UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Automatização de Testes de Software

Inês Chaves Batista

Mestrado em Segurança Informática

Trabalho de Projeto orientado por:
João Pedro Neto

2024

Agradecimentos

Em primeiro lugar, gostaria de expressar a minha sincera gratidão ao meu orientador João Pedro Guerreiro Neto que me guiou, instruiu e motivou. O seu *feedback* permitiu aprofundar e refinar a minha pesquisa, e os resultados apresentados no meu trabalho de projeto seriam impossíveis sem a sua supervisão.

Gostaria também de agradecer o apoio fornecido pela empresa CGI em todo o decorrer deste trabalho, em particular ao Pedro de Almeida Perdigão e aos meus colegas de equipa que se mostraram sempre disponíveis para me ajudar em todo o desenvolver do projeto.

Por último, gostaria de agradecer à minha família e amigos próximos que me motivaram, aconselharam e apoiaram durante todo este processo.

Resumo

A fase de teste é uma das etapas mais demoradas no ciclo de desenvolvimento de software [14]. A execução repetida de testes, a manutenção de scripts de teste e a comparação dos resultados esperados com os resultados reais são tarefas comuns, mas essenciais para garantir a qualidade do software, consumindo rotineiramente grandes quantidades de tempo a equipa de testes. Essa demanda constante de tempo e recursos pode limitar o processo de desenvolvimento, aumentando o tempo de entrega e, em alguns casos, o custo total do projeto. Para mitigar esses desafios, a automatização e as metodologias ágeis de desenvolvimento de software oferecem uma série de vantagens, como *feedback* imediato, maior eficácia no desenvolvimento, menos erros e defeitos, colaboração contínua e melhor qualidade de código [18, 57]. Além disso, a automatização permite que os testes sejam executados com mais frequência e precisão, garantindo que as mudanças no código são verificadas imediatamente, contribuindo para a deteção precoce de erros. A fase de testes é crucial para todas as empresas de desenvolvimento de software, pois assegura que o produto final atenda às expectativas de qualidade e funcionalidade, reduzindo o risco de falhas pós-lançamento, que podem provocar altos custos de manutenção e danos à reputação. Contudo, é importante notar que a automatização de testes, por si só, não é suficiente para garantir a qualidade do software; ela deve ser integrada num domínio mais amplo de *Quality Assurance* (QA), ou Garantia de Qualidade, que abrange práticas e processos para assegurar a excelência do produto final [28]. Essa integração é vital para que a automatização contribua efetivamente para a melhoria contínua dos processos de desenvolvimento. Este trabalho de projeto apresenta um projeto realizado em parceria com uma consultoria de grande escala, cujo objetivo foi automatizar testes de software utilizando a *framework* SpecFlow, uma ferramenta que suporta a metodologia ágil de desenvolvimento *Behavior-Driven Development* (BDD).

Palavras-chave: Automatização, Testes de software, Behaviour Driven Development (BDD), SpecFlow, Qualidade de software

Abstract

The testing phase is one of the most time-consuming stages in the software development cycle [14]. The repeated execution of tests, the maintenance of test scripts, and the comparison of expected results with actual outcomes are common but essential tasks to ensure software quality. These tasks routinely consume large amounts of time for the software testing team. This constant demand for time and resources can often become a limiting factor in the development process, increasing delivery time and, in some cases, contributing to the overall project costs. Therefore, automation and agile software development methodologies help facilitate these tasks, as they quickly translate into a series of advantages for all development team members, including immediate feedback, increased development efficiency, fewer errors and defects, continuous collaboration, and improved code quality [18, 57]. Additionally, automation allows tests to be executed more frequently and accurately, ensuring that changes in the code are immediately verified, which contributes to the early detection of errors and minimizes rework.Moreover, the testing phase is crucial for all software development companies, as it ensures that the final product meets quality and functionality expectations, reducing the risk of post-launch failures, which can result in high maintenance costs and damage to reputation. However, it is important to note that when it comes to software quality, test automation alone is not sufficient; it must be integrated into a broader domain of Quality Assurance (QA), which encompasses a variety of practices and processes to ensure the excellence of the final product [28]. This integration is essential for automation to truly contribute to the continuous improvement of development processes. This work presents a project carried out in partnership with a large-scale consulting company, which aimed at automating software testing using the SpecFlow framework, a tool that supports the agile software development methodology known as Behavior-Driven Development (BDD).

Keywords: Automation, Software Testing, Behavior Driven Development (BDD), SpecFlow, Software Quality

Conteúdo

Lista de Figuras x

Abreviaturas xv

1 Introdução 1

1.1 Motivação . . . 2
1.2 Objetivos . . . 4
1.3 Estrutura do documento . . . 5

2 Fundamentação teórica 7

2.1 Software Development Life Cycle (SDLC) . . . 8
2.1.1 Modelos SDLC . . . 10
2.2 Software Testing Life Cycle (STLC) . . . 11
2.3 Testes de Software . . . 12
2.4 Processo de Testes . . . 13
2.5 Princípios na Realização de Testes . . . 14
2.6 Níveis de Testes de Software . . . 15
2.7 Tipos de Testes de Software . . . 16
2.8 Testes Manuais e Testes Automáticos . . . 17
2.9 Metodologias Ágeis de Desenvolvimento de Software . . . 19
2.9.1 Test-Driven Development (TDD) . . . 20
2.9.2 BDD (Behaviour Driven Development) . . . 21
2.10 Frameworks que suportam BDD . . . 23

3 Análise 25

3.1 Ferramentas e linguagens de desenvolvimento . . . 25
3.1.1 Microsoft Azure Devops . . . 25
3.1.2 BDD (Behaviour Driven Development) . . . 28
3.1.3 Linguagem Gherkin . . . 30
3.1.4 Framework SpecFlow . . . 31

4 Estratégia de testes **35**
4.1 Arquitetura de software . . . 35
4.2 Âmbito da estratégia . . . 37
4.2.1 Metodologia de desenvolvimento . . . 37
4.2.2 Equipa do projeto . . . 39
4.2.3 Processo de testes . . . 41
4.3 Abordagem . . . 42
4.4 Pipeline CI/CD . . . 42
4.4.1 Integração contínua (CI) . . . 42
4.4.2 Entrega contínua (CD) . . . 43
4.4.3 Pipeline CI/CD . . . 43
4.4.4 Azure Pipeline . . . 44

5 Implementação **47**
5.1 Plano de testes e Desenho . . . 47
5.1.1 User story e criterios de aceitação . . . 47
5.1.2 Desenho . . . 49
5.2 Implementação . . . 50
5.3 Execução dos testes . . . 53
5.4 Testes realizados . . . 56
5.5 Gestão de defeitos, alterações e reteste . . . 57

6 Avaliação **60**
6.1 Métricas de Avaliação e Desempenho . . . 60
6.2 Resultados dos KPIs . . . 62
6.3 Automatização e a sua Importância . . . 63

7 Conclusão **67**

Bibliografia **74**

Lista de Figuras

2.1 Defeito, erro e falha em relação a severidade e ao tempo 7
2.2 Software Development Life Cycle (SDLC) 8
2.3 Software Testing Life Cycle (STLC) 12
2.4 Níveis de testes de software, em relação ao tempo e custo/esforço 15
2.5 Diagrama de testes manuais 17
2.6 Diagrama de testes automáticos 18
2.7 Test-Driven Development (TDD) 20
2.8 Behavior-Driven Development (BDD) 22

3.1 Ciclo do DevOps 26
3.2 Serviços do Microsoft Azure DevOps 27
3.3 Exemplo de código em linguagem Gherkin 30
3.4 Estrutura do Specflow 32

4.1 Arquitetura de Software 36
4.2 Processo Scrum 38
4.3 Diagrama da equipa do projeto 39
4.4 Processo de testes 41
4.5 Pipeline CI/CD 44
4.6 Workflow da Azure DevOps Pipeline 44

5.1 Diagrama de cenários de teste 49
5.2 Estrutura do Test Plan no Azure 50
5.3 Estrutura do projeto de teste 51
5.4 Código do ficheiro .feature 52
5.5 Código do ficheiro .cs 53
5.6 Execução de testes no Visual Studio 54
5.7 Stages da pipeline 55
5.8 Parte do *dashboard* do resultado da execução dos testes 55
5.9 Progress Report dos testes 56

Lista de Tabelas

4.1 Conceitos chaves de uma pipeline do Azure Devops 45

5.1 Quantidade de testes e tipo de validações realizadas 57
5.2 Quantidade de testes e tipo de testes realizados 57

6.1 Resultado dos KPIs selecionados 62

Abreviaturas

BA Business Analytics.

BDD Behavior-Driven Development.

BSC Balancing and Settlement Code.

CD Entrega Contínua.

CI Integração Contínua.

DSL Domain Specific Language.

GoCD Go Continuous Delivery.

IaC Infrastructure as Code.

ISD Industry Standing Data.

ISTQB International Software Testing Qualifications Board.

IT Information Technology.

KPIs Key Performance Indicators.

MDD Market Domain Data.

MHHS Market-wide Half Hourly Settlement.

NETA New Electricity Trading Arrangements.

PO Product Owner.

QA Quality Assurance.

SDLC Software Development Life Cycle.

STLC Software Testing Life Cycle.

TA Technical Arquitect.

TDD Test-Driven Development.

TFVC Team Foundation Version Control.

Capítulo 1

Introdução

Ao longo do último século, a tecnologia passou por um rápido desenvolvimento, transformando profundamente a forma como a sociedade aprende e pensa. Como resultado, a sociedade estabelece uma relação de interdependência e é influenciada pela tecnologia e pelos seus avanços científicos. Em particular, nas últimas duas décadas, a indústria do software experimentou um crescimento significativo, tornando-se maior, mais competitiva e com um número cada vez maior de utilizadores [1].

Seja através da interação direta com o utilizador, ou coordenando silenciosamente processos terciários, os sistemas de software tornaram-se no nosso quotidiano tendo vindo a assumir progressivamente mais importância tanto no plano económico como pessoal. Este aumento de importância foi seguido de perto por evoluções nas metodologias de desenvolvimento, melhorias nas linguagens e plataformas [6].

Apesar de toda esta evolução, acredita-se que a maioria das pessoas já teve uma experiência com software que não funcionou como esperado. De forma visível ou não, software que não funciona corretamente pode levar a vários problemas, incluindo perda de dinheiro, tempo ou reputação comercial [6]. Assim, e para mitigar ou prevenir estas falhas, diversos mecanismos de testes de software são rotineiramente executados durante o desenvolvimento de software.

Os testes de software são umas das parte menos compreendida do processo de desenvolvimento e são muitas vezes comparados à descoberta de *bugs*. Deste modo, e de acordo com James Whittaker [63], os testes de software são o processo de execução de um sistema de software de forma a determinar se este corresponde à sua especificação e se é executado corretamente no ambiente pretendido.

No entanto, os testes de software podem ser dispendiosos e trabalhosos [1], visto que o processo de testagem é difícil e demorado e requer uma certa sofisticação técnica e um planeamento adequado [6].

Mesmo que um software seja simples este irá sempre apresentar obstáculos para a equipa de testes. Por esta razão, ao longo dos anos foram desenvolvidos métodos que facilitam esta tarefa, sendo um deles a automatização [6]. A automatização de testes não apenas reduz o custo do teste, mas também reduz o erro humano, tornando todo o processo de teste mais facilmente exequível e permitindo que os testes sejam executados repetidamente e de forma mais eficiente [1].
Neste contexto, o presente trabalho concentra-se na investigação e desenvolvimento de técnicas avançadas para a automatização de testes de software.

1.1 Motivação

Fundada em 1976, a CGI é uma das maiores empresas de consultoria e de serviços de IT do mundo. Esta possui parceria com mais de 150 empresas de tecnologia [10], sendo uma delas uma empresa chave no setor elétrico do Reino Unido que possui mais de 150 funcionários.
A empresa em questão foi criada com o intuito de gerir o *Balancing and Settlement Code* (BSC), um acordo *multi-party* que define as regras e a governança do mecanismo de balanceamento e dos processos de liquidação de desequilíbrios relacionados com a quantidade de eletricidade que os geradores e fornecedores dizem que produzem ou usam e os valores reais. Esta surgiu antes dos *New Electricity Trading Arrangements* (NETA) terem entrado em vigor em março de 2001 e desde então, tornou-se uma entidade chave no setor de eletricidade do Reino Unido. A empresa é uma entidade sem fins lucrativos, financiada por *shareholders* do mercado de eletricidade, que se intitula uma especialista de mercado confiável, independente e fidedigna e que procura continuamente evoluir e inovar para que os seus clientes e consumidores possam ser beneficiados.
Contudo, o setor de energia do Reino Unido está a passar por uma grande transformação de forma a construir um sistema de energia mais ecológico e justo para os seus consumidores. Uma das principais atividades do programa de transformação é implementar o *Market-wide Half Hourly Settlement* (MHHS). O MHHS introduz um mecanismo mais detalhado e preciso para medição e faturação de eletricidade. Visto que tradicionalmente, o consumo de eletricidade é medido e relatado mensalmente ou até mesmo trimestralmente, o MHHS recorre ao uso de medidores inteligentes e avançados, em inglês *"Smart and Advanced Meters"*, que registam o consumo de eletricidade de um cliente em intervalos de meia hora ao longo do dia, denominados por *"half hourly settlement"* em inglês. O que significa que o consumo de eletricidade é medido e relatado a cada 30 minutos, fornecendo uma visão muito mais detalhada e em tempo real dos padrões de consumo.
Deste modo, e devido ao programa de transformação para implementar o MHHS, surgiu um projeto em parceria com a CGI que tem como objetivo fornecer três serviços principais e a manutenção do *Industry Standing Data*[1] (ISD). Para tal, a CGI estabeleceu um *Software Development Center* dedicado ao cliente, combinando os seus *hubs* no Reino Unido e em Portugal. Os projetos

[1] O ISD (*Initial Settlement Data*) refere-se aos dados preliminares usados no processo de liquidação de energia logo após o período de entrega. Este estima o consumo e a geração de eletricidade, permitindo que os encargos financeiros sejam calculados entre as partes envolvidas, ajudando a manter o equilíbrio entre oferta e demanda no mercado de energia. Esses dados são posteriormente ajustados com leituras mais precisas.

de desenvolvimento estão a utilizar a plataforma Azure, que consiste em .NET Core, Kubernetes, Azure Pipelines e GoCD (*Go Continuous Delivery*), e para a camada front-end, é utilizado o Salesforce, que se conecta aos serviços .NET. Complementarmente, a equipa opera usando a *framework* ágil Scrum, que fornece estreita colaboração e alinhamento entre todas as subequipas.

As principais metas e objetivos do programa de transformação são:

- Implementar e concluir novos serviços e mudanças nos sistemas centrais existentes de forma a permitir o fornecimento do MHHS;
- Implementar mudanças nos processos e aplicações afetados pelos novos serviços;
- Remover processos e serviços tornados obsoletos devido a implementação de novos serviços;
- Identificar, definir e fazer alterações no BSC e nos seus documentos de suporte conforme definido pelo MHHS;
- Definir o plano de migração para quando os medidores inteligentes (*smart meters*) passarem para a liquidação de meia hora (*half hourly settlement*);
- Definir o plano de qualificação do participante, em inglês *"participant qualification plan"*, para todos os participantes do MHHS.

Adicionalmente, o projeto em parceria com a CGI, que visa a implementação do MHHS, foi divido em quatro *work packages*, onde cada um possui os seguintes elementos:

- Definição de requisitos, incluindo definição das *user stories* associadas, critérios de aceitação e características não funcionais;
- Design e construção do sistema;
- Design e construção do ambiente de desenvolvimento no Azure;
- Colaboração com o cliente no desenho de processos e práticas de trabalho;
- Testes unitários;
- Testes de sistema;
- Criação de todos os dados de teste para o respetivo *work package*;
- Testes de integração de componentes dentro do respetivo *work package*;
- Fornecimento de suporte para quaisquer outras atividades de teste do programa;
- Fornecimento de correções de defeitos e suporte durante as fases de teste e transição da indústria.

Pondo-se isto, e no âmbito do trabalho de projeto de mestrado, surgiu a oportunidade de trabalhar na empresa CGI, neste projeto de grande escala. O projeto a desenvolver consiste na automatização dos testes de software recorrendo ao uso de ferramentas automatizadas, e este está incluído no *work package* 3. Este *work package* consiste no desenvolvimento da componente ISD e nas alterações relacionadas ao ISD no Portal do Cliente existente. A componente ISD é responsável por gerir os dados do setor e fornecê-los a outros serviços (internos e externos) em suporte ao MHHS. O serviço ISD pode ser considerado uma evolução do serviço existente de *Market Domain Data*[2] (MDD), onde algumas entidades atualmente mantidas dentro do MDD passarão para o ISD e, além disso, o ISD manterá um conjunto de novas entidades específicas do MHHS. Por fim, a componente ISD irá suportar um novo processo de fluxo de trabalho para gerir alterações nos dados que esta contém, sendo esse processo automatizado sempre que possível, e esta será desenvolvida usando tecnologias modernas, como o Salesforce e o Azure.

1.2 Objetivos

Deste modo, e visto que a automatização permite realizar testes de software de forma mais rápida e ágil, o trabalho a desenvolver neste trabalho de projeto tem como objetivo a realização de testes automáticos em um projeto de grande escala recorrendo a diversos métodos de automatização.
Em mais detalhe, este trabalho de projeto tem como objetivos:

- Efetuar uma pesquisa bibliográfica para identificação do estado de arte do tema dos testes de software, considerando a fundamentação teórica do trabalho a realizar;
- Comparar os testes manuais e automatizados apresentando as suas vantagens e desvantagens;
- Identificar a importância e os desafios (prós e contras) da automatização;
- Apresentar o contexto organizacional onde o projeto de automatização de testes de software a desenvolver se enquadra;
- Descrever os processos de desenvolvimento e de testes, assim como as tecnologias e ferramentas utilizadas no projeto;
- Estudar como a automatização de testes auxiliou a equipa associada ao projeto;
- Desenvolvimento e implementação de testes automáticos com a ferramenta de automatização;
- Compreender as melhorias, tirar métricas e fazer *benchmarking*;
- Entender a razão pela qual a ferramenta foi introduzida em comparação com outras.

[2] O MDD (*Market Domain Data*) é um conjunto de dados utilizado no setor de energia, principalmente no mercado de eletricidade, para garantir o bom funcionamento e a coordenação entre os diversos agentes do mercado, como distribuidores, fornecedores, operadores de rede, entre outros.

1.3 Estrutura do documento

Este trabalho de projeto é estruturada em sete capítulos abrangentes, cada um abordando aspetos distintos da pesquisa, seleção, implementação e avaliação de uma estrutura de testes automatizados. As secções a seguir fornecem uma visão geral de cada capítulo, detalhando os principais componentes e áreas de foco.
Os capítulos seguintes deste documento estão organizados da seguinte forma:

- No Capítulo 2, serão apresentados todos os conceitos e definições necessárias para melhor entender o problema que será abordado no âmbito deste projeto, a automatização de testes de software;
- No Capítulo 3, será realizada uma análise ao problema, explicando em detalhe as ferramentas e linguagens de desenvolvimento que foram escolhidas para a resolução do mesmo;
- No Capítulo 4, vai ser descrito o âmbito da estratégia de teste definida para o projeto, descrevendo os papéis e responsabilidades presentes na estratégia, a metodologia de desenvolvimento aplicada e as fases e atividades do processo de testes.
- No Capítulo 5, será explicado, em detalhe, como a solução para o problema apresentado foi implementada no projeto.
- No Capítulo 6, serão referidas as métricas escolhidas para medir o impacto da implementação dos testes automatizados no projeto, apresentando o resultado e a conclusão das mesmas.
- Finalmente, no Capítulo 7, serão apresentadas as principais conclusões do trabalho realizado e quais serão os próximos passos a dar no futuro para melhorar o trabalho desenvolvido.

