords* e, como tal é importante fazer a contabilização das *keywords* que não se repetem. Por exemplo, a *keyword "I create a new 'Audio' Output Group named 'Output Group to Be Deleted' with group 'Fault Controller' for station 'Qatar National Library'"*, é uma *keyword* composta por 16 *keywords* de baixo nível e nestas *keywords* são contabilizados 10 *page elements*, como é possível observar na figura 6.2.

```
I create a new "${type}" Output Group named "${name}" with group "${group}" for station "${station}"
    [Documentation]    Creates a new Output Group (audio and visual).
        I write "${name}" in "Output Group Name" field
        Wait and Click          ${current_page['group_dropdown']}
        Wait and Click          ${current_page['output_group_group']}${group}")]
        Click Element           ${current_page['expand_doha']}
        Click Element           ${current_page['expand_line']}
        Click Element           ${current_page['output_group_devices']}${station}")]
        Run Keyword If          $type == "Audio"
        ...     Run Keywords
        ...     Click Element           ${current_page['expand_station']}    AND
        ...     Click Element           ${current_page['audio_station']}
        ...     ELSE IF         $type == "Text"
        ...     Run Keywords
        ...     Click Element           ${current_page['expand_station']}    AND
        ...     Click Element           ${current_page['text_station']}
        Click Button            ${current_page['add_button']}
        Click Element           ${current_page['save_group']}
```

Figura 6.2: *Keyword* composta por *keywords* de baixo nível*

Na figura 6.2, é possível observar que existem secções na *keyword* com o seguinte formato: '$*type*'. No caso de teste, o valor '*type*' é substituído pela palavra '*Audio*'. Ao utilizar este formato são criadas *keywords* dinâmicas, uma vez que a mesma *keyword* consegue produzir *outputs* diferentes dependendo dos parâmetros de entrada.

Para o caso de teste da figura 6.1, é apresentado na tabela 6.4, o número de *keywords* e *page elements* que constituem este caso de teste.

Tabela 6.4: Composição do caso de teste *Delete Output Group*

Passos	*Keywords* únicas	*Keywords* baixo nível	*Page Elements*	Percentagem de abstração
8	8	59	34	86,5%

Graças à capacidade de abstrair os casos de teste, através da utilização de *keywords* de alto nível, para o caso de teste em análise é possível atingir uma percentagem de abstração de 86,5%. Isto significa que apenas 13,5% são *keywords* visíveis pelo utilizador final, sendo que 86,5% são *keywords* de baixo nível, onde os processos mais específicos estão "ocultos".

Uma das grandes vantagens da automatização dos casos de teste é o facto dos mesmos serem executados num espaço de tempo inferior ao que se os mesmos fossem executados

manualmente. A frase apresentada anteriormente é um dos pilares da automatização de testes e, como tal, é essencial no contexto desta dissertação apresentar os valores obtidos durante a elaboração da mesma de forma a suportar essa afirmação.

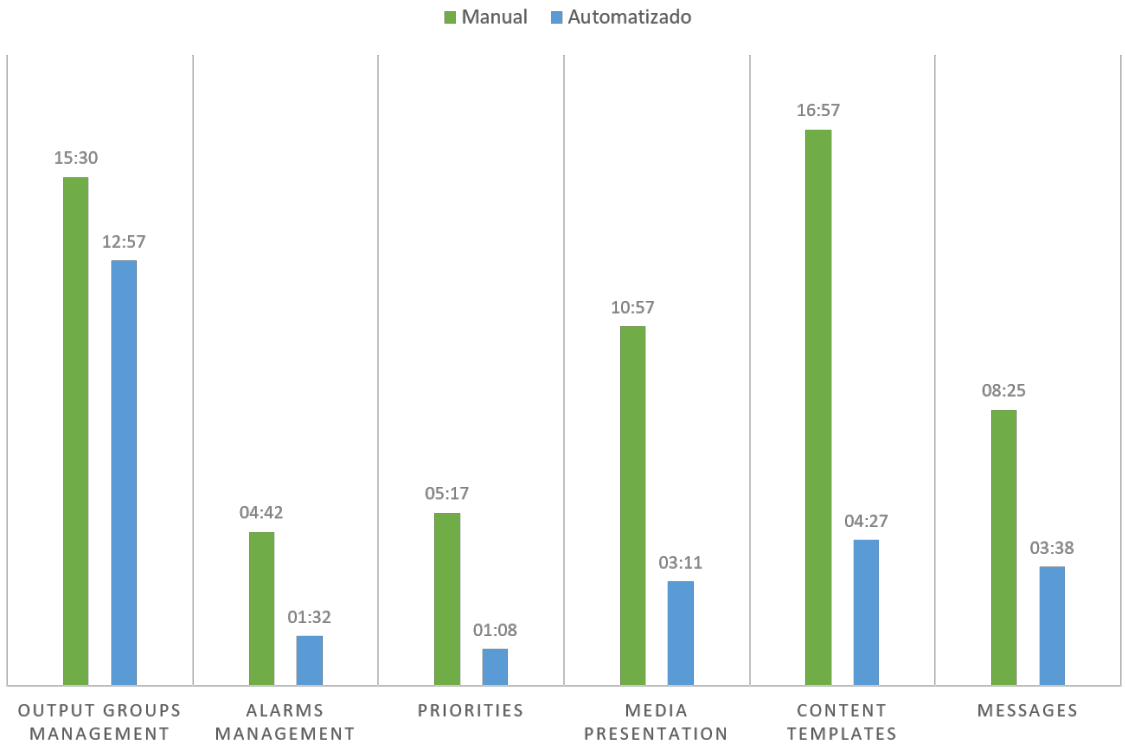


Figura 6.3: Tempos de execução manual e automatizado de cada *suite*

Os tempos manuais foram retirados de execuções cronometradas utilizando um cronómetro como auxílio, os tempos automatizados provêm dos relatórios gerados pelo *Robot Framework*. Todos os resultados obtidos foram produzidos pela máquina disponibilizada pela Thales que é constituída pelas seguintes caraterísticas:

- 16 GB RAM
- Windows 10 *Enterprise*
- 64 bits *Operating System*
- *Intel core* i5-4310M CPU@2.70GHz

Para além das especificações enumeradas anteriormente, é importante ter conhecimento de que apenas é possível aceder ao sistema do APIS 8 através da utilização de uma *Virtual Private Network* (VPN). Dito isto, esta VPN conecta-se a servidores presentes na Alemanha, o que resulta numa maior latência. Devido à existência de inconsistências entre a versão disponibilizada para a elaboração da dissertação e a versão que se encontra a ser executada no *Jenkins*, não foi possível obter os resultados dos tempos de execução provenientes do relatório do *Jenkins*. Como tal, para efeitos de análise de resultados, assume-se que num

potencial caso em que os testes estavam todos a correr no *Jenkins*, os tempos de execução seriam menores aos apresentados, uma vez que correm em servidores específicos para esse fim e possuem uma maior capacidade de execução.

Durante a elaboração da dissertação foi criada uma familiaridade com o sistema e, como tal, o tempo de realização dos passos descritos nos casos de teste de forma manual é menor do que se um outro profissional sem experiência executasse os mesmos testes. O cenário apresentado anteriormente é uma realidade na Thales, profissionais de diferentes áreas são solicitados para executar *suites* de testes sobre ambientes que podem não ter qualquer conhecimento.

É possível observar na figura 6.3 que em todas as *suites* os tempos de execução automatizada mostram uma grande vantagem no porquê de se realizar a transição para a automatização. Serve de exemplo a *suite Content Templates*, onde é possível observar uma diferença de 12 minutos e 30 segundos entre a execução manual e automatizada. Em seguida, na figura 6.4, é apresentado o tempo total de execução manual e automatizada de todas as *suites* trabalhadas durante a dissertação.