Capítulo 2

Fundamentação teórica

Segundo Paul Ammann e Jeff Offutt em *"Introduction to Software Testing"* [1], o desenvolvimento de software pode ser comparado à engenharia, sendo que, paralelamente é uma indústria de software em que falhas e erros podem resultar na perda de vidas, como por exemplo pacientes que são irradiados com raios X e morrem devido a uma falha no software da máquina. Assim sendo, compreender a diferença entre erros, defeitos e falhas é de extrema importância para entender os problemas associados a soluções de software [1].
Igor B. Shubinsky e Hendrik Schabe [49] realizaram um estudo com o objetivo de melhor definir estes conceitos. Segundo os autores, as seguintes definições foram apresentadas (Figura 2.1):

O **defeito** é definido como a incapacidade do sistema de realizar a operação necessária em toda a sua extensão que pode levar o sistema a entrar em um estado inválido (erro).

O **erro** é uma discrepância entre um valor ou condição calculado, observado ou medido e um valor ou condição verdadeiro, especificado ou teoricamente correto. Este desvio, quando presente, pode sob certas condições, transformar-se em uma falha.

Uma **falha** está sempre associada à perda de uma função necessária à execução do software conforme os parâmetros exigidos.

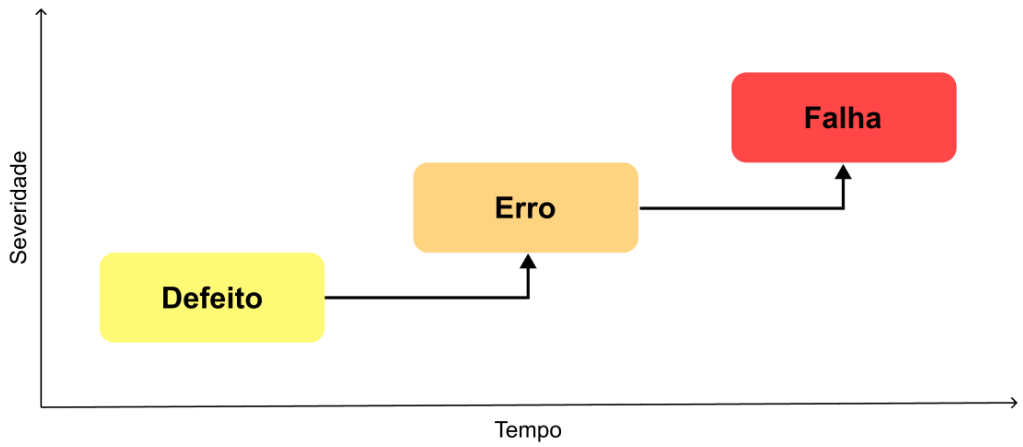

Figura 2.1: Defeito, erro e falha em relação a severidade e ao tempo

O software necessita de intervenção humana para ser desenvolvido visto que este é produzido por grupos de trabalho formados por *developers*, analistas, *testers*, entre outros, portanto, defeitos, erros e falhas podem ocorrer durante todo o ciclo de vida do software [18]. Testar um software ajuda na descoberta desses possíveis problemas visto que, o teste é feito para garantir que um software executa o seu objetivo corretamente e que este está apto para uso, através do controlo e da preservação da qualidade do software [61].
Assim, este capítulo irá abordar o tema de testes de software, mencionando alguns conceitos essenciais como princípios, tipos, métodos, entre outros.

2.1 Software Development Life Cycle (SDLC)

O *Software Development Life Cycle* (SDLC), Ciclo de Vida de Desenvolvimento de Software em português, é uma metodologia estruturada usada na engenharia de software para orientar o desenvolvimento e a manutenção de sistemas de software. Consiste em várias fases distintas que garantem a qualidade, a confiabilidade e a eficácia dos projetos de software [43].

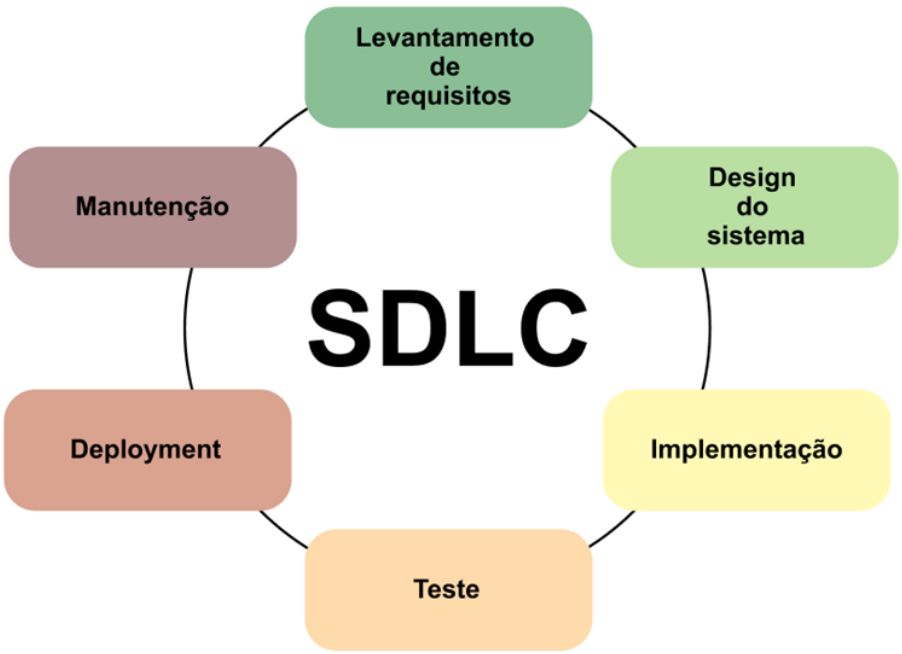

Figura 2.2: Software Development Life Cycle (SDLC)
Baseado em: https://medium.com/@artjoms/software-development-life-cycle-sdlc-6155dbfe3cbc

Deste modo, o SDLC possui as seguintes fases, que podem, ou não, ocorrer em paralelo (Figura 2.2):

1. **Levantamento de requisitos:** Durante esta fase, as partes interessadas do projeto, como clientes, utilizadores e *developers*, colaboram de modo a reunir e documentar os vários requisitos funcionais e não funcionais do software a desenvolver. Esta fase envolve a compreensão das necessidades, dos objetivos e das restrições do projeto por meio de técnicas como entrevistas, *workshops*, questionários e sessões de *brainstorming*. Os requisitos recolhidos são documentados em um documento de requisitos, que serve como base para as fases subsequentes.

2. **Design do sistema:** Nesta fase, os arquitetos de software e *designers* traduzem os requisitos reunidos em um design de sistema detalhado. Estes definem e determinam a arquitetura do software, especificando a estrutura geral, subsistemas e módulos do sistema, bem como interfaces, modelos de dados, algoritmos e base de dados. Fatores como desempenho, escalabilidade, segurança e usabilidade são levados em consideração nesta fase e o resultado da mesma é documentada em um documento de design do sistema que atua como modelo para a equipa de desenvolvimento.

3. **Implementação:** Durante a fase de implementação, os *developers* escrevem o código necessário com base nas especificações fornecidas na fase anterior. Estes utilizam linguagens de programação, *frameworks* e bibliotecas para construir a solução de software. As melhores práticas de codificação, padrões de codificação e documentação adequada são seguidas para garantir a capacidade de manutenção e legibilidade do código. Os sistemas de colaboração e controlo de versão ajudam a gerir alterações de código e facilitam o trabalho em equipa.

4. **Teste:** A fase de teste é crucial para garantir a qualidade e a correção do software. Esta envolve várias técnicas de teste, como testes unitários, testes de integração, testes de sistema e testes de aceitação. Os casos de teste são projetados e o software é validado em relação aos requisitos, de modo a identificar defeitos e erros. O objetivo desta fase é garantir que o software funciona conforme pretendido e atende às necessidades dos utilizadores.

5. **Deployment:** Depois do software passar na fase de teste com sucesso, este é *deployed* no ambiente pretendido. Esta fase inclui atividades como instalação e configuração do software em servidores de produção, plataformas de nuvem ou máquinas do cliente. A instalação e configuração adequadas são essenciais para garantir que o software funciona corretamente no ambiente pretendido.

6. **Manutenção:** A fase de manutenção começa após o *deployment* do software. Durante este período, a equipa de desenvolvimento trata dos defeitos, *bugs* e problemas relacionados e realiza atualizações, correções e os aprimoramentos necessários. As atualizações e os

lançamentos regulares mantêm o software atualizado e alinhado com as expectativas dos utilizadores.

2.1.1 Modelos SDLC

Existem diversos modelos SDLC que fornecem diferentes abordagens para o desenvolvimento de software, sendo que cada modelo é adequado para requisitos e restrições específicos de um dado projeto. É importante selecionar o modelo mais apropriado com base em vários fatores como o tamanho do projeto, a complexidade, os riscos, o envolvimento do cliente, entre outros. Alguns exemplos de modelos SDLC são:

- **Modelo cascata (*Waterfall Model*) [43]:** O modelo cascata é um processo de desenvolvimento de software sequencial, onde cada fase é concluída antes de passar para a fase seguinte. Este segue uma abordagem *top-down*, de cima para baixo (como uma cascata), com fases que incluem recolha de requisitos, design do sistema, implementação, teste, *deployment* e manutenção. Neste modelo a documentação e a compreensão dos requisitos iniciais são enfatizadas, no entanto, existe uma falta de flexibilidade para mudanças tardias no desenvolvimento.

- **Modelo espiral (*Spiral Model*) [7]:** O modelo espiral combina elementos do modelo cascata e elementos do desenvolvimento iterativo, consistindo em várias iterações ou espirais onde cada iteração inclui recolha de requisitos, análise de risco, design de sistema, implementação, teste e avaliação. A gestão de riscos e as melhorias iterativas são aspetos chave do modelo espiral e este adapta-se a projetos com requisitos em evolução e fatores de alto risco.

- **Modelo em V (*V-Model*) [17]:** O modelo em V estende o modelo cascata enfatizando a relação entre as fases de desenvolvimento e as fases de teste correspondentes. Este segue uma abordagem sequencial com fases de desenvolvimento que incluem recolha de requisitos, design do sistema, implementação e integração e fases de teste, que englobam testes unitários, teste de integração, testes de sistema e testes de *user acceptance*. Este modelo garante uma abordagem estruturada para a fase de testes, com o planeamento dos testes a decorrer em paralelo com as fases de desenvolvimento.

- **Modelo Ágil (*Agile Model*) [3]:** O modelo ágil é uma abordagem iterativa e incremental para o desenvolvimento de software, que se concentra na colaboração, na flexibilidade e na entrega de software funcional em iterações curtas chamadas *sprints*. Métodos ágeis como o Scrum e o Kanban envolvem equipas auto-organizadas, envolvimento frequente do cliente, integração contínua e planeamento adaptável. O processo de desenvolvimento é dividido em *user stories* ou recursos, que são priorizados e implementados em iterações. O modelo ágil permite respostas rápidas às mudanças, promove a satisfação do cliente e incentiva a melhoria contínua.

2.2 Software Testing Life Cycle (STLC)

O *Software Testing Life Cycle* (STLC), Ciclo de Vida de Teste de Software em português, é uma abordagem estruturada e metódica utilizada por *testers* de forma a garantir a qualidade e a confiabilidade de um software. Este abrange várias fases distintas, tendo cada uma os seus próprios objetivos, tarefas e resultados específicos.
Dentro do STLC encontram-se as seguintes fases (Figura 2.3):

1. **Análise de requisitos [37]:** Durante esta fase, os *testers* realizam uma análise completa dos requisitos de software de maneira a obter uma compreensão abrangente do escopo do teste e estabelecer objetivos de teste claros. É feita referência ao documento de especificação de requisitos do software, que descreve os requisitos funcionais e não funcionais do software. No final, é produzida uma lista refinada dos requisitos necessários.

2. **Planeamento do teste [8]:** Esta fase envolve a formulação da estratégia geral de teste, a determinação dos objetivos do teste e a definição do ambiente de teste necessário. Os *testers* criam um documento de plano de teste detalhado que descreve o escopo do teste, as entregas do teste, os cronogramas de teste e a alocação de recursos. Esse plano de teste serve como um guia para todo o processo de teste.

3. **Desenvolvimento de casos de teste [19]:** Nesta fase, os *testers* projetam e desenvolvem casos de teste específicos com base nos requisitos identificados e nos objetivos do teste. Os casos de teste consistem em instruções passo a passo que detalham os *inputs* e *outputs* esperados e as condições sob as quais o software deve ser testado.

4. **Configuração do ambiente de teste [20]:** os *testers* estabelecem as configurações de hardware, software e rede necessárias para criar um ambiente de teste indicado. O objetivo é replicar de perto o ambiente de produção, garantindo que os resultados dos testes refletem com precisão as condições do mundo real.

5. **Execução do teste [37]:** Esta fase envolve a execução dos casos de teste previamente desenvolvidos. Os *testers* executam os testes, registam os resultados reais e comparam-nos com os resultados esperados. Quaisquer erros encontrados durante o processo são identificados, registados e relatados. A execução do teste pode ser realizada manualmente ou automaticamente recorrendo a ferramentas especializadas em automatização de testes.

6. **Rastreamento de defeitos [8]:** Os *testers* utilizam ferramentas de rastreamento de defeitos para registar e rastrear sistematicamente os defeitos identificados. Cada defeito recebe um identificador exclusivo e é categorizado com base na sua gravidade e prioridade. Os *testers* comunicam esses defeitos à equipa de desenvolvimento, que os aborda e resolve. O sistema de rastreamento de defeitos auxilia na gestão eficiente do processo de resolução de defeitos.

7. **Relatório de teste [19]:** Os *testers* preparam manual ou geram automaticamente relatórios de teste abrangentes resumindo as atividades de teste, resultados de teste e quaisquer

defeitos descobertos durante o processo de teste. Estes relatórios de teste fornecem aos interessados uma visão geral clara da qualidade do software e permitem que eles tomem decisões informadas com base nas informações obtidas.

8. **Encerramento do teste [20]:** Na fase final, os *testers* avaliam o processo geral de teste e o grau em que os objetivos do teste foram alcançados. Um relatório de encerramento do teste é preparado, abrangendo *insights* valiosos, lições aprendidas, recomendações para melhorias de processo e quaisquer riscos restantes. Este relatório de encerramento de teste serve como uma referência valiosa para futuros projetos de teste.

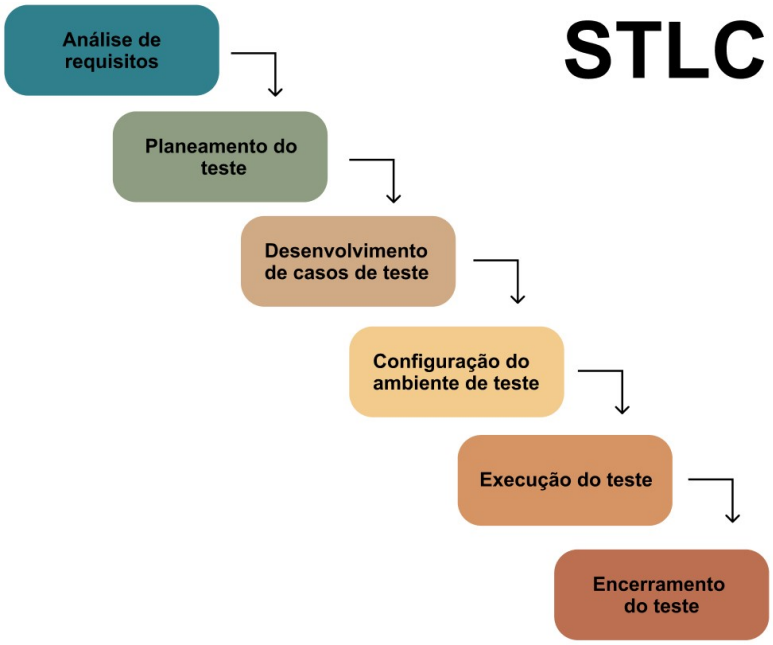

Figura 2.3: Software Testing Life Cycle (STLC)
Baseado em:
https://www.geeksforgeeks.org/software-testing-life-cycle-stlc/

O ciclo de vida de teste de software segue uma abordagem sistemática e estruturada para garantir testes abrangentes e fornecer aplicações de software de alta qualidade. Ao utilizar este ciclo, as organizações podem minimizar os riscos associados a defeitos de software, aumentar a confiabilidade do seu software e, por fim, melhorar a satisfação do cliente.

2.3 Testes de Software

Num mundo ideal, não é possível testar todas as permutações dos parâmetros de um programa mesmo que este seja simples. Criar casos de teste para todas essas possibilidades é impraticável. Testar na totalidade um software complexo é, em regra, um processo impossível e demorado, bem como economicamente inviável e com uma utilização de recursos humanos elevada [38]. No entanto, quanto mais cedo um erro for detetado, menos dispendioso será corrigi-lo. O objetivo

principal de um teste não é demonstrar que o software não possui erros, mas sim dar a confiança que este está a funcionar corretamente [61].
Os testes de software são uma fase integral no ciclo de vida de desenvolvimento de software, envolvendo muitos aspetos técnicos e não técnicos, como especificação, desenho, implementação, instalação, manutenção e problemas de gestão da engenharia de software [61]. É, principalmente, um processo que engloba a validação e verificação do sistema desenvolvido e se este atende aos requisitos definidos pelo utilizador. Portanto, é uma atividade que mede a diferença entre o resultado obtido e o esperado, com o objetivo de encontrar erros ou requisitos ausentes no sistema ou software desenvolvido [20].

2.4 Processo de Testes

Para um teste ter mais probabilidade de atingir os seus objetivos estabelecidos é necessário que exista um conjunto comum de atividades de teste, ou processo de teste [6].
Não existe um processo de testes de software universal, pois cada processo adequado e específico a uma dada situação depende de diversos fatores, e as atividades envolvidas neste processo, bem como a sua implementação e ocorrência, podem ser discutidas na estratégia de teste de cada organização [6].
Os fatores contextuais que influenciam o processo de teste em uma organização incluem [6]:

- Modelo do ciclo de vida de desenvolvimento de software e metodologias de projeto utilizados;
- Níveis e tipos de teste considerados;
- Riscos de produtos e projetos;
- Domínio de negócio;
- Restrições operacionais como orçamentos e recursos, prazos, complexidade e requisitos contratuais e regulamentares;
- Políticas e práticas organizacionais;
- Normas internas e externas estabelecidas.

Adicionalmente, este processo de teste possui, principalmente, o seguinte grupo de etapas [6, 60]:

1. **Planeamento:** Definição e planeamento dos objetivos a cumprir e das abordagens a seguir, tendo em conta o seu contexto;
2. **Monitorização e controlo:** Envolve a comparação entre o progresso atual e o progresso planeado ou esperado, recorrendo a métricas de monitorização definidas;
3. **Análise:** Definição e análise das condições de teste;

4. **Desenho:** Especificação do processo de teste, transformando as condições definidas na fase de análise em casos de teste ou conjuntos de casos teste de alto nível;

5. **Implementação:** Definição da sequência dos casos de teste em um procedimento de teste e definição dos processos necessários para a execução dos testes;

6. **Execução:** Execução do conjunto de testes de acordo com o seu planeamento e comparação dos resultados atuais com os resultados esperados, reportando os defeitos baseados nas falhas observadas e registando o resultado da execução dos testes;

7. **Conclusão:** Armazenamento dos dados e outras informações relevantes sobre as atividades de teste finalizadas.

2.5 Princípios na Realização de Testes

Para testar um software de forma adequada é necessário seguir alguns princípios para ajudar na redução de tempo e esforço despendido. Ao longo dos últimos anos, vários princípios de testes foram sugeridos oferecendo diretrizes gerais comuns, deste modo nos parágrafos seguintes serão descritos os princípios tendo como base o ISTQB (*International Software Testing Qualifications Board*) [6].

1. **Testes mostram a presença de defeitos:** Um teste mostra a presença de defeitos reduzindo a probabilidade de não serem descobertos e de permanecerem no software. No entanto, não prova a sua ausência e mesmo que nenhum defeito seja encontrado, tal não é uma prova de perfeição;

2. **Testes exaustivos são impossíveis:** Testar todas as combinações não é viável na maioria dos casos. Para que o teste não seja exaustivo, a análise de risco, as técnicas de teste e as prioridades em causa devem ser tidas em consideração;

3. **Deve-se testar o quanto antes:** Para encontrar defeitos antecipadamente, as atividades de teste estáticas e dinâmicas devem ser iniciadas o mais cedo possível no ciclo de vida de desenvolvimento de software, de maneira a reduzir ou eliminar mudanças dispendiosas;

4. **Aglomeração de defeitos:** A maioria dos defeitos descobertos durante uma primeira fase de testes estão contidos em módulos e são responsáveis por grande parte das falhas operacionais. A aglomeração de defeitos previstos e de defeitos reais são um *input* importante nas análises de risco;

5. **Paradoxo do pesticida:** Refere-se à noção de que a repetição dos mesmos testes eventualmente perde a eficácia na deteção de novos defeitos, uma vez que eles já identificaram os problemas para os quais foram concebidos. Para continuar a encontrar defeitos, é preciso modificar os dados dos testes existentes ou desenvolver novos testes que examinem áreas do

sistema ainda não exploradas, abordando diferentes cenários, funcionalidades ou mudanças recentes no código;

6. **Testar depende do contexto:** O teste é feito de forma diferente dependendo do contexto. Um exemplo deste tipo de teste, é o cálculo da taxa de envio em umaa aplicação de compras online. Isto é, no ambiente de desenvolvimento, o teste pode usar dados simulados, como uma distância fixa ou um endereço genérico, para verificar a precisão do cálculo. Porém, em produção, o mesmo teste pode falhar se não considerar variáveis contextuais, como promoções regionais, restrições de entrega ou diferentes taxas para regiões específicas, mostrando assim como o contexto pode impactar o resultado do teste.

2.6 Níveis de Testes de Software

Os níveis de teste são grupos de atividades que são organizados e geridos em conjunto. Cada nível é uma fase do processo de teste, realizado em relação ao software em um determinado nível de desenvolvimento, estando relacionados a outras atividades dentro do ciclo de vida de desenvolvimento de software [6].

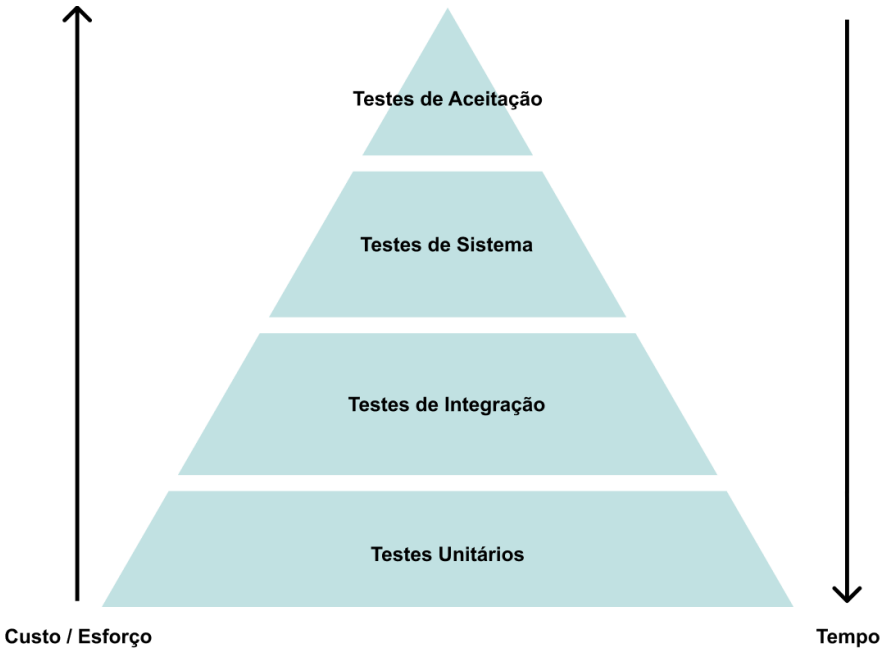

Figura 2.4: Níveis de testes de software, em relação ao tempo e custo/esforço
Baseado em: https://www.exposit.com/blog/what-unit-testing-software-testing-and-why-it-important/

Os níveis de teste podem ser unitários, de integração, de sistema e de aceitação (Figura 2.4).

- **Testes unitários:** Os testes unitários, também denominados de testes de componente, concentram-se em componentes que podem ser testados separadamente [6]. É um tipo de teste que se foca em testar a parte mais pequena do software de forma a verificar a sua funcionalidade em

relação à respetiva especificação [61]. São geralmente feitos isoladamente do resto do sistema, dependendo do modelo de ciclo de vida de desenvolvimento de software e do sistema [6];

- **Testes de integração:** Os testes de integração concentram-se nas interações entre componentes ou sistemas [6]. Estes envolvem o teste de duas ou mais unidades combinadas, que devem funcionar juntas para garantir um controlo de fluxo e de dados sem erros entre as unidades combinadas e um design e integração corretos [61];

- **Testes de sistema:** Os testes do sistema focam-se no comportamento e nos recursos de um sistema em um todo [6], de modo a verificar a sua conformidade com os seus requisitos. Isto é, é verificada a interação geral dos componentes para garantir um funcionamento unânime e sem erros de todos os módulos e programas [61];

- **Testes de aceitação:** Os testes de aceitação, assim como os testes de sistema, geralmente concentram-se no comportamento e nas capacidades do sistema [6], e são realizados para validar o software tendo em consideração os requisitos do cliente, bem como satisfazer requisitos ou padrões legais ou regulamentares [61].

2.7 Tipos de Testes de Software

Os testes podem ser agrupados em grupos de atividades de teste destinados a testar características específicas de um sistema, ou parte de um sistema, com base em objetivos específicos [6].
Existem vários tipos de testes de software, no entanto os com mais relevância para o tema abordado neste projeto são os testes funcionais, os testes não-funcionais e os testes de regressão.

- **Testes funcionais [6, 29]:** Testam o comportamento de um componente ou sistema e têm como objetivo validar esse comportamento em relação à funcionalidade do negócio documentada nos requisitos e nas especificações do software. Ou seja, é fornecido um determinado *input* ao sistema e este é comparado ao *output* gerado, tendo em conta os requisitos funcionais. Complementarmente, o design e a execução dos testes funcionais envolvem o conhecimento detalhado dos problemas de negócio que o software pretende resolver e devem ser realizados em todos os níveis de teste, contudo o foco difere de nível para nível;

- **Testes não funcionais: [6, 22]** Referem-se a quão bem um sistema executa os seus requisitos não funcionais e avaliam as características de um componente ou sistema, como a confiabilidade, o stress, o desempenho, a usabilidade, a eficiência, a segurança, entre outras. Estes podem e devem ser realizados em todos os níveis de teste e feitos o mais cedo possível, visto que a descoberta tardia de defeitos não funcionais pode ser extremamente perigosa para o sucesso de um projeto;

- **Testes de regressão: [6]** Um componente ou sistema irão, na maioria dos casos, sofrer alterações depois de terem sido implementados em ambientes de produção ou mesmo de-

pois de terem sido entregues, seja para corrigir defeitos descobertos, para adicionar novas funcionalidades ou para excluir ou alterar funcionalidades já entregues. Assim, sempre que forem realizadas alterações como parte da manutenção do software, devem ser realizados testes de regressão, tanto para avaliar o sucesso das alterações efetuadas quanto para verificar possíveis efeitos colaterais, como por exemplo regressões, em partes do sistema que permaneceram inalteradas.

2.8 Testes Manuais e Testes Automáticos

Com base na execução do teste, este pode ser classificado em duas categorias: testes manuais e testes automáticos.

Os **testes manuais** (Figura 2.5) são uma técnica de testagem manual do software, ou seja, não recorrem ao uso de qualquer ferramenta automatizada, bibliotecas como o JUnit, ou *scripts* para executar os testes [50]. É um método rigoroso e antigo de teste de software onde se prepara os casos de teste manualmente e os executa para identificar defeitos no software [48]. Neste tipo de teste, o *tester* faz o papel do utilizador final e avalia o software para identificar algum comportamento inesperado ou *bug* [50].

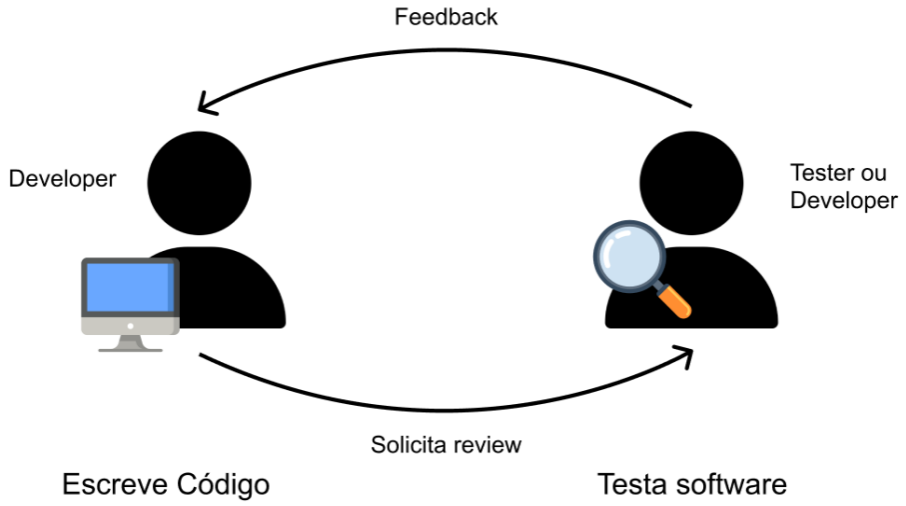

Figura 2.5: Diagrama de testes manuais
Baseado em: https://semaphoreci.com/blog/test-automation

Os testes manuais podem apresentar os seguintes problemas [48]:

- **Difícil repetição:** A repetição de testes manuais pode ser desafiadora em grandes aplicações de software ou em sistemas com um extenso conjunto de dados devido ao tempo, esforço e recursos necessários;
- **Demorado:** Como os casos de teste são executados por recursos humanos, o processo pode tornar-se muito prolongado;

- **Grande investimento em recursos humanos:** Como os casos de teste precisam de ser executados manualmente, é necessário recorrer-se a vários *testers*;
- **Menos credível:** O teste manual é menos credível, pois os testes podem não ser executados com tanta precisão devido a erros humanos;
- **Não programável:** Refere-se à limitação dos testes manuais e ao facto de estes não permitirem a programação de testes complexos ou automatizados. Uma vez que estes testes dependem da execução humana, estes não conseguem lidar eficientemente com grandes volumes de dados, repetições automáticas ou simulações detalhadas, o que limita a abrangência e a precisão dos testes manuais, tornando-os menos eficazes.

Por outro lado, os **testes automáticos** (Figura 2.6) envolvem a escrita de um *script* e o uso de outra ferramenta, sendo uma automatização do processo manual de forma rápida e repetitiva [50]. Essa automatização envolve o desenvolvimento de *scripts* de teste, recorrendo a linguagens de *script* como Python, JavaScript, entre outras, para que os casos de teste possam ser executados por computadores e com o mínimo de intervenção e atenção humana possível. O objetivo da automatização é reduzir o número de casos de teste a serem executados manualmente, sendo possível guardar esse conjunto de testes e reproduzi-los conforme necessário [48].

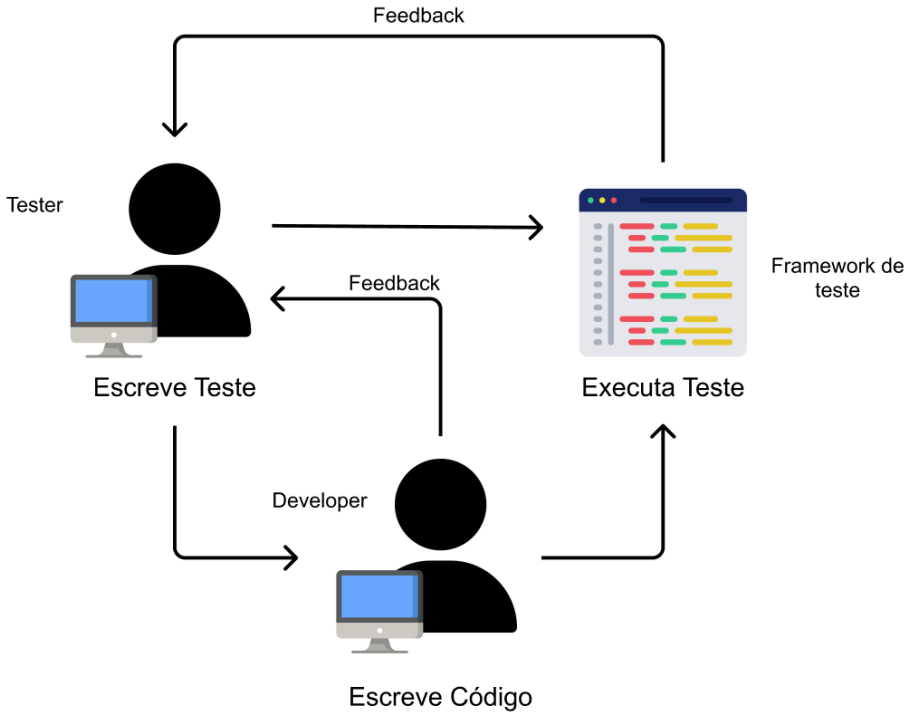

Figura 2.6: Diagrama de testes automáticos
Baseado em: https://semaphoreci.com/blog/test-automation

A automatização exige um investimento monetário e de recursos considerável, no entanto, estes podem ser compensados pelos seguintes benefícios [48]:

- **Rápido:** Mais rápido em termos de execução que os testes manuais;

- **Custo-benefício:** Os casos de teste são executados recorrendo a ferramentas de automatização, portanto, menos *testers* são necessários para a sua automatização;
- **Repetitivo:** O mesmo caso de teste pode ser executado mais que uma vez usando ferramentas de teste;
- **Reutilizável:** Os testes podem ser reutilizados em diferentes versões do software;
- **Programável:** Os *testers* podem programar softwares sofisticados;
- **Abrangente:** Os *testers* podem criar conjuntos de testes que cobrem um número abrangente de recursos da aplicação de software;
- **Mais viável:** Os testes de automáticos executam exatamente a mesma operação sempre que são executados;
- **Cobertura:** Fornece uma cobertura mais ampla nas aplicações.

2.9 Metodologias Ágeis de Desenvolvimento de Software

No início do ano 2001, um grupo de especialistas da área de desenvolvimento de software reuniu-se para fundamentar os valores e os princípios que, quando aplicados corretamente, permitem que as equipas de software trabalhem rapidamente e respondam às mudanças do software [27]. Esta reunião deu origem ao *"Agile Manifesto"*, que apresenta a seguinte ideia [5]:

"Estamos a descobrir melhores formas de desenvolver software produzindo-o e ajudando outros a produzi-lo. Através deste trabalho passamos a valorizar:

- **Indivíduos e interações** em vez de processos e ferramentas;
- **Software que trabalha** em vez de uma documentação completa;
- **Colaboração com cliente** em vez de uma negociação de contratos;
- **Responder à mudança** em vez de seguir um plano.

Ou seja, apesar de haver valor nos itens da direita, valorizamos mais os itens da esquerda."

No desenvolvimento de software, a produção de um software com qualidade e que proporcione uma experiência satisfatória e produtiva ao utilizador é provavelmente o aspeto mais importante. No entanto, este tipo de software é difícil de desenvolver, por isso as equipas responsáveis fazem uso de metodologias de desenvolvimento de software tradicionais ou ágeis que permitem planear e controlar o processo de criação de um software [11].
As metodologias ágeis de desenvolvimento de software estão a crescer em popularidade na indústria de software e são uma inovação no que toca aos testes de software, distanciando-se das

abordagens tradicionais, uma vez que estas abrangem ciclos de teste curtos, rápidos, iterativos e facilmente adaptáveis, permitindo a mudança frequente de requisitos funcionais e não funcionais [20, 60, 11].

Isto é, a integração contínua é a chave para o sucesso do desenvolvimento ágil. Os testes tornam-se uma componente essencial em todas as fases de desenvolvimento, garantindo a qualidade contínua do software. Adicionalmente, a comunicação é de máxima importância e os pedidos dos clientes são recebidos quando é necessário, dando ao cliente a satisfação de que todos os requisitos são considerados e que a qualidade do software está a ser tida em conta ao longo de todo o desenvolvimento [18].

Dois exemplos de metodologias ágeis de desenvolvimento de software são o *Test-Driven Development* (TDD) e o *Behavior-Driven Development* (BDD).

2.9.1 Test-Driven Development (TDD)

O *Test-Driven Development* (TDD), representado na Figura 2.7 e introduzido por Kent Beck [4], é uma metodologia indicada para a escrita de testes de níveis mais baixos (testes unitários) automatizados, com a finalidade de conduzir o design de software e forçar o processo de dissociação das dependências [20, 60].

No processo de teste tradicional, o *tester* identifica defeitos ou erros ao verificar se o sistema se comporta conforme esperado, porém no TDD o sucesso é atingido quando todos os testes escritos deixam de falhar, ou seja, quando cada funcionalidade implementada atende exatamente aos requisitos especificados nos testes, aumentando assim o nível de confiança sobre o sistema [20].

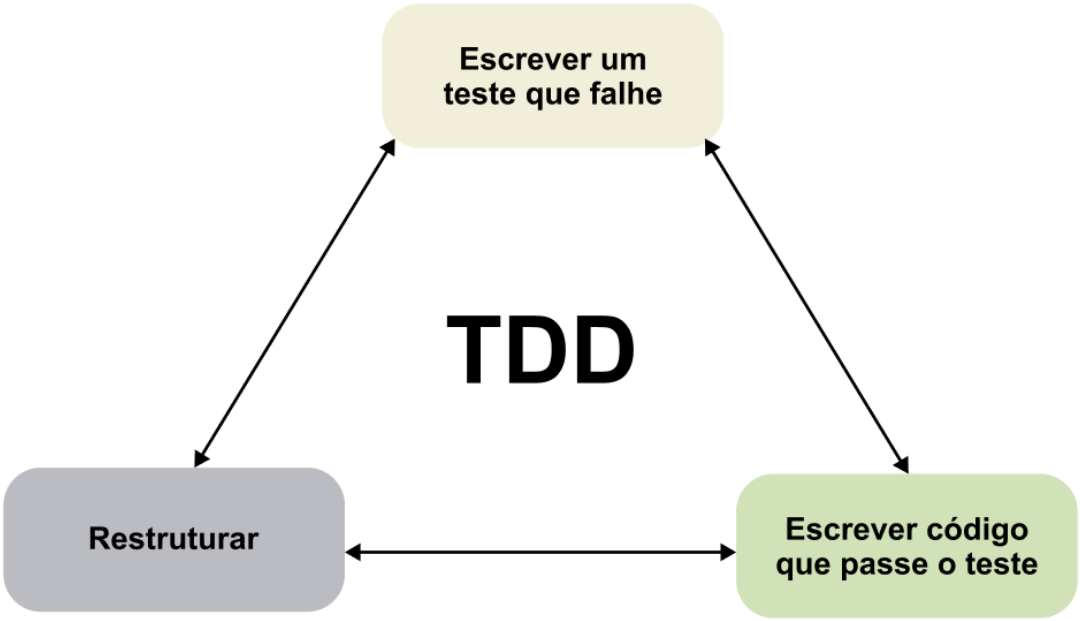

Figura 2.7: Test-Driven Development (TDD)
Baseado em: https://www.impactqa.com/blog/what-is-tdd-test-driven-development-and-its-steps/

Esta metodologia consiste na repetição de ciclos curtos com passos de implementação de testes e de sistema, sendo que a execução desses ciclos possui as seguintes etapas [60, 11]:

1. Selecionar uma funcionalidade ou comportamento que se deseja incluir no sistema;
2. Escrever um teste que cumpra essa funcionalidade ou comportamento e que produza uma falha, ou seja, o teste é projetado para falhar inicialmente uma vez que a funcionalidade ainda não está presente;
3. Executar o teste e observar a razão pela qual ocorreu a falha, garantindo que o teste está a ser executado corretamente, e que é capaz de identificar quando a funcionalidade solicitada não está presente;
4. Escrever a quantidade de código necessária para implementar a funcionalidade e de modo a que o teste desenvolvido nas etapas anteriores passe;
5. Executar todo o conjunto de testes novamente, incluindo o novo teste e os testes preexistentes, de forma a garantir que as alterações feitas não provocam nenhuma falha nas funcionalidades implementadas anteriormente;
    - Se algum teste falhar, deve-se corrigir o código e reexecutar o conjunto de testes;
6. Quando o conjunto de testes estiver sem problemas, o código de produção e os testes são reestruturados, de maneira a melhorar o design do código, remover a duplicação e melhorar a legibilidade;
7. Após a reestruturação, o ciclo continua com o próximo teste e o processo repete-se até que todas as funcionalidades desejadas sejam implementadas.