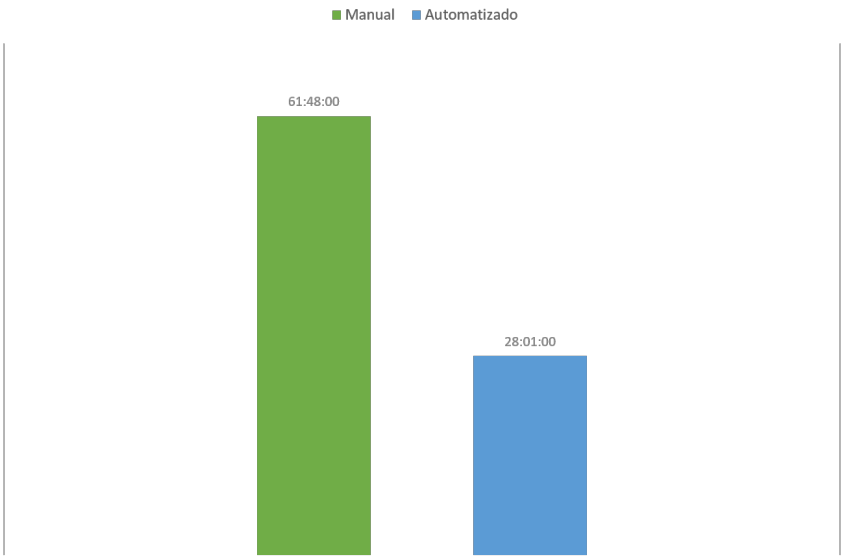


Figura 6.4: Tempo total de execução manual e automatizado de todas as *suites*

No total, em cada execução das 6 *suites*, existe uma diferença de 33 minutos e 47 segundos entre os dois tempos.

É crucial compreender que não deve apenas haver um foco na diferença dos tempos de execução, o ponto mais importante é o facto de que, no total é poupada cerca de uma hora com a automatização. Enquanto que no passado sempre que ocorria o lançamento de uma nova versão, um *tester* tinha que estar uma hora a correr estas 6 *suites*, agora estas correm autonomamente.

Uma vez que os testes automatizados correm todos os dias nos servidores da Thales, é possível haver um acompanhamento contínuo do projeto e estar sempre alerta dos potenciais erros que podem ser injetados no sistema aquando o desenvolvimento de

novas funcionalidades e atualização das mesmas. Tal era impossível no passado, devido ao elevado custo que é executar manualmente todos os testes com a mesma frequência dos testes automatizados, diariamente. Estes testes eram executados na fase de lançamento de uma nova versão, que ocorre de dois em dois meses, resultando na deteção tardia de erros e, consequentemente, um período de tempo muito curto para resolver esses erros, uma vez que esta fase de execução manual é realizada, por vezes, uma semana antes do lançamento.

Na tabela 6.5 são apresentados os tempos de execução, em modo manual e automático, ao longo de diversos intervalos de tempo.

Tabela 6.5: Tabela com tempos de execução ao longo de várias medidas de tempo

	Manual (hh:mm)	Automático (hh:mm)	Diferença
1 dia	01:02	00:28	00:34
1 semana	7:14	3:16	3:58
1 mês	31:00	14:00	17:00
1 ano	377:10	170:19	206:49

Como é possível observar na última linha da tabela 6.5, onde estão a ser analisados os tempos de execução ao fim de um ano, seria necessário ter um *tester* a executar testes manuais durante 15 dias e 16 horas (377h:10min) para conseguir acompanhar o fornecimento de cobertura da automatização. Ou seja, colocando estes valores dentro das típicas 8 horas de trabalho, seriam necessários cerca de 47 dias úteis, mais de 2 meses, de trabalho a tempo inteiro.

Na figura 6.5 observa-se como os tempos de execução vão evoluir ao longo de 3 anos. É possível observar que vai havendo uma diferença cada vez maior entre os dois tempos.

A bateria de testes que oferece a cobertura do APIS 8 continua a crescer, quer seja através da aplicação das técnicas de geração de casos de teste, quer seja pela aplicação de *exploratory testing*. Dito isto, se os testes que futuramente forem criados continuassem a ser executados manualmente aquando o lançamento das novas versões, o tempo necessário para executar toda a bateria de testes iria continuar a aumentar. Graças à automatização, a bateria de testes pode crescer sem haver preocupação com os custos variáveis, naturalmente existe o custo de automatizar os testes mas esse eventualmente é amortizado.

No momento da conclusão da escrita da dissertação confirma-se que 10 *suites* de 32 estão automatizadas, ou seja, existe uma cobertura automatizada de 31,3%.

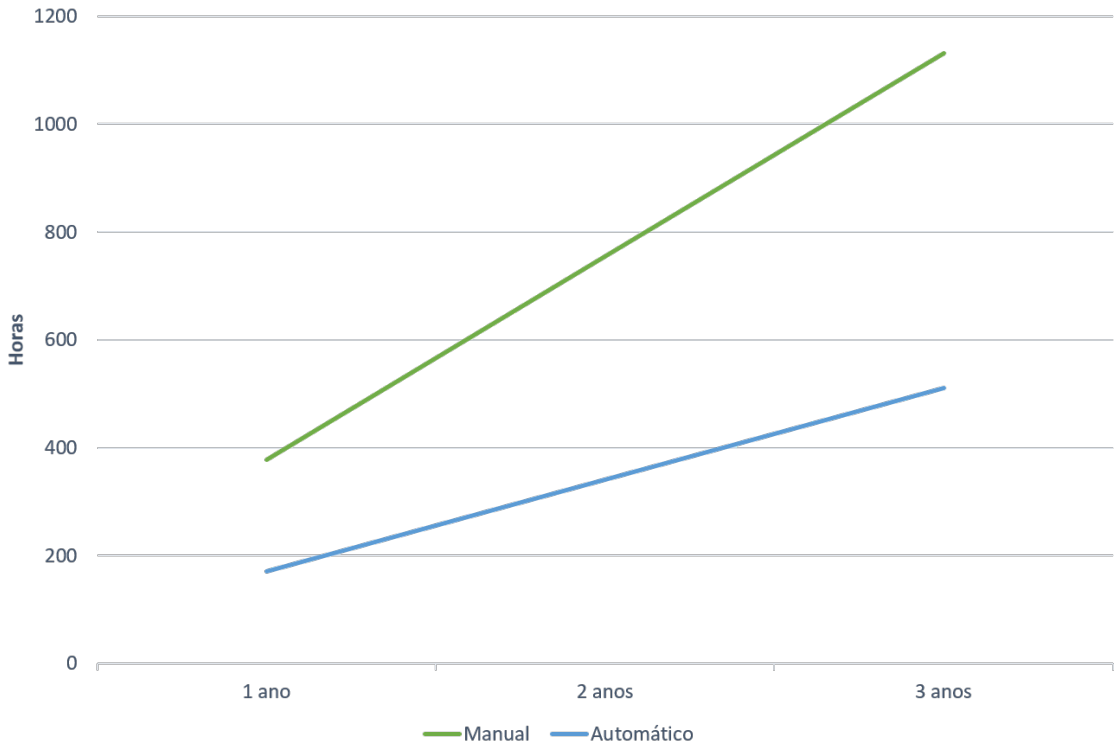


Figura 6.5: Tempo total de execução manual e automatizado de todas as *suites*

Capítulo 7

Conclusão

Neste capítulo final é feita a análise do trabalho realizado durante a dissertação e como esta auxiliou a Thales na sua transição para a automatização. No fim do capítulo são apresentadas sugestões para trabalhos futuros.

7.1 Visão geral do trabalho desenvolvido

Com a finalização da elaboração da dissertação é importante relembrar o ponto de partida desta tese e como o trabalho envolvido na mesma levou à contribuição para a Thales na sua transição para o mundo da automatização e das metodologias de geração de casos de testes.

A Thales apresentou inicialmente a necessidade de automatizar casos de testes para *frontend* já descritos no STD. Os testes descritos neste documento necessitavam de ser reescritos numa linguagem mais clara e universal, de forma a que técnicos e não técnicos possam facilmente executar os passos descritos nestes testes. Como tal, foram estudadas opções que permitissem introduzir a metodologia BDD na escrita dos novos testes. Ao mesmo tempo, a automatização foi uma tecnologia que despertou o interesse da Thales uma vez que a mesma está num processo de transição para a integração contínua do sistema produzido pela empresa, o APIS 8. Dito isto, foi selecionada a *framework Robot Framewok* onde foram escritos todos os casos de testes e de onde foram retirados todos os valores de tempos de execução automatizada. Estes testes beneficiaram de uma escrita legível através da implementação da sintaxe *Gherkin*.

A execução automatizada de todas as *suites*, elaboradas durante este projeto, permitiu concluir que de facto, a transição para a automatização compensa, uma vez que os valores resultantes da comparação entre os tempos de execução manual e automatizada mostram

uma clara vantagem para os tempos automatizados. Relembrando os resultados apresentados no final do capítulo anterior, no tempo total de execução de todas as *suites* havia uma diferença aproximadamente de 34 minutos entre a execução manual e automatizada. Com a automatização, foi possível retirar o esforço necessário por parte de um *tester* para executar as 6 *suites* a cada dois meses, poupando-lhe cerca de uma hora de trabalho. No entanto, como referido anteriormente, graças à automatização agora os testes podem ser executados diariamente, permitindo um sistema que beneficia de integração contínua. Tal não seria possível se os testes fossem executados manualmente todos os dias uma vez que ia consumir imenso tempo por parte de um *tester*.

Para além do processo de escrita dos testes, houve também uma vertente que começou a ganhar destaque durante a elaboração da dissertação, a necessidade de implementação de técnicas de geração de casos de teste. A equipa demonstrou preocupação com a falta de cobertura dada pelos testes executados uma vez que, muitos cenários possíveis aquando o teste das funcionalidades do APIS 8 não estavam a ser verificados, como por exemplo, valores de *inputs* diferentes aos indicados no STD, campos vazios, entre outros. Isto provocado pela utilização de *exploratory testing* na geração de casos de testes. Ficou claro que era necessário implementar um método sistemático que permitisse descobrir casos de testes que, consequentemente, aumentassem a cobertura do sistema, resultando assim, num produto com maior qualidade.

Foram apresentadas duas técnicas de geração de casos de teste, *cause-effect graphing* e *decision table testing*. O *cause-effect graphing* permitiu ajudar na descoberta de múltiplos casos de teste possíveis para testar as funcionalidades do APIS, enquanto que o *decision table testing* ajudou a determinar quais os casos de teste que são cruciais para garantir o bom funcionamento do sistema. Com a aplicação destas técnicas foi possível contribuir com 47,2% (25) novos testes para as 6 *suites* automatizadas. É importante compreender que a técnica de *exploratory testing* não deve ser totalmente ignorada. Esta técnica assenta bastante na experiência do profissional que está a desenvolver os casos de teste e esta é bastante valiosa, dado que nem sempre as técnicas metodológicas detetam por inteiro todos os cenários possíveis e, como tal, é importante conseguir conciliar as três técnicas apresentadas.

O tempo de permanência na Thales durante a elaboração da tese não foi suficiente para conseguir confirmar se, de facto, as técnicas de geração de casos de teste é algo que vai passar a ser utilizado no dia-a-dia perante a criação das *suites* de testes. Contudo, é importante que o conhecimento seja transmitido para todas as equipas, para que possam no futuro, adotar este método sistemático face ao método *exploratory testing*. Para tal, foi escrito um documento que será integrado nos documentos oficiais da Thales, que descreve em pormenor, todo o processo de aplicação das técnicas de geração de casos de teste utilizadas durante a elaboração da tese e como este se adapta às diversas funcionalidades do APIS 8. Além disso, foi elaborado um manual, no mesmo documento, onde os processos da aplicação das técnicas são generalizados, de forma a que seja possível aplicar as mesmas nas diferentes funcionalidades do APIS 8 (ver anexo 2).

De momento, a Thales encontra-se na fase de desenvolvimento do APIS 9 e, como tal, conclui-se que a aplicação da automatização, acompanhada pela aplicação de técnicas de geração de casos de teste, permite a criação de um produto que beneficie de:

- uma cobertura de testes mais alargada;
- um conjunto de casos de teste que servem de manual de utilização do sistema, devido à sua linguagem universal;
- uma execução diária dos casos de teste, permitindo um *feedback* rápido sobre o estado em que se encontra o sistema (com falhas ou não), proporcionando assim a integração contínua do sistema.

7.2 Trabalho futuro

Este trabalho abre caminho para possíveis extensões, entre as quais se destacam:

- Dar continuação ao desenvolvimento dos casos de teste, através da aplicação das técnicas de geração e, consequente automatização dos testes. Existem ainda 22 *suites* de testes por automatizar, ou seja, resta garantir 68,7% de cobertura automatizada. Contudo, algumas destas *suites*, possivelmente, não serão submetidas à transição para a automatização, no entanto, a Thales não conseguiu atribuir nenhum valor devido ao estado desatualizado do STD;
- Durante o desenvolvimento desta dissertação, decorreu também o início da automatização de testes para o *backend*, como tal, algo importante seria a fusão dos testes de *frontend* com os de *backend* permitindo assim isolar por completo a execução de testes num ambiente automático. Isto porque existem certos testes cujos passos utilizam ferramentas, como por exemplo o *swagger* de forma a realizar pedidos *REST*. Para diminuir a complexidade dos casos de teste, por exemplo, as pré-condições de testes de *frontend* podem ser feitas através da utilização de bibliotecas fornecidas pelo *backend* e não da manipulação do HMI;
- Utilização de uma ferramenta que permita uma transição direta dos casos de teste gerados pelas técnicas para o *Robot Framework*. O resultado da aplicação das técnicas *cause-effect graphing* e *decision table testing* é uma tabela de decisão. Esta tabela pode ter múltiplas dimensões e no caso de ser uma tabela extensa com, por exemplo, mais de 15 casos de teste, torna-se complicado estar a extrair cada valor de cada linha da tabela para construir o caso de teste. Com uma ferramenta que fizesse a transição da tabela de decisão para o *Robot* poderia-se evitar falhas humanas e acelerar o processo de criação de testes, assim como da automatização.

Bibliografia

[1] *About Lisa Crispin*. https://lisacrispin.com/about/. Accessed: 21/09/2020.

[2] *Acceptance Test Driven Development (ATDD)*. https://www.agilealliance.org/glossary/atdd. Accessed: 15/02/2020.

[3] S. Acharya e V. Pandya. "Bridge between Black Box and White Box–Gray Box Testing Technique". Em: *International Journal of Electronics and Computer Science Engineering* 2.1 (2012), pp. 175–185.

[4] *Adapting Model View Controller To Test Automation*. https://huddle.eurostarsoftwaretesting.com/adapting-mvc-to-test-automation/?_sm_au_=iVVWTVnf5DfQ6n05f7CV7K0qc3s8c. Accessed: 20/02/2020.