Um dos benefícios deste método é que, ao se escrever testes antes da implementação da funcionalidade, antecipa-se o entendimento da funcionalidade a ser produzida, evitando escrever código demasiadamente complexo ou que não vá ao encontro aos requisitos do negócio [60]. Outro benefício importante é que este concentra-se na promessa de aumentar a qualidade do produto de software e a produtividade dos *developers* [11].

2.9.2 BDD (Behaviour Driven Development)

O *Behavior-Driven Development* (BDD), ilustrado na Figura 2.8, foi proposto por Dan North [40], e é principalmente uma extensão do TDD com foco nos aspetos comportamentais do sistema, em vez dos aspetos ao nível de implementação [20].
Esta metodologia sintetiza e refina as práticas de engenharia de software que ajudam as equipas a produzir e a entregar software de alta qualidade mais rapidamente, facilitando a comunicação entre a equipa do projeto e as partes interessadas do negócio [11], pois usa uma linguagem específica (Domain Specific Language - DSL [16]) de fácil entendimento para qualquer pessoa ligada ao projeto ou negócio chamada Gherkin [52].

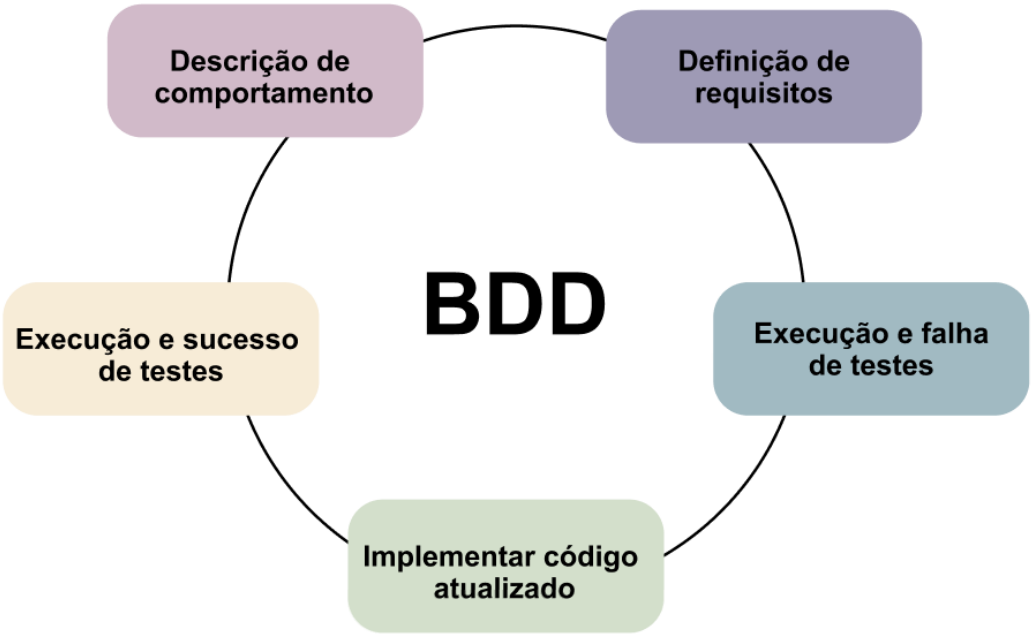

Figura 2.8: Behavior-Driven Development (BDD)
Baseado em: https://www.geeksforgeeks.org/behavioral-driven-development-bdd-in-software-engineering/

O processo do BDD é semelhante ao do TDD e segue as seguintes etapas [40]:

1. Criar um entendimento compartilhado entre todas as partes envolvidas no projeto. Esta etapa envolve esforços colaborativos entre as partes interessadas, os *developers* e os *testers*, de forma a definir os requisitos do sistema recorrendo a uma linguagem acessível a todos os membros da equipa (técnicos e não técnicos).

2. Os requisitos identificados na etapa anterior são guardados em ficheiros *feature*, recorrendo ao uso de *frameworks* que suportam BDD, e servem como especificações executáveis e documentação.

    - Os ficheiros *feature* contêm *scenarios* que descrevem os requisitos ou comportamentos do software na perspetiva do utilizador. Estes *scenarios* são escritos na sintaxe Gherkin e possuem palavras-chave, como *Given*, *When* e *Then*, que representam os comportamentos do software de acordo como o estado inicial (*Given*), as ações tomadas (*When*) e os resultados esperados (*Then*).

3. Para automatizar os *scenarios*, os *developers* escrevem o código necessário recorrendo a *frameworks* de teste. Estes testes automáticos vão servir como testes executáveis que verificam se o sistema se comporta conforme especificado.

4. Os testes são executados regularmente para verificar o comportamento e funcionalidade do mesmo, e são geralmente integrados em uma pipeline de desenvolvimento para detetar regressões e garantir a conformidade com o comportamento especificado.

5. Ao longo do processo de desenvolvimento, os *scenarios* e os ficheiros *feature* são continuamente revistos, reestruturados e atualizados com base no *feedback* ou na mudança de requisitos.

2.10 Frameworks que suportam BDD

Segundo o autor Dan North [40], o BDD consiste em formalizar a linguagem em torno da nomenclatura e construção do teste, ou seja, permite um entendimento partilhado de como um software se deve comportar, descobrindo os requisitos ausentes com base em exemplos concretos [57]. Tal deve-se ao facto de que, nomear os testes de acordo com o comportamento esperado facilita o entendimento por toda a equipa de desenvolvimento e promove um alinhamento mais eficiente. Adicionalmente, quando os testes são formalizados estes podem ser lidos por outras partes interessadas no projeto, como analistas de negócio ou clientes, fortalecendo a comunicação entre todos os envolvidos [42].
Existem várias *frameworks* que suportam o BDD, como o Cucumber, o Specflow, o JBehave, o RSpec e o Quantum. De seguida, será apresentada uma breve definição de cada uma delas.

- **Cucumber [53]:** É uma *framework open-source* que valida especificações executáveis e descreve como o sistema se deve comportar, de forma a que qualquer membro da equipa consiga entender. Esta oferece suporte a várias linguagens de programação, incluindo Ruby, .NET, Java, Javascript, entre outras;

- **Specflow [56]:** É uma *framework open-source* que utiliza métodos .NET e C#, e permite a escrita de ficheiros *feature* e a automatização de código. Esta recorre ao uso da linguagem Gherkin para descrever *user stories* e possui suporte para estruturas de teste como MSTest, NUnit, xUnit e MbUnit;

- **JBehave [21]:** É uma *framework* para linguagem Java, desenvolvida por Dan North, que é constituída por dois componentes nomeados Jbehave Web e Jbehave Main. Esta fornece *plugins* de integração para o Eclipse, o Netbeans e o IntelliJ IDEA;

- **RSpec [44]:** É uma *framework* para a linguagem Ruby, inspirada no JBehave, que é composta por múltiplas bibliotecas projetadas para trabalhar em conjunto ou independentemente com outras ferramentas como o Cucumber;

- **Quantum [51]:** É uma *framework open-source* baseada em Java, desenvolvida pela Perfecto, que facilita e acelera a automatização de testes completos com suporte ao BDD.

Capítulo 3

Análise

Como já referido anteriormente, no âmbito do trabalho de projeto de mestrado, surgiu a oportunidade de trabalhar na empresa CGI, no projeto que envolve o programa de transformação do setor de eletricidade do Reino Unido. Mais especificamente, o projeto a desenvolver resume-se à realização de testes funcionais automáticos referentes às diversas funcionalidades implementadas pela equipa de *developers* do projeto.
Tal foi proposto uma vez que um processo automatizado de teste irá facilitar o desenvolvimento do projeto, que possui um negócio particularmente sofisticado e árduo, pois contem uma grande quantidade de regras de negócio. Para tal, vai-se recorrer ao uso da *framework* Specflow, em conjunto com o paradigma BDD.
Adicionalmente, estas ferramentas foram pré-selecionadas pela CGI para a realização do projeto uma vez que, a empresa já está familiarizada com as mesmas devido ao seu uso pelos técnicos e engenheiros da entidade em outros projetos.
De seguida, será abordado em mais detalhe a ferramenta de desenvolvimento escolhida para realizar o projeto, o Azure DevOps, bem como a estrutura e o funcionamento do Specflow, mencionando também a metodologia BDD e a linguagem Gherkin para melhor entendimento da *framework*.

3.1 Ferramentas e linguagens de desenvolvimento

3.1.1 Microsoft Azure Devops

O DevOps (Figura 3.1) é uma cultura diferente das culturas corporativas tradicionais e requer uma mudança de mentalidade, processos e ferramentas. Este está frequentemente associado a práticas de Integração Contínua (CI) e Entrega Contínua (CD), que são práticas de engenharia de software que fazem uso da *Infrastructure as Code* (IaC) e que envolvem a codificação da estrutura e configuração da infraestrutura [25].
O termo DevOps foi introduzido, em 2007-2009, por Patrick Depois, Gene Kim e John Willis, e representa a combinação das palavras inglesas *Development* (Dev) e *Operations* (Ops). Este termo deu origem a um movimento que defende a união dos *developers* e das operações em equipas [25].
A cultura DevOps foi integrada pois percebeu-se que os *developers* e operadores de IT trabalha-

vam como unidades separadas em vez de trabalharem em conjunto e em equipa para um objetivo comum [45]. Esta consiste num conjunto de práticas que reduzem as barreiras entre os *developers*, que querem inovar e entregar o mais rápido possível, e operadores de IT, que querem garantir a estabilidade dos sistemas de produção e a qualidade do sistema de mudanças produzido [25].
Adicionalmente, o DevOps esforça-se para resolver problemas comuns entre os *developers* e as operações [45], portanto é muitas vezes visto como uma extensão dos processos ágeis que possibilitam a redução dos prazos de entrega e envolvem os *developers* e *timings* de negócios, mas muitas vezes são prejudicados por não incluírem a parte das operações [25].
A comunicação e a ligação entre Dev e Ops permite um melhor acompanhamento de implementação de produção *end-to-end* e implementações mais frequentes e de melhor qualidade, economizando dinheiro para a empresa [25].

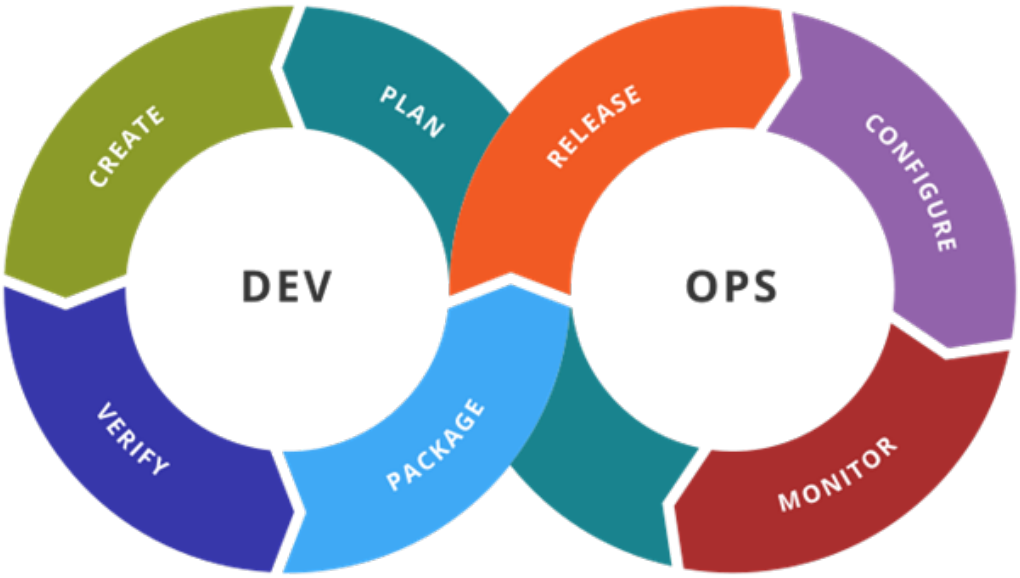

Figura 3.1: Ciclo do DevOps
Obtido em:
https://imanjunad.medium.com/introduction-to-dev-ops-4cb794d2c0fc

Para facilitar esta colaboração e melhorar a comunicação entre Dev e Ops, existem vários elementos-chave nos processos a serem implementados, como por exemplo [25]:

- Implementação de aplicações mais frequentes com integração e entrega contínua (CI/CD);
- Implementação e automatização de testes unitários e de integração, com um processo focado em BDD ou TDD;
- Implementação de um método de recolha de *feedback* dos utilizadores;
- Aplicações de monitorização e infraestrutura.

Complementarmente, alguns dos benefícios de estabelecer uma cultura DevOps dentro de uma empresa são os seguintes [25]:

- Melhor colaboração e comunicação nas equipas, fornecendo um impacto humano e social na empresa;
- Tempos de entrega mais curtos, resultando em um melhor desempenho e satisfação do utilizador final;
- Custos de infraestrutura reduzidos;
- Redução significativa de tempo através de ciclos iterativos que reduzem erros de aplicação, e ferramentas de automatização que reduzem tarefas manuais para que as equipas se concentrem mais no desenvolvimento de novas funcionalidades.

Assim, e uma vez que o DevOps resolveu vários problemas da engenharia de software, incluindo o atrito e o atraso na entrega de software e resolução de problemas [45], a ferramenta de desenvolvimento escolhida, para o projeto a desenvolver neste trabalho de projeto, foi o Microsoft Azure Devops.
O Microsoft Azure DevOps abrange um conjunto de diversas ferramentas e serviços fornecidos pela Microsoft, que coletivamente facilitam todo o espectro de atividades envolvidas no desenvolvimento de software. Este conjunto de ferramentas abrange desde o controlo de versão e integração contínua até entrega contínua, teste, gestão de projetos e muito mais.
O seu objetivo principal é promover a colaboração eficiente, simplificar o fluxo de trabalho de desenvolvimento e agilizar a entrega de soluções de software de alto nível [33].

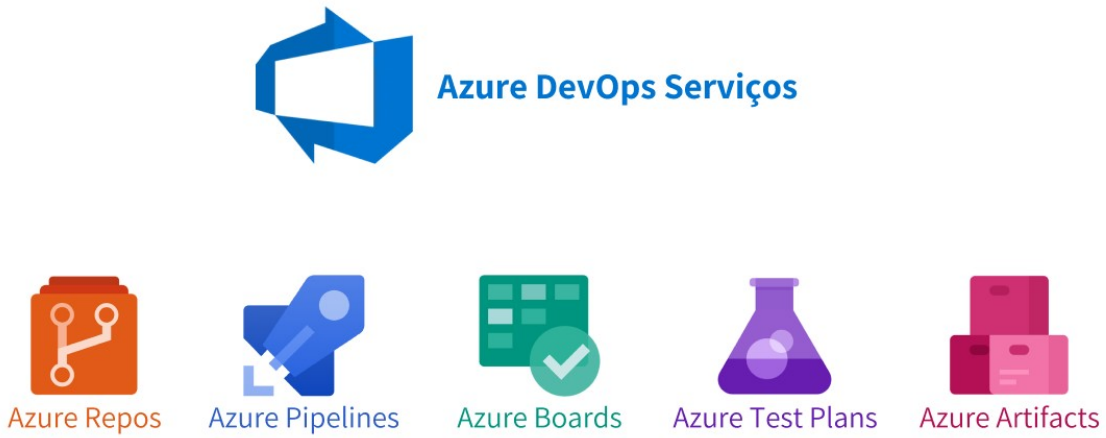

Figura 3.2: Serviços do Microsoft Azure DevOps

O Microsoft Azure DevOps é a abordagem da Microsoft para as soluções DevOps, e inclui vários serviços (Figura 3.2) como:

- **Azure Repos [31]:** Conjunto de ferramentas de controlo de versão que podem ser usadas para gestão de código. Oferece compatibilidade com o Git, um sistema de controlo de

versão distribuído, e com o *Team Foundation Version Control* (TFVC). Entre as suas funcionalidades multifacetadas encontram-se o rastreamento meticuloso de alterações, revisão de código colaborativo e gestão de *branches* divergentes e *pull requests*.

- **Azure Boards[32]:** Serviço baseado na Web que permite às equipas planear, acompanhar e discutir o trabalho durante todo o processo de desenvolvimento, enquanto oferece suporte a metodologias ágeis, como o Scrum e o Kanban. Este fornece uma plataforma personalizável para a gestão de itens de trabalho, permitindo que as equipas colaborem de forma eficaz e simplifiquem o seu fluxo de trabalho.

- **Azure Pipelines [34]:** Este serviço cria e testa automaticamente projetos de código. Oferece suporte a todas as principais linguagens e tipos de projetos, e combina integração contínua, entrega contínua e testes contínuos para construir, testar e entregar código. Adicionalmente, este serviço pode ser subdividido em duas dimensões principais: **Build Pipelines** e **Release Pipelines**.

- **Azure Test Plans [35]:** Fornece ferramentas avançadas e poderosas que podem ser usadas por toda a equipa de maneira a impulsionar a qualidade e a colaboração em todo o processo de desenvolvimento. Este proporciona todos os recursos necessários para diversos tipos de testes e recolha de *feedback* para todas as partes interessadas.

- **Azure Artifacts [30]:** Permite que os *developers* partilhem o seu código de forma eficiente, bem como a gestão de todos os seus *packages* em um só sítio. Com este serviço, os *developers* podem consumir *packages* de diferentes *feeds* e registos públicos ou publicar os seus próprios *packages* e compartilhá-los dentro de uma equipa, entre organizações ou até mesmo para o publico em geral. Adicionalmente, o serviço dá suporte a vários tipos de *packages*, como o NuGet, o npm e o Maven.

Estes serviços são extensíveis e flexíveis e podem ser usados com diferentes plataformas e *clouds*, de modo a que o utilizador possa escolher que solução pretende usar nestes serviços.
O Azure DevOps também oferece uma variedade de extensões e suporte para extensões criadas pelo utilizador, existindo extensões para ferramentas como o Docker, o Slack, o GitHub, o Sonar Qube, o AWS, entre outras. [45]

3.1.2 BDD (Behaviour Driven Development)

Como já referido anteriormente, o BDD é um tipo de metodologia ágil de desenvolvimento de software e foi projetado como uma extensão e melhoria ao TDD. Este foi desenvolvido como uma forma de tornar as melhores práticas ágeis mais acessíveis e eficazes, concentrando os testes no comportamento necessário [42].
Esta metodologia ágil foi escolhida pela CGI para ser integrada no projeto, pois esta amplia a contribuição no desenvolvimento da solução, incluindo a parte comercial e o utilizador final que podem ter pouco conhecimento na área de desenvolvimento de software. Devido a esse ciclo de

feedback mais alargado, o BDD pode ser usado facilmente em ambientes de integração e entrega contínua [18].
Adicionalmente, ao ser implementado no projeto de automatização de testes, o BDD fornece algumas vantagens [18, 57]:

- **Colaboração:** Permite colaboração contínua e maior visibilidade entre toda a equipa;
- **Satisfação do utilizador:** Ao focar nas necessidades do negócio, os utilizadores ficam satisfeitos, o que se traduz em fidelidade do cliente e melhores resultados de negócios;
- **Menos erros:** Ao envolver os *testers* desde o início proporciona que menos erros e defeitos sejam produzidos em cada iteração de implementação;
- **Eficácia:** Como se estende o sistema em pequenos incrementos, pode-se validar os valores de negócios mais cedo e evitar recursos desnecessários;
- ***Feedback* rápido:** As mudanças são mais fáceis e seguras, pois há um *feedback* constante que permite acompanhar o andamento do projeto;
- **Qualidade do código:** O BDD tem um impacto positivo na qualidade do código, pois promove o design emergente que garante uma arquitetura altamente coesa e evita engenharia em excesso.