[5] M. Adler e M. A. Gray. "A formalization of Myers cause-effect graphs for unit testing". Em: *ACM SIGSOFT Software Engineering Notes* 8.5 (1983), pp. 24–32.

[6] *Advanced Passenger Information System Release 8.9.7 User Manual*. Rel. téc. Thales Portugal, 2017.

[7] *Agile*. https://www.atlassian.com/agile. Accessed: 29/01/2020.

[8] *Agile Project Success Rates are 2X Higher than Traditional Projects (2019)*. https://vitalitychicago.com/blog/agile-projects-are-more-successful-traditional-projects/. Accessed: 20/11/2020.

[9] *ATDD vs. BDD, and a potted history of some related stuff*. https://lizkeogh.com/2011/06/27/atdd-vs-bdd-and-a-potted-history-of-some-related-stuff/. Accessed: 06/02/2020.

[10] J. Bach. "Exploratory testing explained". Em: *Online: http://www. satisfice. com/articles/et-article. pdf* (2003).

[11] *Bamboo pricing*. https://www.atlassian.com/software/bamboo/pricing. Accessed: 29/01/2020.

[12] *Bamboo VS Jenkins*. https://www.atlassian.com/software/bamboo/comparison/bamboo-vs-jenkins. Accessed: 29/01/2020.

[13] L. Bass, I. Weber e L. Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[14] *BDD, Regression Testing and some feature requests*. https://lizkeogh.com/2007/10/25/bdd-regression-testing-and-some-feature-requests/. Accessed: 06/02/2020.

[15] S. Bisht. *Robot framework test automation*. Packt Publishing Ltd, 2013.

[16] P. N. Boghdady, N. L. Badr, M. Hashem e M. F. Tolba. "A proposed test case generation technique based on activity diagrams". Em: *International Journal of Engineering & Technology IJET-IJENS* 11.03 (2011), pp. 1–21.

[17] *Cause and Effect Graph – Dynamic Test Case Writing Technique For Maximum Coverage with Fewer Test Cases*. https://www.softwaretestinghelp.com/cause-and-effect-graph-test-case-writing-technique/. Accessed: 25/05/2020.

[18] E. Collins, A. Dias-Neto e V. F. de Lucena Jr. "Strategies for agile software testing automation: An industrial experience". Em: *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. IEEE. 2012, pp. 440–445.

[19] *Cucumber*. https://cucumber.io/. Accessed: 31/01/2020.

[20] *Cucumber, Gherkin*. https://cucumber.io/docs/gherkin/reference/. Accessed: 29/01/2020.

[21] *Engineering higher quality through agile testing practices*. https://www.atlassian.com/agile/software-development/testing/. Accessed: 03/02/2020.

[22] M. Fewster e D. Graham. *Software test automation*. Addison-Wesley Reading, 1999.

[23] M. Fowler. *Continuous integration*. https://www.martinfowler.com/articles/continuousIntegration.html. 2006.

[24] V. Garousi e J. Zhi. "A survey of software testing practices in Canada". Em: *Journal of Systems and Software* 86.5 (2013), pp. 1354–1376.

[25] D. Graham. "Technical versus non-technical skills in test automation". Em: *Conference for Tte Association for Software Testing (CAST)*. 2010, pp. 3–5.

[26] D. Graham e M. Fewster. *Experiences of test automation: case studies of software test automation*. Addison-Wesley Professional, 2012, pp. 54–65.

[27] M. Härlin. *Testing and Gherkin in agile projects*. Thesis. Linköpings universitet, 2016.

[28] *History of BDD*. https://cucumber.io/docs/bdd/history/. Accessed: 06/02/2020.

[29] D. Hoffman. "Test automation architectures: Planning for test automation". Em: *Quality Week*. 1999, pp. 37–45.

[30] B. Homès. *Fundamentals of software testing*. John Wiley & Sons, 2012.

[31] I. Hooda e R. S. Chhillar. "Software test process, testing types and techniques". Em: *International Journal of Computer Applications* 111.13 (2015).

[32] E. Horowitz e Z. Singhera. "Graphical user interface testing". Em: *Technical eport Us CC S-93-5* 4.8 (1993).

[33] *IBM Rational Robot Version 7.0.3.5*. https://www.ibm.com/support/pages/ibm-rational-robot-version-7035. Accessed: 19/11/2020.

[34] "IEEE Draft Guide: Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide)-2008 (4th edition)". Em: *IEEE P1490/D1, May 2011* (2011), pp. 1–505. DOI: 10.1109/IEEESTD.2011.5937011.

[35] *Introducing BDD*. https://dannorth.net/introducing-bdd/. Accessed: 06/02/2020.

[36] ISTQB. "ISTQB Worldwide Software Testing Practices REPORT". Em: *International Software Testing Qualifications Board* (2018).

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

[38] F. ISTQB. "Foundation Level Syllabus version 2018 v3.1". Em: *International Software Testing Qualifications Board* (2018).

[39] F. ISTQB. "Advanced Level Syllabus Agile Technical Tester version 2019 v1.0". Em: *International Software Testing Qualifications Board* (2019).

[40] J. Itkonen e K. Rautiainen. "Exploratory testing: a multiple case study". Em: *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE. 2005, 10–pp.

[41] N. Jayachandran. "Understanding roi metrics for software test automation". Em: (2005). Thesis.

[42] *Jenkins*. https://jenkins.io/doc/. Accessed: 29/01/2020.

[43] N. KAMA, Y. YAHYA, N. M. AZMI, S. ADLI e N. M. M. Z. ISMAIL. "Adopting Keyword-driven Testing Framework into Jenkins Continuous Integration Tool: iProperty Group Case Study". Em: *Recent Advances in Computer Science* (2015), pp. 44–50.

[44] M. Leotta, D. Clerissi, F. Ricca e P. Tonella. "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution". Em: *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE. 2013, pp. 272–281.

[45] B. Maciel. *Integrarion, verification, validation and qualification plan - APIS 8.9*. Rel. téc. Thales Portugal, 2017.

[46] G. J. Myers, C. Sandler e T. Badgett. *The art of software testing*. John Wiley & Sons, 2004.

[47] A. Naseer e M. Zulfiqar. *Investigating exploratory testing in industrial practice: A case study*. 2010.

[48] S. Nidhra e J. Dondeti. "Black box and white box testing techniques-a literature review". Em: *International Journal of Embedded Systems and Applications (IJESA)* 2.2 (2012), pp. 29–50.

[49] V. Oliveira. *INTEGRATION VERIFICATION VALIDATION & QUALIFICATION PLAN (IVVQP) FOR SYSTEM APIS 9*. Rel. téc. Thales Portugal, 2020.

[50] M. Ortu, G. Destefanis, B. Adams, A. Murgia, M. Marchesi e R. Tonelli. "The jira repository dataset: Understanding social aspects of software development". Em: *Proceedings of the 11th international conference on predictive models and data analytics in software engineering*. 2015, pp. 1–4.

[51] *Page Object*. https://martinfowler.com/bliki/PageObject.html. Accessed: 20/02/2020.

[52] *Page Object — Design Pattern*. https://medium.com/@nelson.souza/page-object-design-pattern-ed5f6374d32d. Accessed: 20/02/2020.

[53] *Page Object Model (POM) & Page Factory: Selenium WebDriver Tutorial*. https://www.guru99.com/page-object-model-pom-page-factory-in-selenium-ultimate-guide.html. Accessed: 20/02/2020.

[54] T. Pajunen, T. Takala e M. Katara. "Model-based testing with a general purpose keyword-driven test automation framework". Em: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 242–251.

[55] A. M. Paradkar e A. Sinha. *Generation of test cases for functional testing of applications*. US Patent 8,683,446. 2014.

[56] B. Pettichord. "Success with test automation". Em: *Proceedings of the ninth international*. 1996.

[57] S. Planning. "The economic impacts of inadequate infrastructure for software testing". Em: *National Institute of Standards and Technology* (2002).

[58] *Robot Framework*. https://robotframework.org/. Accessed: 31/01/2020.

[59] *Robot Framework vs Cucumber comparison of testing frameworks*. https://knapsackpro.com/testing_frameworks/difference_between/robotframework/vs/cucumber. Accessed: 22/09/2020.

[60] *Robot Framework vs. Selenium*. https://www.udemy.com/tutorial/robot-framework-level-1/robot-framework-vs-selenium/. Accessed: 17/11/2020.

[61] *Robot Framework VS Selenium*. https://www.expounddigital.com/robot-framework-vs-selenium/. Accessed: 03/02/2020.

[62] *Selenium2Library*. https://robotframework.org/Selenium2Library/Selenium2Library.html. Accessed: 09/10/2020.

[63] *SeleniumLibrary*. https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html. Accessed: 06/02/2020.

[64] *Site dos CTT*. https://www.ctt.pt/particulares/index. Accessed: 02/10/2020.

[65] S. Sivanandan et al. "Agile development cycle: Approach to design an effective Model Based Testing with Behaviour driven automation framework". Em: *20th Annual International Conference on Advanced Computing and Communications (ADCOM)*. IEEE. 2014, pp. 22–25.

[66] P. Sousa. *APIS V8.9 SOFTWARE DEVELOPMENT PLAN (SDP)*. Rel. téc. Thales Portugal, 2017.

[67] P. R. Srivastava, P. Patel e S. Chatrola. "Cause effect graph to decision table generation". Em: *ACM SIGSOFT Software Engineering Notes* 34.2 (2009), pp. 1–4.

[68] S. Stolberg. "Enabling agile testing through continuous integration". Em: *2009 agile conference*. IEEE. 2009, pp. 369–374.

[69] 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.

[70] Thales. *Products Factory - Teams JIRA and DDQS (Tailoring)*. Rel. téc. Thales Portugal, 2020.

[71] *Thales Group About Us*. https://www.thalesgroup.com/en/global/about-us. Accessed: 29/01/2020.

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

[73] *Thales Group History*. https://www.thalesgroup.com/en/global/group/history. Accessed: 29/01/2020.

[74] *Thales IVVQ Manager*. https://jobs.thalesgroup.com/job/madrid/ivvq-manager/1766/15197023. Accessed: 06/02/2020.

[75] *The Standish Group*. https://www.standishgroup.com/. Accessed: 12/02/2020.

[76] W.-T. Tsai, Y. Huang e Q. Shao. "Testing the scalability of SaaS applications". Em: *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2011, pp. 1–4.

[77] *Waterfall definition*. http://www.informatik.uni-bremen.de/uniform/vm97/def/def_w/WATERFALL.htm. Accessed: 14/11/2020.

ANEXO I

Anexo 1 Tabela de decisão gerada por *cause-effect graphing*

Test case #	Inputs (Causes)								Expected Outputs (Effects)
	Timespan start	Timespan end	Destination	Message type	Language	Text message	Template	Add extra message	Result
1	Empty	Empty	Filled	Free message	English	Filled		No	Message created
2	Empty	Empty	Filled	Free message	Arabic	Filled		No	Message created
3	Empty	Empty	Filled	Free message	English	Filled		Yes	Message created
4	Empty	Empty	Filled	Free message	Arabic	Filled		Yes	Message created
5	Empty	Empty	Filled	Free message	Both	Filled		Yes	Message created
6	Empty	Empty	Filled	Templates			Selected		Message created
7	Empty	> Current Date	Filled	Free message	English	Filled		No	Message created
8	Empty	> Current Date	Filled	Free message	Arabic	Filled		No	Message created
9	Empty	> Current Date	Filled	Free message	English	Filled		Yes	Message created
10	Empty	> Current Date	Filled	Free message	Arabic	Filled		Yes	Message created
11	Empty	> Current Date	Filled	Free message	Both	Filled		Yes	Message created
12	Empty	> Current Date	Filled	Templates			Selected		Message created
13	< Current Date	Empty	Filled	Free message	English	Filled		No	Message created
14	< Current Date	Empty	Filled	Free message	Arabic	Filled		No	Message created
15	< Current Date	Empty	Filled	Free message	English	Filled		Yes	Message created
16	< Current Date	Empty	Filled	Free message	Arabic	Filled		Yes	Message created
17	< Current Date	Empty	Filled	Free message	Both	Filled		Yes	Message created
18	< Current Date	Empty	Filled	Templates			Selected		Message created
19	< Current Date	> Current Date	Filled	Free message	English	Filled		No	Message created
20	< Current Date	> Current Date	Filled	Free message	Arabic	Filled		No	Message created
21	< Current Date	> Current Date	Filled	Free message	English	Filled		Yes	Message created
22	< Current Date	> Current Date	Filled	Free message	Arabic	Filled		Yes	Message created
23	< Current Date	> Current Date	Filled	Free message	Both	Filled		Yes	Message created
24	< Current Date	> Current Date	Filled	Templates			Selected		Message created
25	> Current Date	Empty	Filled	Free message	English	Filled		No	Message created
26	> Current Date	Empty	Filled	Free message	Arabic	Filled		No	Message created
27	> Current Date	Empty	Filled	Free message	English	Filled		Yes	Message created
28	> Current Date	Empty	Filled	Free message	Arabic	Filled		Yes	Message created
29	> Current Date	Empty	Filled	Free message	Both	Filled		Yes	Message created
30	> Current Date	Empty	Filled	Templates			Selected		Message created
31	> Current Date	> Current Date	Filled	Free message	English	Filled		No	Message created
32	> Current Date	> Current Date	Filled	Free message	Arabic	Filled		No	Message created
33	> Current Date	> Current Date	Filled	Free message	English	Filled		Yes	Message created
34	> Current Date	> Current Date	Filled	Free message	Arabic	Filled		Yes	Message created
35	> Current Date	> Current Date	Filled	Free message	Both	Filled		Yes	Message created
36	> Current Date	> Current Date	Filled	Templates			Selected		Message created
37	Empty	Empty	Empty						Empty fields
38	Empty	> Current Date	Empty						Empty fields
39	< Current Date	Empty	Empty						Empty fields
40	< Current Date	> Current Date	Empty						Empty fields
41	> Current Date	Empty	Empty						Empty fields
42	> Current Date	> Current Date	Empty						Empty fields
43	Empty	Empty		Templates			Not selected		Empty fields
44	Empty	> Current Date		Templates			Not selected		Empty fields
45	< Current Date	Empty		Templates			Not selected		Empty fields
46	< Current Date	> Current Date		Templates			Not selected		Empty fields
47	> Current Date	Empty		Templates			Not selected		Empty fields
48	> Current Date	> Current Date		Templates			Not selected		Empty fields
49	Empty	Empty		Free message		Empty		Yes	Empty fields
50	Empty	> Current Date		Free message		Empty		Yes	Empty fields
51	< Current Date	Empty		Free message		Empty		Yes	Empty fields
52	< Current Date	> Current Date		Free message		Empty		Yes	Empty fields
53	> Current Date	Empty		Free message		Empty		Yes	Empty fields
54	Empty	> Current Date		Free message		Empty		Yes	Empty fields
55	Empty	Empty		Free message		Empty		No	Empty fields
56	Empty	> Current Date		Free message		Empty		No	Empty fields
57	< Current Date	Empty		Free message		Empty		No	Empty fields
58	< Current Date	> Current Date		Free message		Empty		No	Empty fields
59	> Current Date	Empty		Free message		Empty		No	Empty fields
60	Empty	> Current Date		Free message		Empty		No	Empty fields
61	> Current Date	Same as							Failed

Figura I.1: Tabela de decisão resultante da aplicação da técnica *cause-effect graphing* com todos os 61 casos de teste

ANEXO II

Anexo 2 Manual de aplicação das técnicas de geração de casos de teste

Test case generation techniques

Marco Monteiro

Outubro 2020

Index

Test case generation techniques

The generation of test cases is one of the pillars of any test process of a system and its automation saves time and effort, reducing the possibility of errors and failures occurring during the execution of the system. A system is always composed of multiple functionalities and used by multiple people, so the generation of different usage flows is inevitable due to the different combinations of possible inputs and consequent generated outputs.