No entanto, e apesar da integração do BDD apresentar benefícios para o projeto, este também traz alguns desafios [57]:

- **Curva de aprendizagem:** Existe sempre uma curva de aprendizagem que exige que toda a equipa dedique tempo e esforço de aprendizagem sempre que uma nova metodologia ou prática é introduzida. Deste modo, adotar uma nova prática pode ser um grande desafio para algumas equipas;
- **Sobrecarga de tempo inicial:** Criar e manter os ficheiros *feature* e *scenarios* requer uma sobrecarga e investimento inicial de tempo e esforço.

Por fim, o projeto de automatização de testes de software, a realizar com o auxílio do BDD, irá possui três fases de desenvolvimento [57]:

- **Descoberta:** Refere-se à prática estruturada e colaborativa que especifica os requisitos do software com base em exemplos. Esta tem como objetivo superar os *blind spots* desconhecidos no desenvolvimento de software e reduzir a aprendizagem *bottleneck*;
- **Formulação:** Constante na prática de transformar exemplos-chave da fase de descoberta em documentação formalizada, como os *scenarios* Gherkin de fácil leitura;
- **Automatizar:** Nesta fase os *scenarios* formalizados são transformados em testes de aceitação automáticos. Os testes de aceitação automáticos reduzem significativamente o esforço

dos testes de regressão e fornecem *feedback* imediato quando a documentação da especificação não corresponde ao comportamento do sistema.

Em relação à fase de descoberta, esta foi realizada previamente pela equipa, em colaboração com o cliente, onde foram assinalados os requisitos que o software necessita de conter para que o mesmo funcione como é suposto. Apesar desta fase não ter sido elaborada no âmbito deste projeto, a equipa encontra-se sempre disponível para dar o contexto necessário para que as próximas fases sejam realizadas como esperado.
Por outro lado, as fases de formulação e automatização serão efetuadas com uso da linguagem Gherkin e da *framework* SpecFlow, que serão abordados de seguida.

3.1.3 Linguagem Gherkin

Dentro do BDD serão definidas as *user stories* do projeto, que irão incluir os critérios de aceitação e das quais irão derivar os testes de aceitação. As *user stories* vão ser integradas nos ficheiros *feature* do projeto e vão conter os *scenarios* dos casos de testes, que por sua vez serão escritos em uma linguagem de domínio específica chamada Gherkin.
Esta linguagem é muito semelhante à linguagem natural e foi projetada para descrever casos de uso em software de forma não técnica e legível, portanto todos os membros da equipa envolvida em um projeto podem ler e entender a sua escrita [18, 58].
O Gherkin foi escolhido, pois quando usado para escrever *user stories* torna-se uma vantagem importante, visto que o BDD não se propõe apenas a automatizar os testes, mas também a melhorar a comunicação entre especialistas do domínio e *developers* de software [18].
O código apresentado abaixo (Figura 3.3), mostra um exemplo básico de um ficheiro *feature* que recorre ao uso da linguagem Gherkin.

```
1   Feature: Calculator
2       In order to avoid silly mistakes
3       As a math idiot
4       I want to be told the sum of two numbers
5
6   Scenario: Add two numbers
7       Given I have entered 50 into the calculator
8       And I have entered 70 into the calculator
9       When I press add
10      Then the result should be 120 on the screen
```

Figura 3.3: Exemplo de código em linguagem Gherkin

No código apresentado na Figura 3.3 é possível identificar algumas das palavras-chave que correspondem à sintaxe Gherkin. Estas também podem ser chamadas de *steps*, e cada uma delas é

analisada e executada por uma framework de teste BDD para verificar a expectativa do software [9].
De seguida, será apresentada uma descrição de cada um desses *steps* [18, 58].

- **Feature:** Todos os ficheiros *feature* começam com a palavra "Feature". Esta representa o nome da funcionalidade que se pretende testar e deve ser única, clara e explícita, fornecendo uma descrição de alto nível de uma funcionalidade do software, de maneira a agrupar *scenarios* relacionados;
- **Scenario:** Descreve os contextos específicos e os resultados de uma *user story*. Este inclui as etapas necessárias para colocar o sistema no estado que se deseja, sendo possível ter vários *scenarios* em um ficheiro *feature*;
- **Given:** Usada para descrever o contexto inicial do sistema, ou seja, as pré-condições para o *scenario*;
- **And:** Usada para estender os *steps* e ajuda a elaborar ainda mais o ficheiro *feature*;
- **When:** Especifica as ações ou eventos que serão executados quando o teste for iniciado;
- **Then:** Define o resultado dos *steps* anteriores e detalha as validações a serem feitas;
- **Tags:** Podem ser colocadas encima da palavra "Feature", para agrupar features relacionadas, ou da palavra "Scenario", independentemente do ficheiro e da estrutura do diretório;
- **Parâmetros delimitados:** Parâmetros delimitados "< >" são usados como referências que são referidas nas tabelas de exemplo, sendo que o SpecFlow substitui automaticamente esses parâmetros ao executar testes;
- **Tabelas de dados:** São úteis para fornecer uma lista de valores para um *step definition*.

3.1.4 Framework SpecFlow

O projeto de automatização de testes de software, a ser elaborado no âmbito deste projeto, irá utilizar a *framework* SpeckFlow que estará integrada no IDE Visual Studio e esta servirá como base para a estrutura do projeto.
O SpecFlow (Figura 3.4) é uma *framework open-source* de automatização de testes para .NET, que integra a metodologia BDD e que pode ser encontrada para consulta no GitHub. Esta é construída sobre o paradigma BDD e pode ser usada para definir, gerir e executar automaticamente testes de aceitação legíveis a todos os envolvidos em projetos .NET. Ademais, o IDE Visual Studio inclui uma extensão relativa ao SpecFlow que adiciona vários recursos úteis a este, no entanto este não é exclusivo do Visual Studio, podendo também ser utilizado com o IDE Rider e o VSCode [59].
Dentro dos benefícios que o Specflow irá fornecer ao projeto, encontram-se [56]:

- **Ajuda a equipa a definir os requisitos:** Ao usar o BDD para chegar aos critérios de aceitação, é possível encontrar rapidamente os requisitos necessários e garantir que todos os interpretem da mesma maneira;
- **Linguagem comum que é compreendida por todos:** O uso do Gherkin ajuda a estabelecer um entendimento partilhado por toda a equipa;
- **Facilita a manutenção e extensão dos testes automáticos existentes:** Permite que os casos de testes sejam separados da automatização, reduzindo o esforço necessário para manter os testes;
- **Relatórios facilitados:** Junta a documentação de um caso de teste com os resultados da automatização do teste em um só sítio.

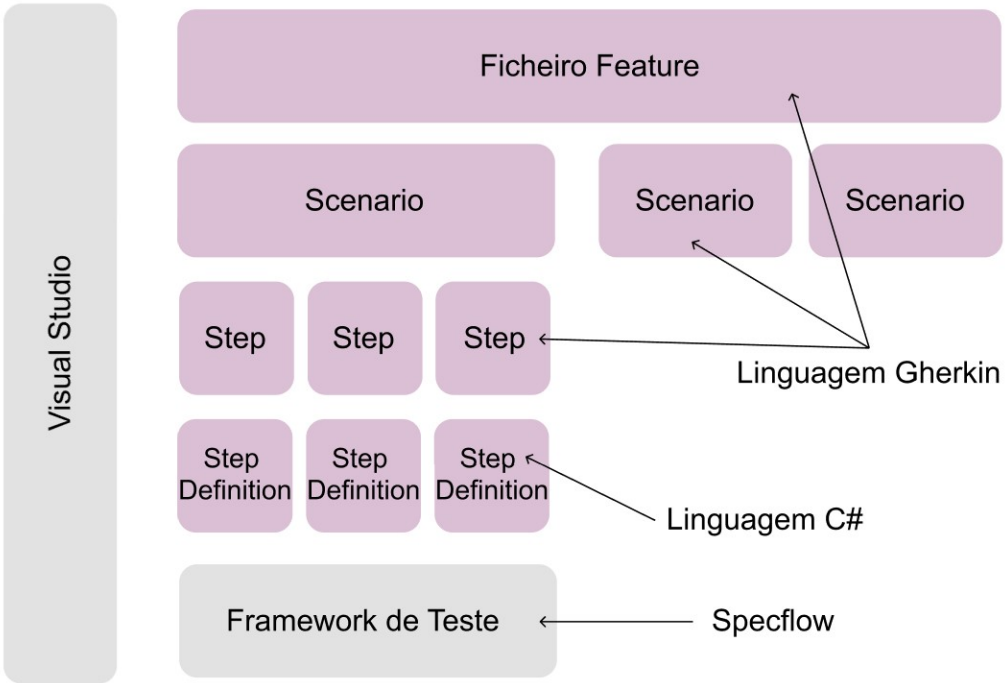

Figura 3.4: Estrutura do Specflow
Baseado em:
https://www.slideshare.net/slideshow/using-speckflow-for-bdd/39742396

Como já referido, os testes em SpecFlow a desenvolver no projeto vão ser escritos recorrendo ao Gherkin, uma linguagem que suporta mais de 70 idiomas e permite escrever casos de teste usando linguagens naturais. Posteriormente, esses testes serão anexados ao código do software utilizando *bindings* [59].

Uma vez que os ficheiros *feature* do Gherkin não podem ser executados sozinhos, a automatização que associa a especificação do Gherkin à interface da aplicação tem de ser desenvolvida primeiro. Essa automatização é chamada de *binding*. As classes e métodos de *binding* podem ser definidos no projeto SpecFlow ou em *binding assemblies* externos [59].

Existem vários tipos de *bindings* no SpecFlow, sendo o mais usado no projeto o *step definition*. O *step definition* automatiza o *scenario* ao nível do *step*, isso significa que, em vez de fornecer automatização para todo o *scenario*, a automatização é feita para cada um dos *steps* separadamente. O benefício desse modelo é que as definições dos *steps* podem ser reutilizadas em diferentes *scenarios*, tornando possível construir (parcialmente) vários *scenarios* a partir de *steps* existentes com menos (ou nenhum) esforço de automatização [59].

É de notar que, os *steps definition* a serem utilizados neste projeto vão ser desenvolvidos pela equipa de *devolpers* do projeto, recorrendo à linguagem de programação C#. Assim, aprofundar em mais detalhe este tópico vai para além do âmbito do trabalho a realizar, visto que este se foca mais no processo de teste e subsequentemente no desenvolvimento dos ficheiros *feature*.

Capítulo 4

Estratégia de testes

Este capítulo irá fornecer uma visão detalhada da estratégia de testes utilizada no projeto. Neste será examinada a arquitetura de software, bem como definido o âmbito da estratégia, cobrindo a metodologia de desenvolvimento, os papéis da equipa do projeto e o processo de testes. Esta secção irá explicar como estes elementos colaboram para garantir testes abrangentes e eficazes.
Será também discutida a abordagem de testes, delineando as estratégias e métodos de alto nível personalizados que serão utilizados. Por fim, o capítulo conclui com uma análise aprofundada da pipeline CI/CD, explicando como a integração contínua e a entrega contínua são incorporadas na estratégia de testes e enfatizando o uso do Azure Pipeline.
Resumidamente, este capítulo irá apresentar uma visão coesa da estratégia de testes, demonstrando como cada componente trabalha em conjunto para garantir a produção de um produto de software de alta qualidade.

4.1 Arquitetura de software

Compreender a arquitetura de software é essencial, pois esta fornece um modelo para o desenvolvimento do mesmo, bem como para o seu possível crescimento no futuro. Uma arquitetura bem construída facilita a tomada de decisões informadas e a gestão de riscos eficaz, garantindo a adaptabilidade do sistema aos requisitos em constante evolução e às mudanças tecnológicas.
Assim sendo, a Figura 4.1 ilustra a arquitetura de software do projeto de automatização de testes adotada pela estratégia de testes proposta.

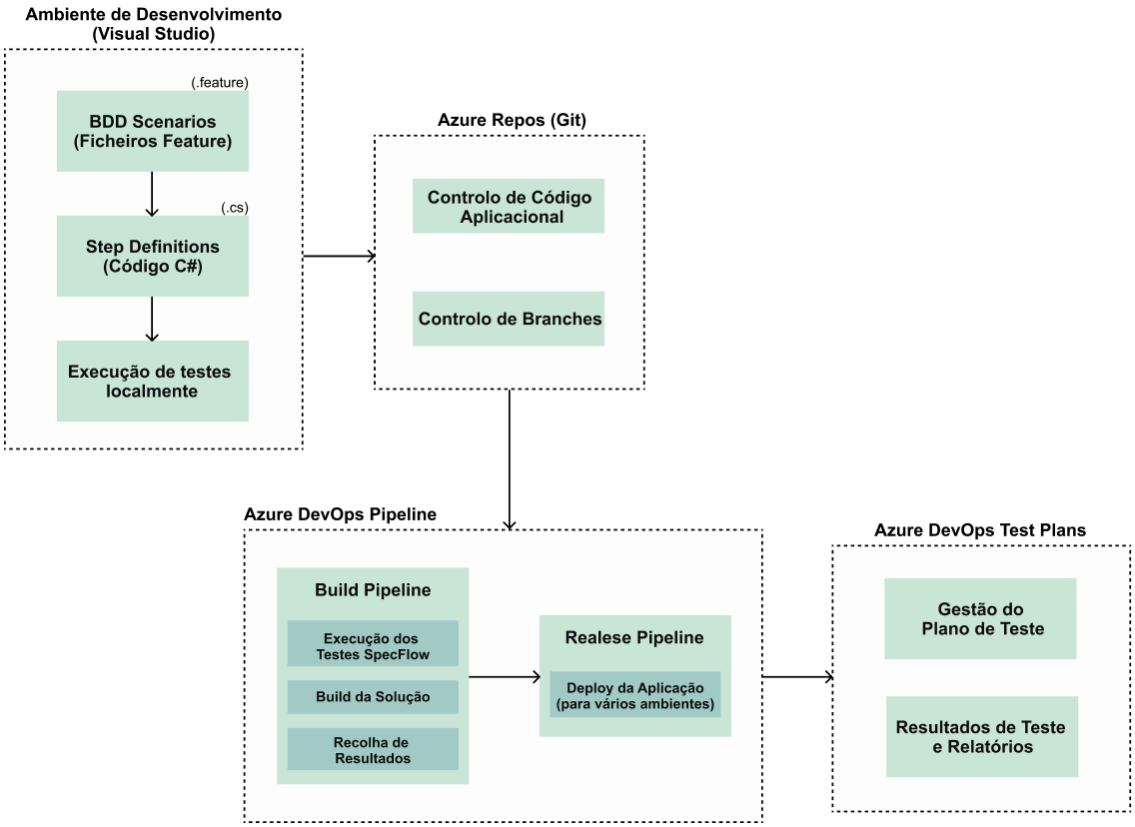

Figura 4.1: Arquitetura de Software

A arquitetura de software escolhida utiliza o Visual Studio em combinação com o SpecFlow para estabelecer um ambiente de desenvolvimento robusto com base nos princípios do *Behavior-Driven Development* (BDD), princípios estes que foram mencionados na Subseção 2.9.2 e na Subseção 3.1.2. Esta integração agiliza a execução dos testes, permitindo que os *testers* executem testes diretamente do ambiente de desenvolvimento integrado (IDE).
Dentro da *framework* SpecFlow, os ficheiros *feature*, contendo os diversos *scenarios*, vão ser meticulosamente criados recorrendo ao uso da sintaxe Gherkin e estes vão encapsular efetivamente as diversas *user stories* e critérios de aceitação, tornando-os compreensíveis para as partes interessadas técnicas e não técnicas.
Para converter os *scenarios* de alto nível descritos nos ficheiros *feature* em testes executáveis, vão ser desenvolvidos *step definitions* em linguagem de programação C#. Estes *step definitions* servem como ponte entre as etapas abstratas do Gherkin e os *scripts* de teste executáveis, mapeando cada *step* do Gherkin para um código de teste específico.

No que toca a abordagem de controlo de código, a mesma será centralizada no Azure Repos, onde é armazenado todo o código de teste do SpecFlow e *step definitions*. Este repositório centralizado garante que todos os membros da equipa podem aceder à versão mais atual do código, promovendo um ambiente de desenvolvimento colaborativo e consistente. Ainda, ao manter o código no Azure Repos, também é preservado um histórico detalhado de alterações, o que é essencial para

rastrear o progresso e solucionar problemas.
O sistema de *branching* do Azure Repos oferece suporte à gestão de diferentes recursos e correções de forma independente. Cada implementação ou correção nova é desenvolvida na sua própria *branch* e depois é efetuado um *merge* para a *branch* principal após revisão e testes rigorosos.

Por outro lado, a pipeline de integração contínua e entrega contínua (CI/CD) vai ser gerida através do Azure DevOps Pipelines, que automatiza os processos de *build*, teste e *deploy*. Esta automatização garante que as alterações no código base são continuamente integradas e entregues, mantendo a alta qualidade do código e *deploy* rápido.
Durante a fase de CI, a *build pipeline* é acionada automaticamente sempre que alterações de código são confirmadas. Esta pipeline compila a solução e executa os testes SpecFlow para verificar se o código permanece de um estado de *deploy*. A deteção precoce de problemas por meio destes testes automáticos garante que os problemas são resolvidos prontamente.
Se a *build* e os testes forem bem-sucedidos, a fase de CD começa, onde a *release pipeline* faz *deploy* automático da aplicação em vários ambientes.

Por fim, a gestão do plano de testes é realizada por meio do Azure DevOps Test Plans, que oferece ferramentas abrangentes para a gestão e revisão de resultados dos testes automáticos. Este serviço fornece uma visualização clara dos resultados dos testes, permitindo rápida identificação e resolução de problemas, e ainda facilita a criação e a gestão de *test plans*, *test suites* e *test cases* de forma organizada.
Adicionalmente, os resultados dos testes automáticos ao serem integrados no Azure DevOps Test Plans, permite uma revisão e análise detalhadas dos mesmos, e fornece informações valiosas sobre a qualidade do projeto.

4.2 Âmbito da estratégia

4.2.1 Metodologia de desenvolvimento

A metodologia de desenvolvimento a ser usada no projeto é o Scrum. O Scrum é uma *framework* ágil projetada para aprimorar a colaboração, a transparência e a adaptabilidade na gestão e no desenvolvimento de projetos complexos, especialmente na área de desenvolvimento de software. A *framework* é composta por **papeis**, **eventos**, **artefactos** e **regras** distintos que, coletivamente, formam uma abordagem estruturada e iterativa para a gestão de projetos [36].
Dentro da *framework* Scrum, os papeis principais incluem [36]:

- O **Product Owner**, que representa as partes interessadas e tem a tarefa de maximizar o valor do produto, bem como definir e priorizar o conteúdo do product backlog;
- O **Scrum Master**, outro papel crítico, que atua como facilitador/moderador do processo Scrum, garantindo a adesão aos princípios do Scrum e promovendo um ambiente de equipa colaborativo e auto-organizado;

- A **equipa de desenvolvimento**, composta por indivíduos multi-funcionais, que são responsáveis por entregar o produto de forma incremental e que é fundamentalmente auto-organizada.