An application such as APIS has different possibilities to perform the same action and it is necessary to validate all of these user interactions with the system. As such, it is complicated to analyze all possible combinations of inputs and their resulting outputs.

Currently the IVVQ team responsible for the automation process use a non-systematic method to discover test cases. This method is called, exploratory testing. In sum, exploratory testing is any test where the tester controls the design of the test cases while these tests are being performed, in turn the tester uses the information gained while testing the application to be able to produce tests with higher quality and covering more possible cases. One of the major disadvantages of this technique is the fact that it relies heavily on experience and skills of the person applying it, and sometimes the qualities of the person applying it may not be enough. As such, the application of this technique is more prone to errors compared to the application of systematic techniques.

Without the existence of a systematic method that allows choosing a subset of inputs to be tested, it is difficult to know if the selected subset will not lead to the execution of tests that are not relevant, consequently resulting in an inefficient investment of resources.

To work around this problem there are several techniques for generating test cases. These techniques are diverse but all aim at creating test cases that are useful and crucial to guarantee the quality of the software produced.

Two techniques are presented that fit with the type of system that is APIS8 and the need to obtain a systematic method of generating tests and avoid the informal exploratory testing method. These techniques are: Cause-Effect Graphing and Decision Table testing.

Cause-effect graphing

Cause-effect graphing is a black box testing technique focusing only on the external behavior of the system. The technique of generating cause-effect test cases graphing presents itself as a systematic method that allows selecting a large number of possible test cases and at the same time identify ambiguities and incomplete components that have not yet been discovered when specifying application. It aims to achieve a maximum level of coverage for the different functionalities of the application in order to guarantee its quality.

It's a test case generation technique that allows us to identify possible inputs and the consequent outputs, this is possible through the construction of a graph that represents the logical relationships between inputs and outputs, which can be expressed through Boolean expressions.
There are three fundamental components when applying this technique: causes, effects and restrictions. The causes are the inputs and the effects the outputs, that is, the causes are conditions that affect the system's output and the effects are the system's responses to multiple combinations of inputs. Constraints are limitations built into the system.

The process begins with the identification of the causes, effects and restrictions of the system. Below is the construction of the graphs that represent a combination of logic connectors composed by us, these being the causes and effects, and arcs that establish the connection between the nodes. Arcs are Boolean operators, "and", "or" and "not". The last step of the application of this technique consists of an analysis of all usage flows represented in the graphs leading to the construction of a decision table. Each column in the table represents a test case that can potentially be converted to code in order to test a functionality.

Example

To illustrate the application of the Cause-effect graphing technique we will use the scenario concerning messages to passengers extracted from the STD of the Thales APIS8 project.
In particular, for this example the selected functionality is that of “Create text messages” via free text or template.
Figure 1 is taken directly of the APIS8 system of the scenario associated with the creation of messages to passengers.

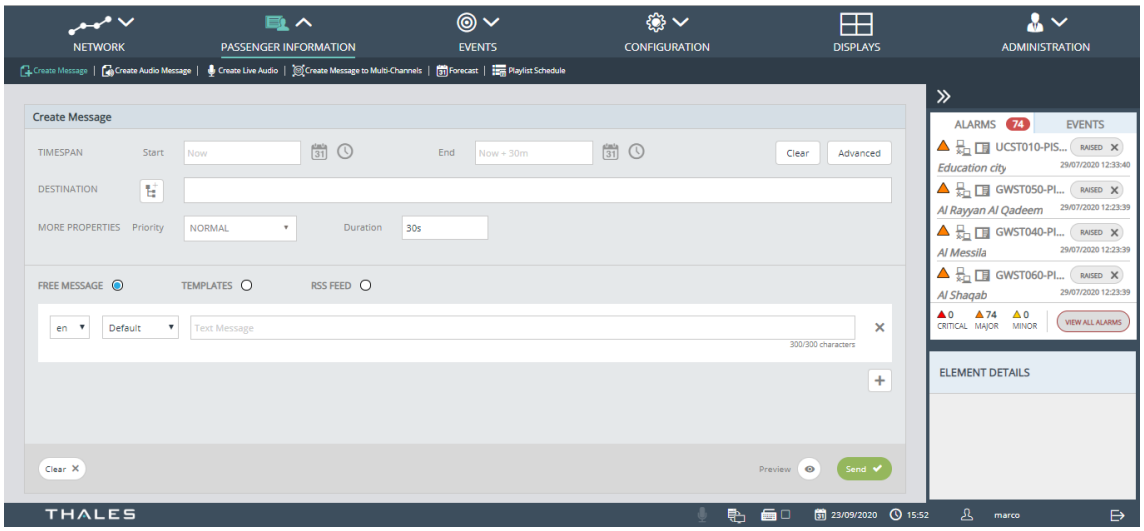


Figure 1 - Create text message functionality

Step 1

Before starting to apply the technique, it is essential to identify which elements are present in the HMI that will be interacted during the application of the techniques. These elements are text boxes ready for inputs, and buttons for example.

In this example, they are:

- Timespan start/end;
- Destination;
- Message type;
- Add extra message;
- Language.

Step 2

Next step is to identify the causes, effects and restrictions.
The causes, represented by "C", are:

- C1: Timespan start = empty;
- C2: Timespan start < current date;
- C3: Timespan start > current date;
- C4: Timespan end = empty;
- C5: Timespan end < current date;
- C6: Timespan end > current date;
- C7: Destination = filled;
- C8: Destination = empty;
- C9: Message type = free message;
- C10: Message type = templates;
- C11: Message type = rss feed;
- C12: Language = english;
- C13: Language = arabic;
- C14: Text message = filled;
- C15: Text message = empty;
- C16: Template = selected;
- C17: Template = not selected;
- C18: Timespan start = same as timespan start and time;
- C19: Add extra message = yes;
- C20: Add extra message = no.

The effects, represented by "E", are:

- E1: Message created;
- E2: Empty fields;
- E3: Failed to create message.

The restrictions are:

- O (Exactly one must be true);
- E (At most one).

Step 3

After identifying the causes, effects and restrictions it's time to start drawing the graphs. To do this, we connect the different causes in order to obtain a certain effect. For this example, we draw three graphs, one for each effect. In this demo the effect where the message is successfully created is displayed below.

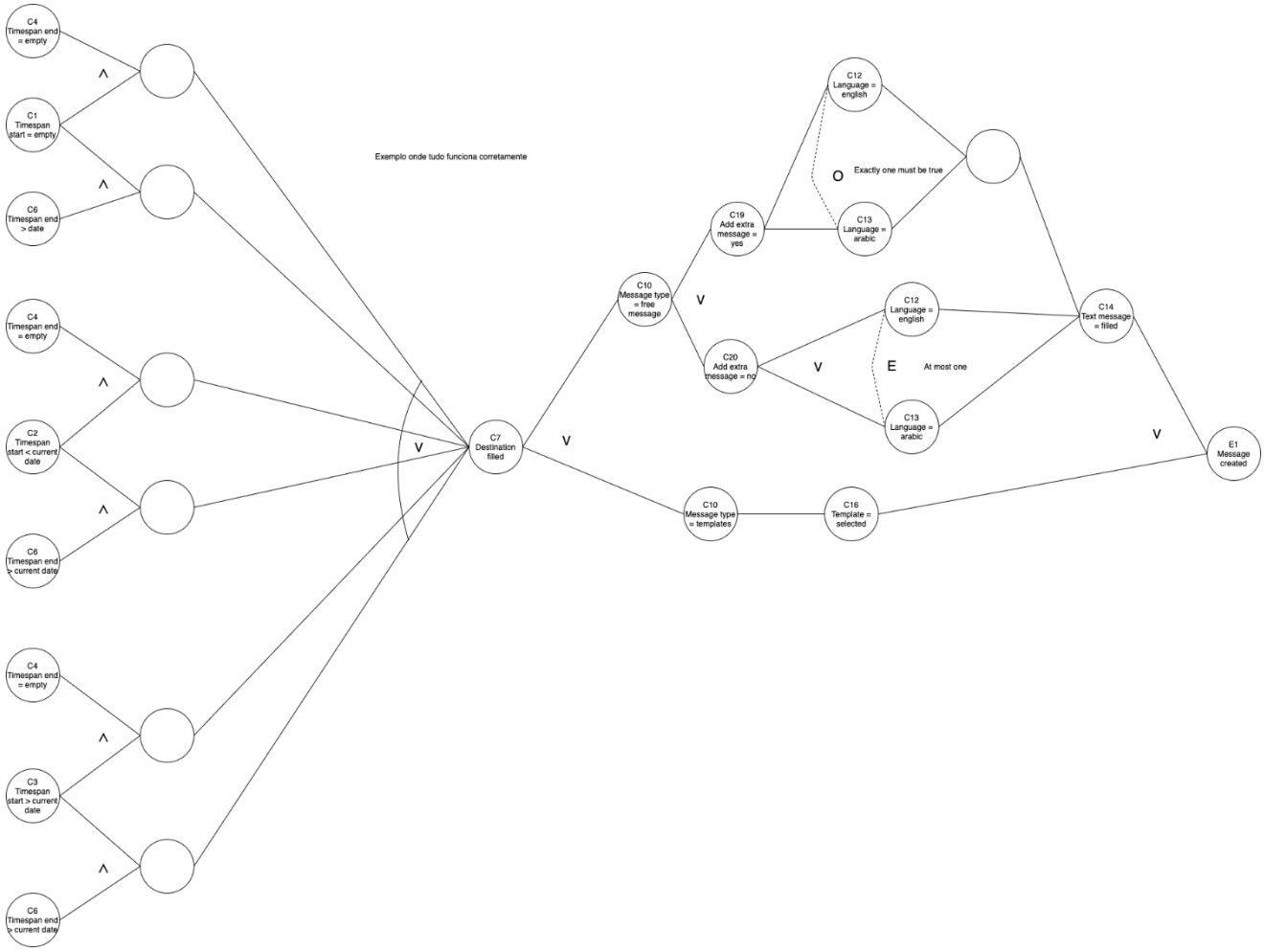


Figure 2 - Cause-effect graph for the effect of created message

An analysis of the graph in figure 2 is now presented:

- It is possible to observe on the left side, the multiple possible combinations between start and end dates;
- For the effect that we intend, any possible combination of the present dates is valid;
- The destination must be filled in for the message to be created with success;
- After cause 7, destination filled, the graph diverges in two possibilities, either a free or template message is sent;
- If the template is selected, it is mandatory to select one of the several available, and then the message will therefore be created successfully;
- If it is a free text message, there is the possibility to add one or two text messages. If it's just a single message we can choose whether it's written in Arabic or English, but at most only one of them, hence the constraint E (at most one). If there are two text messages, there is a possibility to create a message in English and the other in Arabic, or, create both in the same language, hence the restriction O (exactly one must be true), since at least one of the languages must be selected;
- Before creating the message, it is essential to ensure that the text fields are in fact filled, if they are, then the message is created successfully.

Step 4

The construction of the decision table follows with all the identified branches through the analysis of the graph previously drawn. Each column in the table is a test case.
The table presented here is just an excerpt from the mother table where the remaining possible ramifications for the effect of the created message are present, as well as the remaining effects. In total, 61 test cases were calculated for the scenario studied.

Test case	1	2	3
Causes:			
1(Timespan start = empty)	1	1	1
2(Timespan start < current date)	0	0	0
3(Timespan start > current date)	0	0	0
4(Timespan end = empty)	1	1	1
5(Timespan end < current date)	0	0	0
6(Timespan end > current date)	0	0	0
7(Destination = Filled)	1	1	1
8(Destination = Empty)	0	0	0
9(Message type = Free message)	1	1	1
10(Message type = Templates)	0	0	0
11(Message type = RSS Feed)	0	0	0
12(Language = english)	1	0	1
13(Language = arabic)	0	1	0
14(Text message = filled)	1	1	1
15(Text message = empty)	0	0	0
16(Template = Selected)	0	0	0
17(Template = Not selected)	0	0	0
18(Timespan end = same as timespan start and time)	0	0	0
19(Add extra message = yes)	0	0	1
20(Add extra message = no)	1	1	0
Effects:			
1(Message created)	1	1	1
2(Empty fields)	0	0	0
3(Failed to create message)	0	0	0

Figure 3 - Decision Table

This table is composed of the causes, in green, and the effects in orange.
It is built as follows:
- analyze each possible branch of the graph;
- set the value to 1 if the cause and effect occurred in that branch and zero if not.

Step 5

For the purpose of improving the interpretation of the table, it was converted into the following.

Test case #	Inputs (Causes)								Expected Outputs (Effects)
	Timespan start	Timespan end	Destination	Message type	Language	Text message	Template	Add extra message	Result
1	Empty	Empty	Filled	Free message	English	Filled		No	Message created
2	Empty	Empty	Filled	Free message	Arabic	Filled		No	Message created
3	Empty	Empty	Filled	Free message	English	Filled		Yes	Message created
4	Empty	Empty	Filled	Free message	Arabic	Filled		Yes	Message created
5	Empty	Empty	Filled	Free message	Both	Filled		Yes	Message created
6	Empty	Empty	Filled	Templates			Selected		Message created
7	Empty	> Current Date	Filled	Free message	English	Filled		No	Message created
8	Empty	> Current Date	Filled	Free message	Arabic	Filled		No	Message created
9	Empty	> Current Date	Filled	Free message	English	Filled		Yes	Message created
10	Empty	> Current Date	Filled	Free message	Arabic	Filled		Yes	Message created
11	Empty	> Current Date	Filled	Free message	Both	Filled		Yes	Message created
12	Empty	> Current Date	Filled	Templates			Selected		Message created
13	< Current Date	Empty	Filled	Free message	English	Filled		No	Message created
14	< Current Date	Empty	Filled	Free message	Arabic	Filled		No	Message created
15	< Current Date	Empty	Filled	Free message	English	Filled		Yes	Message created
16	< Current Date	Empty	Filled	Free message	Arabic	Filled		Yes	Message created

Figure 4 - Decision Table with simplified interpretation

This conversion consists of multiple steps, they are:

1. Analyze the original decision table;
2. For each cause and effect, simplify its presentation, that is, in this simplified table you can see that for example, one of the causes is simply "Destination" and in each cell we fill in whether the value is Filled or Empty according to if we put 1 or 0 in the respective cause. Changing 1's and 0's for text makes the table more understandable and easier to extract the values for each test case;
3. Identify which causes are not part of certain branches. Per example, in test case 6, as a template was selected, the option to add an extra free text message does not exist. Then the cell value is omitted.