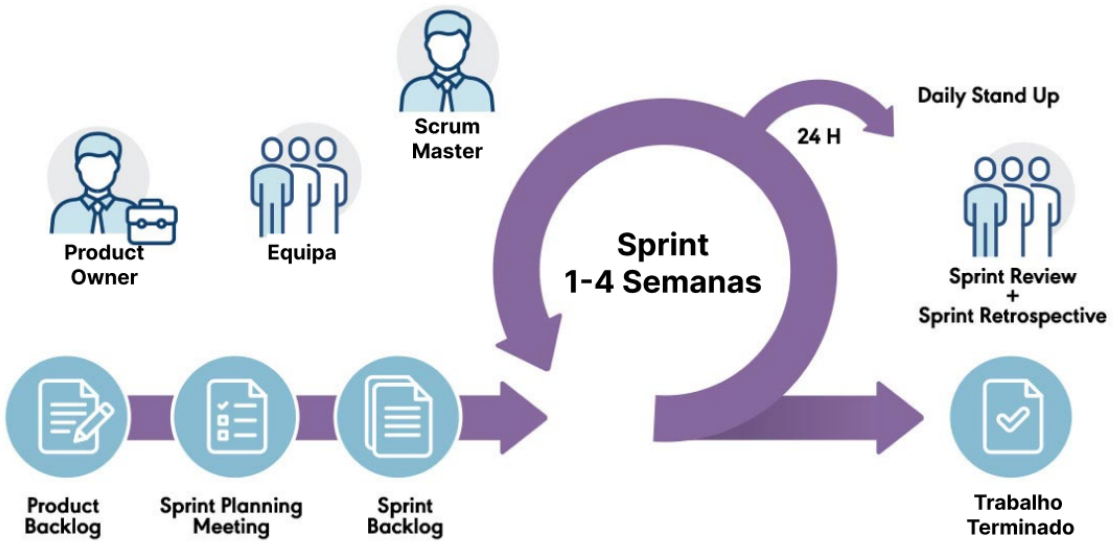

Figura 4.2: Processo Scrum
Obtido em: https://www.pm-partners.com.au/insights/the-agile-journey-a-scrum-overview

O Scrum (Figura 4.2) opera em iterações com limite de tempo (*time-boxing*, em inglês) conhecidas como **Sprints**, que normalmente duram de 2 a 4 semanas. Estas Sprints fornecem uma cadência consistente e previsível para o projeto. O processo começa com o **Sprint Planning**, onde o Product Owner apresenta um product backlog priorizado, e a equipa de desenvolvimento seleciona um conjunto de tarefas para trabalhar durante a próxima Sprint, definindo um **Sprint Goal** de forma colaborativa [36].

Durante a Sprint, a equipa participa em **Daily Stand Ups**, que consistem em breves reuniões diárias onde os membros atualizam a equipa sobre o trabalho realizado, abordando três perguntas-chave: "O que eu fiz ontem? O que vou fazer hoje? Existem problemas?". No final de cada Sprint, a equipa realiza uma **Sprint Review** onde é apresentado o trabalho concluído ao Product Owner e às partes interessadas, e uma **Sprint Retrospective**, onde se reflete sobre a Sprint e é discutido o que correu bem e o que correu mal durante a mesma [36].

Os principais artefactos do Scrum incluem o **Product Backlog**, uma lista dinâmica e priorizada de todos os recursos e melhorias necessárias para o produto, o **Sprint Backlog**, um subconjunto do Product Backlog escolhido para o Sprint atual durante o Sprint Planning, e o **Increment**, a soma de todos os itens do Product Backlog concluídos no final de uma Sprint, potencialmente entregáveis e que atendem à Definition of Done [36].

Por último, o Scrum é regido por regras específicas, incluindo o **time-boxing** que garante uma cadência e adaptabilidade consistentes e promove a colaboração entre os membros da equipa e

as partes interessadas, a **melhoria contínua** que é incentivada por meio de inspeções regulares e adaptações com base no *feedback*, e uma **Definition of Done** que estabelece critérios claros para a conclusão de cada item do Product Backlog.

Concluindo, o Scrum é uma estrutura leve e flexível que permite que as equipas respondam rapidamente às mudanças dos requisitos, facilitando, em última análise, a entrega de produtos de alta qualidade [36].
No caso do projeto em questão, a metodologia de desenvolvimento Scrum vai ser implementada através do uso do Azure Devops, mais especificamente o serviço Azure Boards, acima mencionado, uma vez que este permite a utilização de diversas ferramentas e recursos que vão facilitar a gestão ágil do projeto.
É ainda de notar que o processo de implementação do Scrum no Azure DevOps foi realizado a posteriori, portanto não será abordado em detalhe neste trabalho de projeto.

4.2.2 Equipa do projeto

A equipa responsável por este projeto é constituída por vários elementos onde cada um possui as suas responsabilidades. Esta é constituída pelo cliente (partes interessadas), pelo Product Owner, pelo scrum master, e por uma Scrum Team que consiste em um Business analytics, em um Technical Arquitect, em três *developers* e um *tester*, e esta está representada na Figura 4.3.

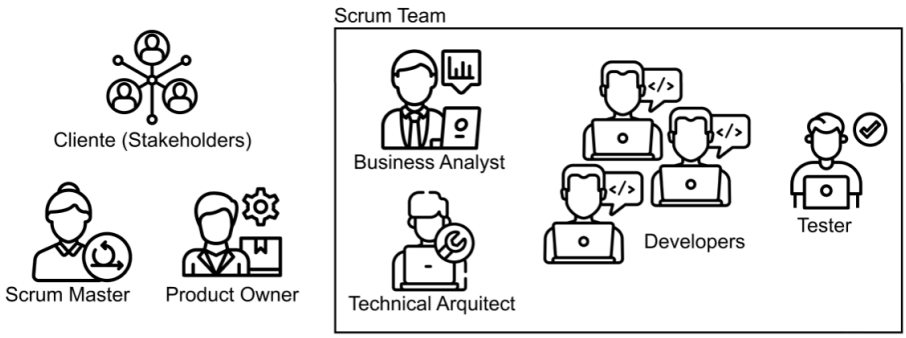

Figura 4.3: Diagrama da equipa do projeto

Resumidamente, as responsabilidades de cada membro da equipa são as seguintes:

- **Product Owner (PO) [36]:** Responsável pelo que a equipa desenvolve e pelo porquê de tal ser desenvolvido. Este é responsável por manter o product backlog atualizado e em ordem de prioridade.
- **Scrum master [36]:** Garante que o processo Scrum é seguido pela equipa e está continuamente à procura de como a equipa pode melhorar, enquanto resolve impedimentos e outros problemas que surgem durante a Sprint.
- **Scrum team [36]:** Constroem o produto e são responsáveis pela engenharia do mesmo e pela qualidade que o acompanha.

- **Business Analytics (BA) [47]:** Responsável pela análise sistemática de dados, de modo a extrair *insights* significativos e a facilitar a tomada de decisões baseada em dados (*data-driven decision-making*, em inglês) dentro de um ambiente organizacional. Este recorre a uma variedade de técnicas, incluindo análise descritiva para resumir dados históricos, análise de diagnóstico para compreender as causas dos problemas, análise preditiva para prever tendências futuras e análise prescritiva para fornecer recomendações de ações. Adicionalmente, o BA permite que as organizações obtenham uma compreensão abrangente do seu desempenho, otimizem processos e impulsionem iniciativas estratégicas.

- **Technical Arquitect (TA) [2]:** Responsável por projetar e supervisionar a implementação de soluções técnicas complexas. Esta função consiste na tradução dos requisitos de negócios em especificações técnicas abrangentes, na tomada de decisões estratégicas sobre a arquitetura do sistema e garantir que as soluções resultantes estão alinhadas aos objetivos organizacionais. Este desempenha um papel fundamental na orientação do ciclo de vida do desenvolvimento, colaborando com as diversas partes interessadas e mantendo-se atualizado sobre as tecnologias emergentes para fornecer recomendações informadas e otimizar os sistemas existentes.

- **Devolpers [55]:** Responsáveis por desenhar, codificar e manter aplicações ou sistemas de software. Desempenham um papel crucial no ciclo de vida de desenvolvimento de software e contribuem para a inovação e eficiência dos sistemas de IT, atendendo a requisitos específicos de negócios ou as necessidades dos utilizadores. Adicionalmente, são responsáveis pela criação de algoritmos, escrita de código em diversas linguagens de programação e colaboração com equipas multi-funcionais para garantir a funcionalidade e a qualidade das soluções de software.

- **Tester [38]:** É responsável por identificar e relatar defeitos, colaborando com os *devolpers* e outras partes interessadas de maneira a garantir que o software atende os requisitos especificados. É um papel crucial no fornecimento de software de alta qualidade, pois deteta e resolve problemas antes da fase de *deployment*, através do desenho e execução de casos de teste.

4.2.3 Processo de testes

O processo de testes desde projeto e os seus respetivos responsáveis estão ilustrados na Figura 4.4, e este segue as fases e as atividades do processo de testes descritas na Seção 2.4 do Capítulo 2.

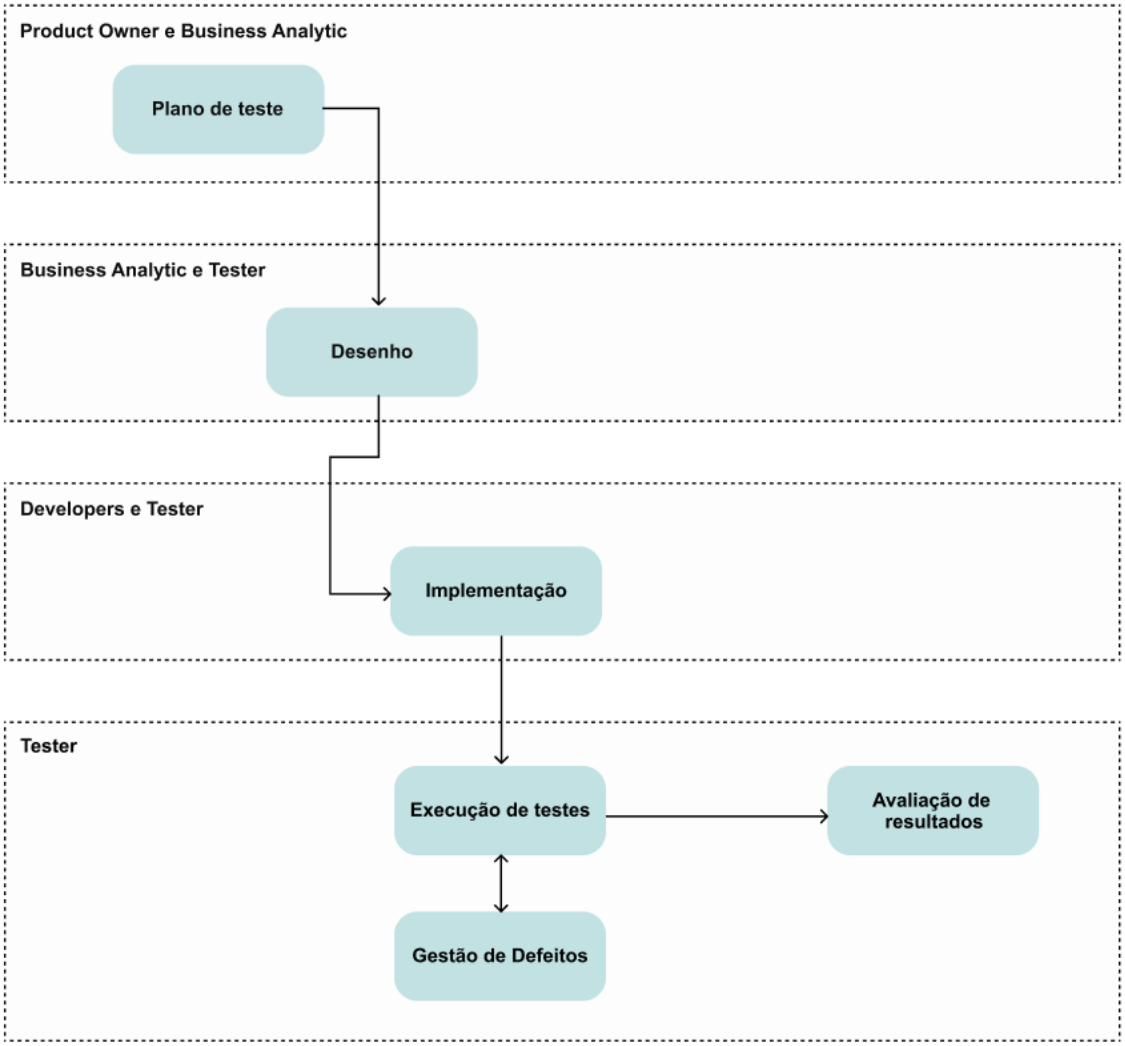

Figura 4.4: Processo de testes

Primeiramente, e após o cliente ter estabelecido e definido os requisitos necessários para o desenvolvimento do projeto, o PO e o BA colaboram estreitamente para refinar esses requisitos em *user stories* detalhadas e em critérios de aceitação, que serão depois utilizados para definir um plano de teste que permite entender quais são as funcionalidades que precisam ser testadas.
Na fase de desenho, o BA e o *tester* identificam e descrevem os cenários de testes necessários em cada *user stories*, de forma a estruturar os procedimentos de teste. De seguida, na fase de implementação, os *developers* e o *tester* trabalham em estreita colaboração e desenvolvem todos os recursos essenciais para a realização e implementação dos testes automáticos. Após a conclusão do desenvolvimento, decorre a fase de execução onde o *tester* executa os testes automáticos e faz

registo dos defeitos ou erros encontrados.
Por fim, os resultados dos testes são avaliados, medindo quantitativamente o progresso dos testes e são gerados relatórios de teste para posterior avaliação.
É ainda de notar que geralmente existe *feedback* a ser transmitido entre as diversas fases e atividades do processo de testes.

4.3 Abordagem

A abordagem de teste refere-se à estratégia adotada para conduzir os testes, definindo como o processo de teste será realizado e podendo variar conforme o contexto do projeto. Esta abordagem inclui a escolha dos tipos de teste, as ferramentas utilizadas, os níveis de teste a serem aplicados (unitários, integração, sistema e aceitação) e os critérios de aprovação para os testes. Uma abordagem bem definida garante que todas as áreas críticas do software são testadas de maneira eficiente e eficaz.
No caso do projeto a desenvolver neste trabalho de projeto, os testes automáticos que serão desenvolvidos são testes funcionais e testes de aceitação. Para a implementação dos mesmos, será utilizada a metodologia ágil de desenvolvimento BDD, escrita em linguagem Gherkin, em conjunto com a *framework* SpecFlow, tópicos estes que já foram explicados em detalhe em capítulos anteriores.
Estes testes automáticos vão ter como objetivo verificar se o código produzido pelos *developers* na fase de desenvolvimento traduz e cumpre todos os requisitos estabelecidos pelo cliente, bem como todos os critérios de aceitação apresentados nas diversas *user stories* definidas pelo Product Owner e pelo Business Analyst.

4.4 Pipeline CI/CD

4.4.1 Integração contínua (CI)

A **Integração Contínua (CI)** é uma prática de desenvolvimento amplamente estabelecida na indústria de desenvolvimento de software, na qual os membros de uma equipa integram e fazem *merge* do trabalho de desenvolvimento com frequência [46]. A CI permite que as empresas de software tenham ciclos de lançamento mais curtos e frequentes, melhorem a qualidade do software e aumentem a produtividade de suas equipas [46].
Esta prática possui a automatização dos processos de *build* e de testes de software [46], tornando possível a existência de *builds* consistentes e a probabilidade de encontrar possíveis problemas o mais cedo possível no SDLC. Tal é exequível uma vez que a CI consegue, através da automatização, proporcionar um SDLC consistente e com interações mínimas com o utilizador [62].
O uso eficaz da CI requer o compromisso da equipa, uma vez que esta consiste fundamentalmente num conjunto de diretrizes que no final produz uma *build*, sendo que para tal, as equipas devem trabalhar de forma consistente durante todo o SDLC [62].
Adicionalmente, trabalhar com uma CI permite que as equipas obtenham melhor visibilidade do

projeto e tenham maior confiança no software desenvolvido, fornecendo informações sobre a situação atual do projeto com base na qualidade geral da *build*. Além disso, também aumenta a confiança no produto de software, pois os testes são executados automaticamente em cada *build*, de modo a verificar o estado correto do software antes do sucesso de uma *build*, reduzindo assim os riscos gerais [62].

4.4.2 Entrega contínua (CD)

A **Entrega Contínua (CD)** é um processo de desenvolvimento de software que estende a CI, que prepara automaticamente o software para lançamento no ambiente de produção [62]. Frequentemente, as pipelines CD fazem *deploy* do software de *build* com sucesso em ambientes de pré-produção automaticamente, o que permite que as equipas de desenvolvimento executem testes mais versáteis, antes de fazer *deploy* do software em produção, e com mais confiança no desempenho geral dos sistemas de software [62].
Este procedimento permite que as equipas tenham um software pronto para produção e para *deploy* [62], garantindo também que uma aplicação se encontra sempre no estado de produção após passar com sucesso nos testes automáticos e nas verificações de qualidade. Ademais, a CD aplica um conjunto de práticas, como a CI e a automatização do *deployment* a fim de entregar software automaticamente para um ambiente semelhante ao de produção [46].
Por fim, esta prática oferece vários benefícios, como a redução do risco de *deployment*, o custo mais baixo e a obtenção de *feedback* do utilizador mais rápida [46].

4.4.3 Pipeline CI/CD

Assim sendo, e tendo em conta os conceitos definidos anteriormente, uma pipeline (Figura 4.5) é um plano de execução, que contém as etapas necessárias para os processos de CI e CD. Esta é usada para melhorar a qualidade geral do lançamento de software e na maioria das vezes, a pipeline fornece estatísticas sobre como o estado de *build* dos projetos, fornecendo assim monitorização e informações valiosas sobre o estado do projeto nas fases de integração e teste para a equipa de desenvolvimento [62].
Dentro dos benefícios de uma pipeline CI/CD encontram-se a descoberta antecipada de defeitos, o aumento a produtividade dos *developers* e a aceleração dos ciclos de lançamento [64]. No entanto, esta também apresenta alguns desafios e barreiras quando utilizada, o que implica que a criação de uma pipeline CI/CD não é uma tarefa simples. Além disso, esta pode ser aplicada incorretamente, o que pode limitar a sua eficácia e introduzir problemas de manutenção [64].

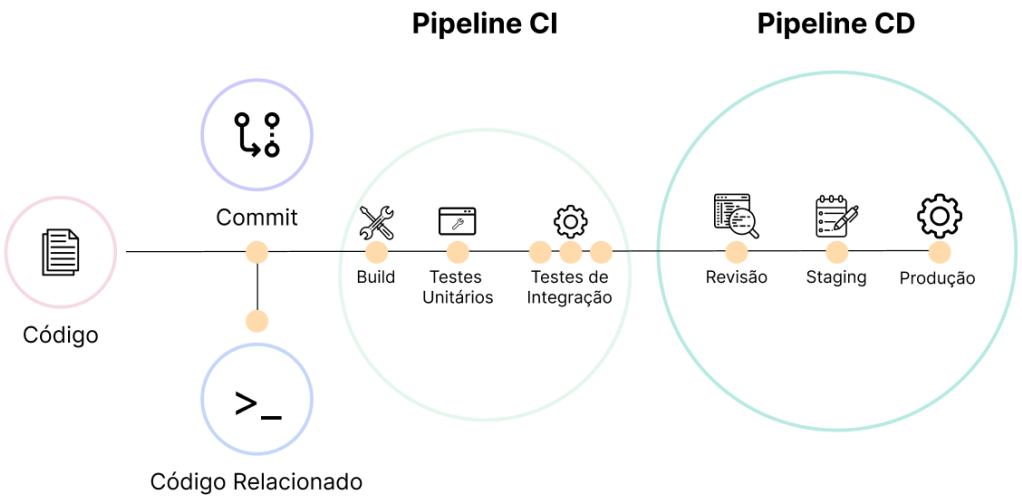

Figura 4.5: Pipeline CI/CD
Baseado em: https://blog.openreplay.com/what-is-a-ci-cd-pipeline/

4.4.4 Azure Pipeline

No projeto vai utilizar a pipeline CI/CD fornecida pelo Azuze DevOps, pois esta é uma solução poderosa que automatiza todo o processo de *build*, teste e *deploy* de aplicações. Esta automatização não acelera apenas o ciclo de lançamento, mas também aprimora a qualidade geral e a confiabilidade do software que são *deploy* no Azure. Ao aproveitar o Azure Pipelines, a equipa pode obter integração contínua e *deployment* contínuo perfeitos, permitindo uma entrega de valor mais rápida e eficiente aos utilizadores finais.

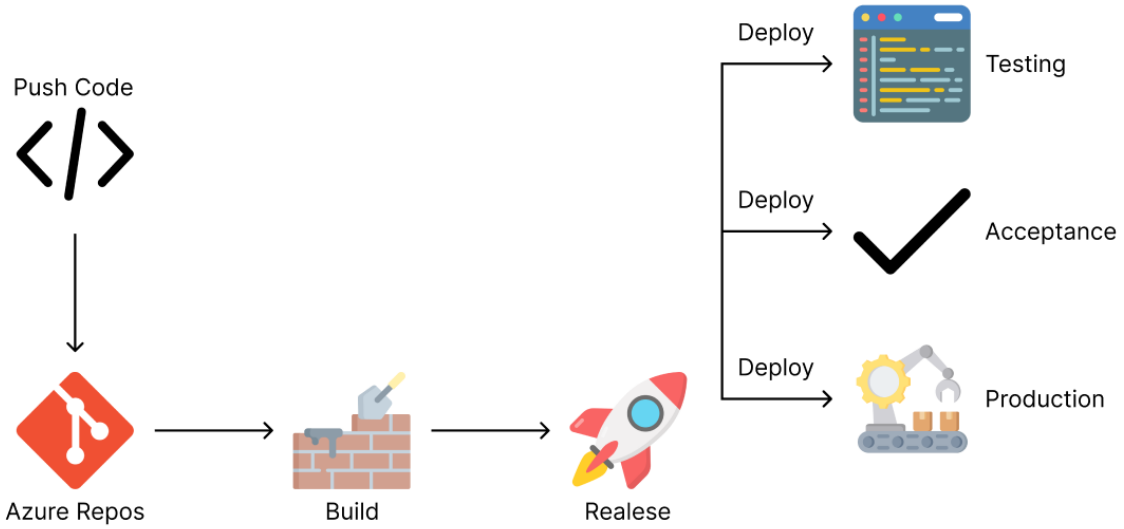

Figura 4.6: Workflow da Azure DevOps Pipeline
Baseado em: https://idocs.info/designing-an-azure-devops-pipeline-to-deploy-blueprints-in-azure/

O *workflow* da Azure Pipeline (Figura 4.6 e Tabela 4.1) começa, com um **trigger**, que pode ser acionado quando um evento como a alteração de código, realizado pelo *developers* e/ou *testers*, é enviado para um *version-controlled repository*, como GitHub ou Azure Repos. O Azure Pipelines monitoriza estes repositórios e inicia automaticamente a pipeline quando deteta novos *commits* (submissão de código) ou *pull requests*, iniciando assim o processo de integração contínua e entrega contínua (CI/CD).

Conceito	**Descrição**
Trigger	Diz à pipeline para executar.
Pipeline	Formada por um ou mais *stages* e pode ser *deploy* para um ou mais ambientes.
Stage	Forma de organizar os *jobs* em uma pipeline e cada *stage* pode ter mais que um *job*.
Job	Cada *job* é executado num agente, mas também pode não ter nenhum agente.
Agente	Cada agente executa um *job* que contém um ou mais steps.
Step	Pode ser uma *task* ou um *script* e é a unidade mais pequena da pipeline.
Task	É um *script* predefinido que executa uma ação.
Artefacto	Conjunto de ficheiros ou pacotes publicados por uma execução.

Tabela 4.1: Conceitos chaves de uma pipeline do Azure Devops

A componente principal do Azure Pipeline é a própria **pipeline**, que define um conjunto de *steps* que necessitam de ser executados para transformar o código-fonte numa aplicação funcional. A pipeline é composta por vários ***stages***, sendo os mais comuns *build*, *test* e *release*, e cada um deles representa uma fase principal do processo de CI/CD.

Cada *stage* pode ser dividido em diversos *jobs*. Um ***job*** é um conjunto de *steps* (unidade mais pequena de uma pipeline), que são executados sequencialmente ou em paralelo num único agente, e estes ajudam a organizar e isolar diferentes *tasks* que precisam de ser executadas no *stage*. Ou seja, dentro de cada *job*, existem *steps* individuais. Os ***steps*** são executados sequencialmente num job e podem incluir uma variedade de *tasks*. Uma ***task*** é um *script* predefinido ou personalizado que executa uma ação específica, como executar um *script*, invocar uma ferramenta de *build* ou executar um conjunto de testes.

Adicionalmente e como referido, cada *job* é executado num **agente**, que é uma máquina virtual (VM) ou um *container* e pode ser configurado com um sistema operacional específico e dependências de software necessárias, de modo a construir e testar um projeto e a fornecer o ambiente necessário para executar os *steps* definidos no *job*.

Conforme a pipeline avança nos seus *stages*, *jobs* e *steps*, esta produz artefactos. Os **artefatos** são os *outputs* do processo de compilação e estes são armazenados e geridos pelo Azure Pipelines, tornando-os disponíveis para *stages* e *deployments* futuros.

Normalmente, o primeiro *stage* deste processo é o ***build stage***. Neste, a pipeline recupera o código-fonte mais recente do repositório, compila-o em artefactos executáveis e é executado num agente designado. Depois de um *build stage* bem-sucedido, a pipeline prossegue para o ***test stage***, onde o projeto passa por um grupo de testes automáticos abrangentes de maneira a validar a sua funcionalidade, desempenho e segurança. Por fim, e após passar em todos os testes, a pipeline avança para o ***release stage***, onde os artefactos são preparação para o seu *deployment* no ambiente de destino.

Capítulo 5

Implementação

Após delinear a estratégia de testes no capítulo anterior, este capítulo vai descrever a implementação dessa estratégia no projeto e irá apresentar o processo de testes para um dos requisitos de negócio solicitados pelo cliente. Para proporcionar uma compreensão mais clara do requisito a ser automatizado, será exposta uma *user story* e os seus respetivos critérios de aceitação, juntamente com os seus diversos cenários de teste. Em seguida, será demonstrado como a estratégia discutida anteriormente foi implementada para esse requisito, detalhando o plano, o design, a implementação, a execução e a avaliação dos resultados dos testes.
Assim sendo, o requisito de negócio estabelecido pelo cliente e escolhido para ser exemplificado neste trabalho de projeto foi o processo de ingestão de dados de uma das componentes do projeto. Esse requisito tem como objetivo o desenvolvimento de todo o código necessário para ingerir o conjunto de dados enviados pelos IDNOs[1] nessa componente, de forma a manter a integridade dos dados e a conformidade com *templates* predefinidos.
Por fim, é importante destacar que a autora deste trabalho de projeto atuou como a única *tester* da equipa. Portanto, e salvo indicação em contrário, todo o processo e o trabalho que serão descritos a seguir foram realizados pela mesma. Cabe também ressaltar que, ao longo de todo o processo, os demais membros da equipa (BA, TA e *developers*) estiveram sempre disponíveis para auxiliar e esclarecer dúvidas sobre o trabalho de desenvolvimento dos testes automáticos.

5.1 Plano de testes e Desenho

5.1.1 User story e criterios de aceitação

Na metodologia de desenvolvimento ágil escolhida, o Scrum, as ***user stories*** são uma componente essencial do processo de desenvolvimento, uma vez que estas vão fornecer uma descrição direta, clara e breve de um recurso ou funcionalidade do ponto de vista do utilizador final ou cliente.

[1] Os IDNOs (*Independent Distribution Network Operators*) são empresas que possuem e operam redes de distribuição de eletricidade privadas mais pequenas. Eles fornecem uma alternativa aos *Distribution Network Operators* (DNOs) já estabelecidos, que são regulamentados pela Ofgem para garantir concorrência justa e um serviço confiável.

O formato mais comum para a escrita de uma *user story* é o seguinte [12]:

ENQUANTO [determinado utilizador]
EU QUERO [executar uma ação]
PARA QUE [obtenha um certo benefício]

Este modelo garante que cada *user story* responde as seguintes questões [12]:

- **Papel:** Quem quer o recurso? Quem é o utilizador?
- **Objetivo:** O que eles querem exatamente? Qual é a intenção do recurso?
- **Motivo:** Porque é eles o querem? Qual é o seu valor para o cliente?

Já no que toca aos **critérios de aceitação**, estes são as condições que uma *user story* deve atender para a mesma ser aprovada pelo *Product Owner*. Este oferecem definições claras das expectativas e explicam como os testes tem de ser desenvolvidos para confirmar a implementação da *user story*. Ademais, os critérios garantem que a *user story* fornece o valor pretendido e atende aos requisitos do utilizador.
Deste modo, e a partir da análise do requisito de negócio acima mencionado, o *Product Owner* e o *Business Analytics* da equipa elaboram uma *user story* detalhada e definiram os seus respetivos critérios de aceitação, sendo ambos descritos em seguida.

User Story 109519: Ingestão de dados da componente X
ENQUANTO conta do Azure Storage
EU QUERO fazer a ingestão do conjunto de dados de um ficheiro CSV para a componente X.
PARA QUE possa guardá-lo e prosseguir com o processo de validação e extração.

AC#1 O ficheiro submetido é guardado no Azure
DADO o ficheiro CSV submetido no Azure Blob Storage
QUANDO o Azure recebe o ficheiro
ENTÃO ele ingere e guarda os dados numa tabela numa *storage account* do Azure

AC#2 O ficheiro submetido tem as colunas na ordem correta
DADO o ficheiro submetido pelo IDNOS
QUANDO tem um número total de 6 colunas na seguinte ordem:

Host MPID	Host LLFC	Embedded MPID	Embedded LLFC	Effective from	Effective to

ENTÃO o ficheiro é aceite, caso contrário falha e a seguinte mensagem "The File is non compliant with the file template"é mostrada.

AC#3 O ficheiro submetido tem nome das colunas correto

DADO o ficheiro submetido pelo IDNOS

QUANDO tem colunas com a nomenclatura: Host MPID, Host LLFC, Embedded MPID, Embedded LLFC, Effective from, Effective to

ENTÃO o ficheiro é aceite, caso contrário falha e a seguinte mensagem "The File is non compliant with the file template"é mostrada.

5.1.2 Desenho

Depois da análise e definição da *user story* e respetivos critérios de aceitação, e em conjunto com o *Business Analytics* da equipa, foram identificados e ilustrados os cenários de teste necessários para a *user story* acima referida.

O processo de projeção dos possíveis cenários de teste consistiu na identificação e breve descrição dos testes que podiam ser desenvolvidos de forma a demonstrar que o requisito pedido pelo cliente foi elaborado corretamente. A Figura 5.1 mostra o resultado desse processo de investigação.

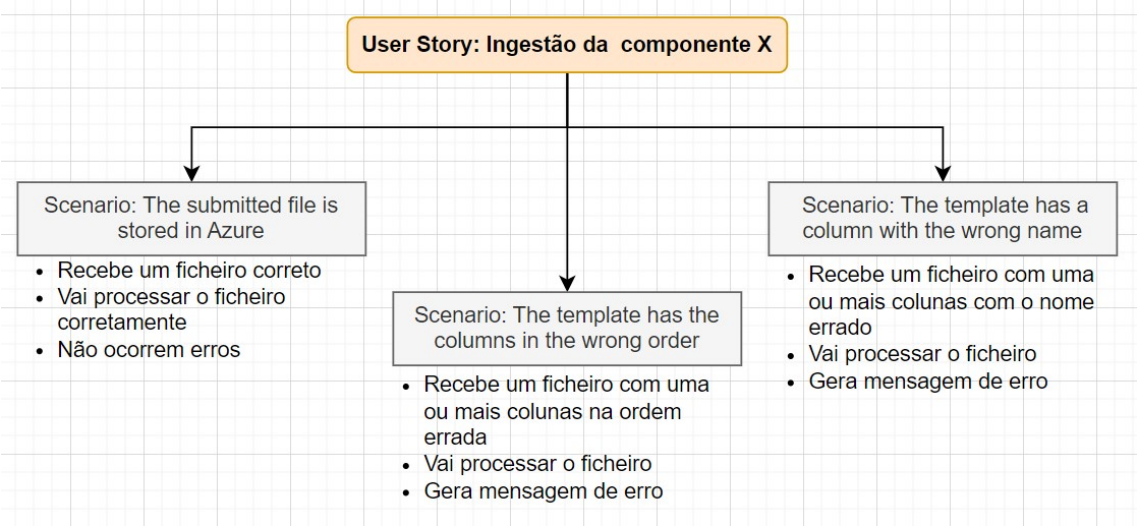

Figura 5.1: Diagrama de cenários de teste

Na Figura 5.1 é possível visualizar que para a *user story*, intitulada de "Ingestão da componente X", foram idealizados três cenários de teste. O primeiro cenário de teste que corresponde ao cenário ideal, onde se vai submeter um ficheiro CSV no formato correto e preestabelecido e os outros dois cenários que vão corresponder a cenários de erro, onde os ficheiros submetidos vão conter erros em relação a ordem das colunas e a nomenclatura das mesmas.

Para além da conceção do diagrama de cenários de teste, também foi projetada e criada a estrutura e organização do *test plan* e suas respetivas pastas no Azure DevOps.

Como é possível ver na Figura 5.2, existe um *test plan* para cada iteração do Scrum com o nome de "WP3 - ISD Azure - PI*i*", onde o *i* corresponde ao número da iteração. Dentro de cada *test plan* existem vários *test suites* que correspondem às diversas *sprints* presentes numa iteração.

Adicionalmente cada *test suite* é divido nas pastas "Functional"e "Non Functional"que vão conter os testes funcionais e não funcionais desenvolvidos, respetivamente. Por fim, e relativo a pasta "Functional", esta foi ainda subdivida de modo a representar as várias componentes que o projeto possui, onde dentro da pasta de cada uma será possível encontrar todos os testes automáticos feitos para essa componente numa certa *sprint* e numa determinada iteração.

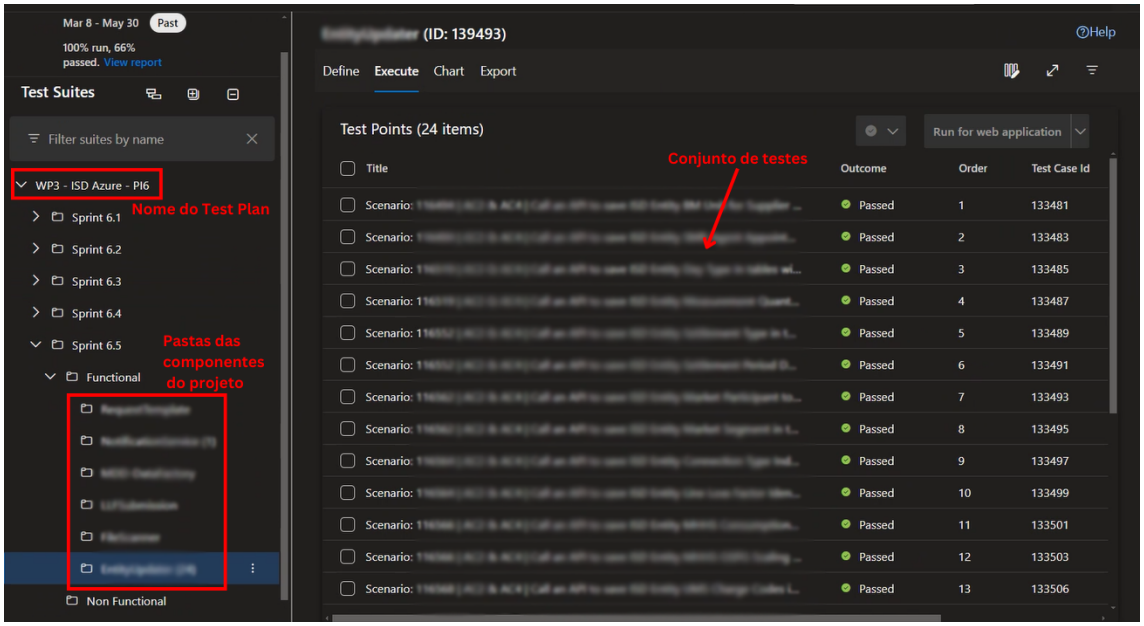

Figura 5.2: Estrutura do Test Plan no Azure

5.2 Implementação

A fase de implementação dos testes automáticos começa após uma nova *user story* ter sido iniciada, e em paralelo com o desenvolvimento do código necessário para implementar o requisito solicitado na mesma, pelos os *developers*, no projeto de desenvolvimento. Isto é, enquanto os *developers* produzem o código de desenvolvimento, em simultâneo, o *tester* da equipa inicia a construção do código de teste que vai verificar se o código produzido pelos *developers* está correto e a funcionar conforme é esperado.

Antes de passar à explicação do processo de desenvolvimento dos testes automáticos, é importante clarificar o modo como o projeto de testes está organizado. Adicionalmente, é também importante referir que, antes da criação do projeto, recorreu-se à instalação do Visual Studio, tendo sido adicionada a extensão relativa ao SpecFlow, bem como os NuGet *packages* e soluções extra necessárias para desenvolver o projeto. No entanto, este tópico vai além do âmbito do projeto, por isso não será abordado em detalhe.

O projeto de teste intitulado "WP3-ISD-QA"é constituído por vários sub-projetos que correspondem às diversas componentes do projeto e cada componente está organizada como ilustrado na Figura 5.3.

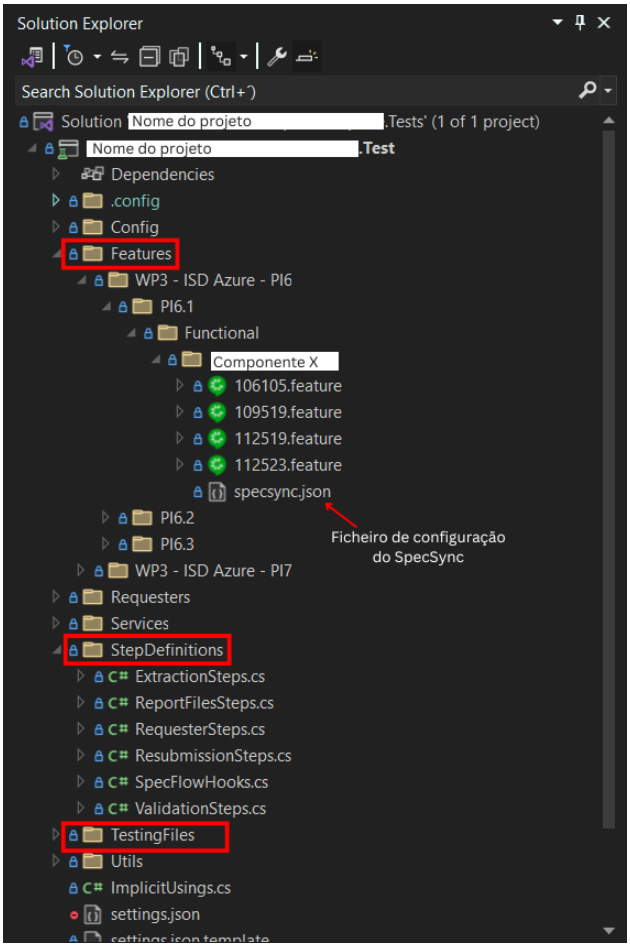

Figura 5.3: Estrutura do projeto de teste

Através da Figura 5.3, é possível verificar as diversas pastas do projeto de teste para cada uma das componentes, sendo as mais relevantes as pastas *Features*, *StepDefinitions* e *TestingFiles*.