The table shown is an excerpt from the parent table which, as previously mentioned has 61 test cases, for this demonstration the 16 test cases are presented for the effect of successfully created message.

Conclusion

This was the last step of applying the Cause-Effect Graphing technique. However, the application of this technique has generated 61 test cases, which in turn are not entirely crucial, many of these tests are redundant. This is because certain values of input will not affect the output, so these values can be suppressed, thereby reducing the need to validate the value of these inputs. So, it is important to apply a test case generation technique that will allow to find out which tests are crucial to be tested in the application thus ensuring the correct functioning of the system. The technique selected for this is Decision Table Testing.

Decision Table Testing

The decision table testing technique can be applied independently or together with the cause-effect graphing technique. As explained in the cause-effect technique, the final phase is the generation of a decision table. However, it is possible to reduce the number of test cases present in this decision table, reducing the tests only to the more crucial. This is because there are several test cases in which certain inputs do not have any impact on the system output, these inputs can be suppressed/ignored. This technique allows to present a table that represents in a very clear way, which crucial test cases are to be created, and presents the values of inputs and its effects in a very easily interpretable way.

Example

Step 1

The following example is based on the decision table generated upon completion of the application of the cause effect technique. Before proceeding with the application of the technique it is important to explain that there was a transformation of the table generated by the cause effect for this now presented. The changes are:

- The causes adopt the style of interrogations so that the values inserted in the table cells are true or false (Boolean values);
- The bottom line of the table indicates the effect being studied, if the message is created or not. In order to simplify this demonstration, it was assumed that only matters whether the message is created or not, and if the message is not created it may be due to empty fields or errors in the insertion data;
- Cells with "-" means that this value does not exist for the test being tested, that is, it is not a step performed during the test execution.

Figure 5 is a fragment of the mother table where 61 test cases are being evaluated.

Conditions	Timespan start empty?	T	T	F	F	F	F
	Timespan start < current date?	F	F	T	T	F	F
	Timespan start > current date?	F	F	F	F	T	T
	Timespan end empty?	T	F	T	F	T	F
	Timespan end < current date?	F	F	F	F	F	F
	Timespan end > current date?	F	T	F	T	F	T
	Same as?	F	F	F	F	F	F
	Destination filled?	F	F	F	F	F	F
	Free message?	-	-	-	-	-	-
	Template selected?	-	-	-	-	-	-
	Language English?	-	-	-	-	-	-
	Text message filled?	-	-	-	-	-	-
	Extra message?	-	-	-	-	-	-
Actions	Message created?	F	F	F	F	F	F

Figure 5 - Decision Table with section related to the test of empty destination

Step 2

We start by identifying the input values that appear to have no impact on the output.

It can be seen that the values of timespan start and end have no impact in the final effect, since this is always false. As such, we will mark in yellow the cells that have timespan values.

Conditions	Timespan start empty?	T	T	F	F	F	F
	Timespan start < current date?	F	F	T	T	F	F
	Timespan start > current date?	F	F	F	F	T	T
	Timespan end empty?	T	F	T	F	T	F
	Timespan end < current date?	F	F	F	F	F	F
	Timespan end > current date?	F	T	F	T	F	T
	Same as?	F	F	F	F	F	F
	Destination filled?	F	F	F	F	F	F
	Free message?	-	-	-	-	-	-
	Template selected?	-	-	-	-	-	-
	Language English?	-	-	-	-	-	-
	Text message filled?	-	-	-	-	-	-
	Extra message?	-	-	-	-	-	-
Actions	Message created?	F	F	F	F	F	F

Figure 6 - Decision Table with cells to be surpressed

Step 3

After the identification of irrelevant cells, we can suppress them and replace the values of V and F by “-”, that is, the value of that input does not affect the output to be tested.

When executing this action, it is also possible to observe that the line that has the condition/cause "Destination filled?" always has a false value, so we can suppress the six columns and summarize to just one test case.

Conditions	Timespan start empty?	-
	Timespan start < current date?	-
	Timespan start > current date?	-
	Timespan end empty?	-
	Timespan end < current date?	-
	Timespan end > current date?	-
	Same as?	-
	Destination filled?	F
	Free message?	-
	Template selected?	-
	Language English?	-
	Text message filled?	-
	Extra message?	-
Actions	Message created?	F

Figure 7 - Decision Table with cells surpressed

The end result is just a test case, where the important thing is to test the fact that if we insert an empty destination in the creation of the message, it is not created. This whole process is iterative, and as such, the entire table will be analyzed with the aim to reduce the number of test cases to only those that are critical to test.

Step 4

Now we just need to repeat step 1 and 2 until all unnecessary cells are suppressed.

The final result is figure 8. The cells with "-" and yellow are the cells where there is no impact no matter the value of the input. The ones that only have "-" are the cells where we cannot enter any input values.
In short, of the 61 potential test cases, it was possible to reduce to just 6 that are essential to be tested.

Conditions	Timespan start empty?	-	-	-	-	-	-
	Timespan start < current date?	-	-	-	-	-	-
	Timespan start > current date?	-	-	-	T	-	-
	Timespan end empty?	-	-	-	-	-	-
	Timespan end < current date?	-	-	-	-	-	-
	Timespan end > current date?	-	-	-	-	-	-
	Same as?	F	F	F	T	F	F
	Destination filled?	F	-	-	-	T	T
	Free message?	-	F	T	-	T	F
	Template selected?	-	F	-	-	-	T
	Language English?	-	-	-	-	-	-
	Text message filled?	-	-	F	-	T	-
	Extra message?	-	-	-	-	-	-
Actions	Message created?	F	F	F	F	T	T

Figure 8 - Final Decision Table

This concludes the application of both techniques. In the following chapter it is possible to consult a simplified step by step on how to apply these techniques.

Step by step manual for applying test case generation techniques

Cause-effect graphing

1. Identify HMI elements which inputs must be evaluated;
2. Identify possible values for inputs;
3. Identify possible outputs;
4. Identify restrictions of the system;
5. Note down all causes (inputs), effects (outputs), restrictions;
6. Create the cause-effect graph linking logic operators between causes and effects;
7. Select a branch of the graph, each branch from start to finish is a test case (only ends on an effect node);
8. Mark down with value 1 if the cause and effect occur in that branch, mark with 0 if not;
9. Repeat step 7 and 8 until all branches have been analyzed;
10. Simplify decision table (how to simplify is explained in step 5 on the Cause-effect graphing chapter).

Decision table testing

1. Transform decision table generated from cause-effect graphing technique to a table that adjusts to the appliance of decision table testing technique (how to explained in decision table testing chapter, step 1);
2. Check which input values do not have any impact on the output;
3. Mark with a different color the cells with disposable input values;
4. Replace those cells with “-”;
5. If possible collapse columns, resulting in the decrease of the number of test cases;
6. Repeat steps 2 to 5 until all irrelevant inputs are removed;
7. Mark with “-” and a different color the cells where there is no impact no matter the value of the input;
8. Mark with “-” the cells where it’s impossible to enter any input values in that test case;