- Na **pasta *Features*** estarão localizados todos os ficheiros *feature* do projeto, e esta possui uma estrutura de sub-pastas igual à do *test plan*, acima mencionada. Tal foi implementado de modo a colocar os testes automáticos no sítio correto do *test plan* quando for realizada a sincronização do código para o serviço Test Plan do AzureDevOps. Adicionalmente, esta sincronização será feita através do SpecSycn, assunto que será abordado mais adiante.

- Por outro lado, a **pasta *StepDefinitions*** vai conter todos os ficheiros .cs, relativos os *step definitions*.

- Por fim, a **pasta *TestingFiles*** vai englobar todos ficheiros CSV que vão ser utilizados nos testes.

Passando agora para o processo de desenvolvimento dos testes automáticos. Primeiramente, e sempre que se pretende criar novos testes ou fazer alterações a testes já existentes, cria-se uma nova *branch* a partir da *main branch*, onde todo o código necessário para os testes será realizado, com o nome *feature/[nº_US]-[pequena_descrição]*, de modo a facilitar a identificação da mesma, caso seja necessário. Este processo é feito através do Git e facilitado pelo Visual Studio.
Em seguida, vai criar-se um novo ficheiro *feature*, na pasta "Features"do projeto, cujo nome corresponde ao número da respetiva *user story*. No exemplo da Figura 5.4, o nome do ficheiro *feature* corresponderá ao número *user story* mencionada anteriormente, "Ingestão da componente X".

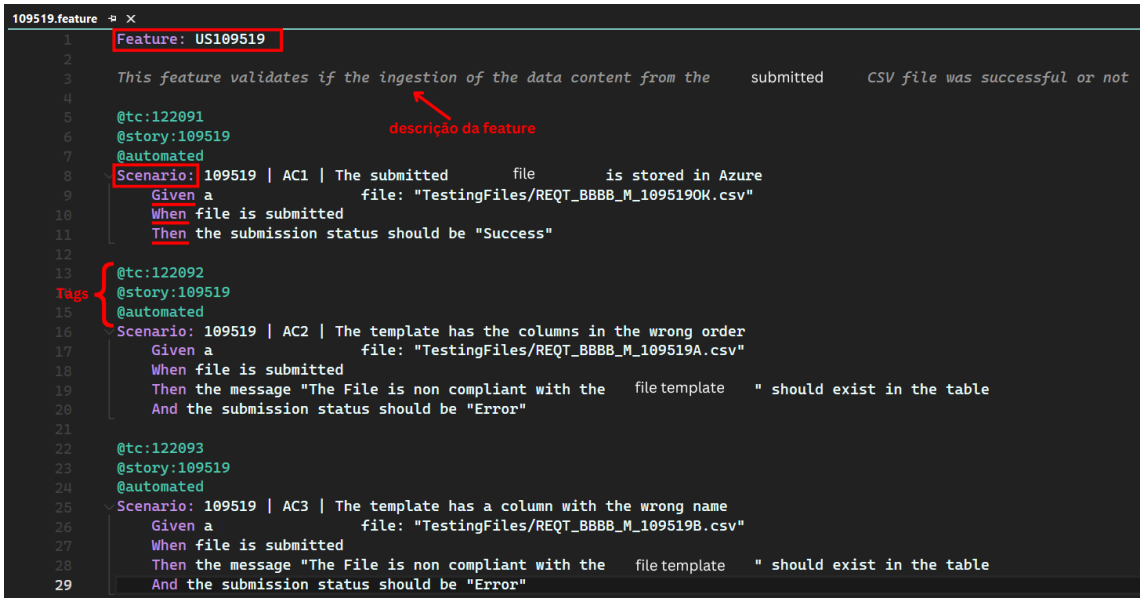

Figura 5.4: Código do ficheiro .feature

Ainda na Figura 5.4 é possível ver todo o código produzido relativo aos *scenarios* BDD, que traduzem a informação contida na descrição da *user story* em testes percetíveis por todos os que os leem, através do uso da linguagem Gherkin.
No código acima pode-se notar que foram desenvolvidos os mesmo três cenários de teste descritos no diagrama de cenários de teste (Figura 5.1), e também é possível identificar algumas das palavras-chave, conhecidas como *steps*, correspondentes à sintaxe Gherkin.
O ficheiro *feature* começa com a palavra “Feature” que representa o nome da funcionalidade que se pretende testar, que neste caso é o número da *user story* 109519, seguida de uma pequena descrição relativa a funcionalidade. Depois encontram-se três *tags* (@):

- A ***tag* @tc** que representa o número do *test case* (caso de teste, em Português);
- A ***tag* @story** que indica o número da *user story*;
- A ***tag* @automated** que indica que o "Scenario"abaixo corresponde a testes automáticos.

Complementarmente, a tag @tc foi gerada automaticamente através da solução SpecSync, e em conjunto com as outras duas *tags*, ajudará a colocar os diversos cenários de teste no sítio correto do Test Plan do Azure DevOps.
Também é possível observar que o *step* "Scenario"resume o contexto e os resultados da funcionalidade. Neste caso, existem três cenários de teste, onde cada um corresponde a um dos critérios de aceitação da *user story* que se pretende testar. Dentro do "Scenario", existem os *steps* "Given", "When"e "Then"que descrevem as pré-condições para o *scenario*, especificam as ações ou eventos a ser executados, e definem o resultado dos *steps* anteriores, respetivamente.

Após a conclusão do ficheiro *feature*, os *developers* da equipa precedem à criação dos ficheiros .cs que possuem os vários *step definitions* que automatizam os *scenarios* ao nível do *step*. Na Figura 5.5, está ilustrado o código dos *step definitions*, escrito em C#, relativo aos *steps* definidos nos *scenarios* acima demonstrados. Como já referido, os *steps definitions* foram criados por outros membros da equipa, por isso estes não serão abordados em pormenor.

```
[Given(@"a                file: ""(.*)""")]
public void      GivenASubmittedFileTemplate   (string file)
{
    FilePath = file;
}

[When(@"file is submitted")]
public void WhenFileIsSubmitted()
{
    if (FilePath != null)
    {
        _blobContainerDriver.UploadFileInBlobStorage(ContainerPath, FilePath);
    }
}

[Then(@"the message ""(.*)"" should exist in the table")]
public void ThenMessageShouldExistInTable(string message)
{
    var exists = _tableDriver.RetryTableQuery<   FileTemplateReportMessage   >(ReportMessagesTable, GetTableQuery(message), MaxAttempts, MillisecondsTimeout);

    exists.Should().BeTrue();
}

[Then(@"the submission status should be ""(.*)""")]
public void ThenSubmissionStatusShouldBe(string status)
{
    var exists = _tableDriver.RetryTableQuery<   FileTemplateSubmission   >(SubmissionsTable, GetTableQueryStatus(status), MaxAttempts, MillisecondsTimeout);

    exists.Should().BeTrue();
}
```

Figura 5.5: Código do ficheiro .cs

5.3 Execução dos testes

Depois de ambos os ficheiros *feature* e *step definitions* estarem concluídos, recorre-se à execução local dos testes desenvolvidos. Isto é realizado através do Test Explorer do IDE Visual Studio, como é possível ver na Figura 5.6. No Test Explorer, são listados todos os testes disponíveis, e pode-se executar cada teste individualmente ou todos os testes de uma vez.
Como é visível na figura 5.6, todos os testes relativos à *feature* 109519 passaram na execução local. A execução local dos testes, através do Test Explorer, vai fornecer *feedback* imediato sobre o sucesso ou falha de cada teste, permitindo identificar rapidamente quaisquer problemas.

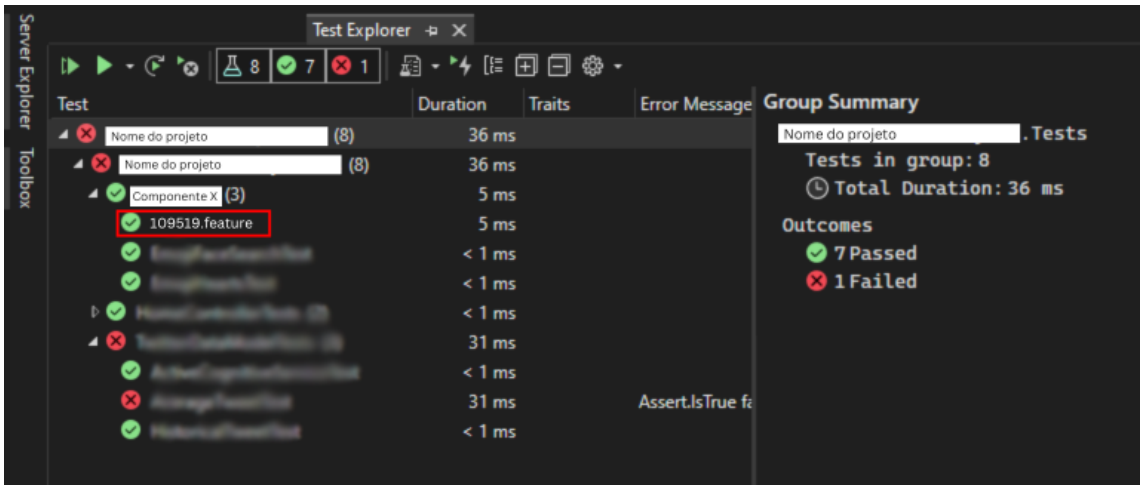

Figura 5.6: Execução de testes no Visual Studio

Quando todos os testes passam localmente, estes estão prontos para serem testados no serviço de pipeline do Azure DevOps. Para tal, primeiro é preciso sincronizar o *test plan* do Azure DevOps através do **SpecSync** e submeter as alterações do código através do **Git**.

Como já referido brevemente, o **SpecSync** é uma ferramenta que facilita a sincronização de especificações BDD com o Azure DevOps. Este permite que os cenários de teste definidos nos ficheiros *feature* são sincronizados com o Azure DevOps Test Plans, garantindo que a documentação e a implementação dos testes estão alinhadas corretamente [54].
Para o SpecSync funcionar é necessário recorrer à sua instalação e à integração do ficheiro de configuração (*specsync.json*) no projeto. No caso do projeto elaborado neste trabalho de projeto, foram necessários dois ficheiros de configuração, um localizado no diretório raiz do projeto e outro localizado dentro da pasta que contém os ficheiros *feature*. Estes ficheiros vão definir como o SpecSync se deve comportar e quais parâmetros utilizar para a sincronização.
Após configurar o ficheiro *specsync.json*, pode-se sincronizar os testes com o Azure DevOps utilizando o seguinte comando no terminal: *dotnet specsync pull*. Este comando vai procurar por novos testes ou alterações em testes existentes e atualizá-los no Test Plan do Azure DevOps.

Depois da sincronização dos testes com o Azure DevOps Test Plans pelo SpecSync, procede-se à submissão das alterações realizadas no código através do **Git**. Esse processo é facilitado pelas ferramentas disponibilizadas pelo Visual Studio e é essencial para manter o código atualizado e compartilhado entre todos os membros da equipa.
Para isso, inicialmente realiza-se o *commit* e o *push* das mudanças para o repositório do Azure DevOps. Em seguida, no próprio repositório do Azure DevOps, é criado um *pull request*[2] para a branch principal (*main branch*). Os membros da equipa podem então rever o *pull request*, podendo

[2]Um *pull request* é uma solicitação para que outros membros da equipa verifiquem as alterações de codigo antes de integrá-las na main branch.

fazer comentários, solicitar alterações ou aprovar diretamente as alterações propostas. Finalmente, quando o *pull request* é aprovado, as alterações são inseridas na *main branch*, sendo este processo realizado automaticamente, através da interface do Azure DevOps, a partir do momento em que se dá "*complete*"ao *pull request* aprovado.

Por fim, e depois da aprovação do código das alterações pelo resto da equipa, é possível prosseguir com a execução dos testes utilizando o serviço de pipeline do Azure DevOps.
A pipeline de testes do Azure DevOps foi configurada a posterior, de maneira a refletir a estrutura do projeto. A configuração pode ser realizada através de um ficheiro YAML ou pela interface gráfica do Azure DevOps, no entanto este processo não será explicado pois foi realizado por outro elemento da equipa.
A pipeline é dividida em vários *stages* (Figura 5.7), que correspondem às diferentes componentes do projeto e cada *stage* tem um conjunto específico de testes que devem ser executados, garantindo que todas as partes do projeto são verificadas.

Figura 5.7: Stages da pipeline

Quando a pipeline é acionada, seja automaticamente após um *commit* ou manualmente, todos os *stages* definidos são executados sequencialmente ou em paralelo conforme configurado, e em cada *stage*, os testes correspondentes são executados.
Após a execução da pipeline, é possível visualizar imediatamente a percentagem de testes que foram aprovados. Esta informação é apresentada numa *dashboard* (Figura 5.8) intuitiva no Azure DevOps, permitindo uma rápida avaliação do estado dos testes. Caso hajam testes a falhar, o Azure DevOps fornece detalhes sobre quais testes falharam e as razões para tal falha. Esta informação é crucial para a equipa de desenvolvimento conseguir corrigir os problemas antes de integrar o código alterado em ambientes de produção. Adicionalmente, o Azure DevOps fornece *logs* detalhados da execução de cada teste, o que facilita também todo o processo de identificação e correção de problemas pela equipa.

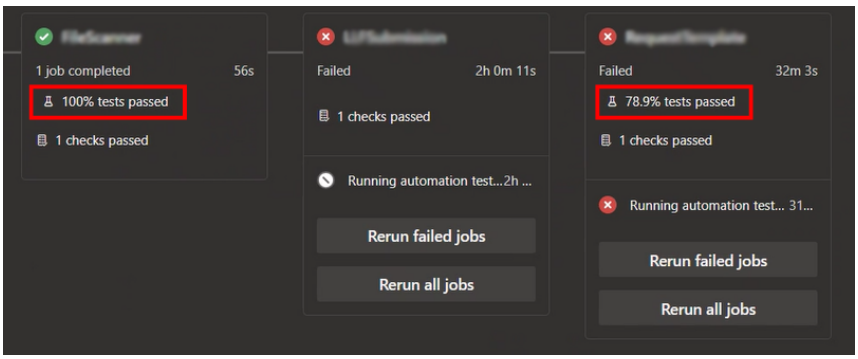

Figura 5.8: Parte do *dashboard* do resultado da execução dos testes

Como é possível ver na Figura 5.8, numa das execuções efetuadas, num dos *stages* 100% os testes passaram e noutro apenas passaram 78.9%.

O progresso e os resultados da execução da pipeline são apresentados no *"Progress Report"* (Figura 5.9). Este relatório detalha a execução de cada *stage*, os testes executados, os resultados (pass/fail), e outras métricas relevantes. O *progress report* serve como *feedback* para a equipa, ajudando a manter a qualidade do código. Com base nos resultados apresentados, a equipa pode tomar decisões informadas sobre o próximo passo no ciclo de desenvolvimento.

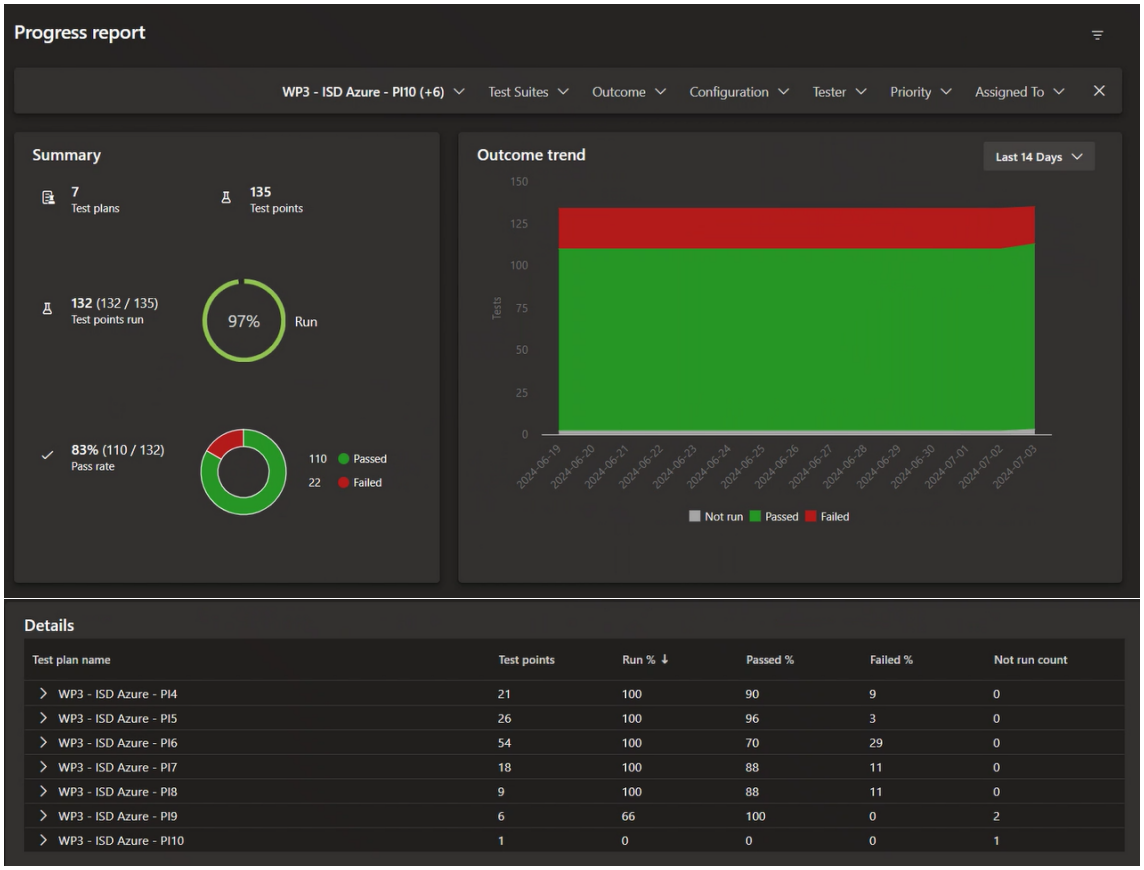

Figura 5.9: Progress Report dos testes

5.4 Testes realizados

Até ao momento, e conforme mostrado na Figura 5.9, foram realizados um total de 132 testes automáticos. Os testes podem ser classificados em duas principais dimensões: de acordo com as validações realizadas e de acordo com o tipo de testes efetuados.

No que diz respeito às **validações realizadas**, os testes foram divididos em validações técnicas e validações de negócio. As **validações técnicas** incluem avaliações de desempenho, que medem a eficiência e a rapidez com que o sistema executa operações, avaliações de segurança para identificar vulnerabilidades e garantir a proteção de dados sensíveis, e avaliações da integridade dos dados

para assegurar que não ocorrem alterações ou perdas durante o processamento e armazenamento de dados. Por outro lado, as **validações de negócio** garantem que o sistema atenda aos requisitos específicos do cliente e às necessidades do negócio. Estas envolvem testes de funcionalidade para confirmar que todas as funcionalidades estão de acordo com as especificações, avaliações de usabilidade para assegurar que o sistema é intuitivo e eficiente para os utilizadores finais, e verificações de conformidade com requisitos regulatórios para garantir que o sistema cumpre todas as normas legais aplicáveis. Assim, a quantidade de testes para cada tipo de validação está descrita na Tabela 5.1. É ainda de notar que, apenas certas componentes do projeto possuem testes de validações técnicas e de validações de negócio, o que justifica o número de testes apresentados na tabela ser inferior ao número total de testes realizados.

Tipo de validação	**Quantidade de testes**
Validações técnicas	9
Validações de negócio	40

Tabela 5.1: Quantidade de testes e tipo de validações realizadas

Quanto ao tipo de **testes realizados**, os testes dividem-se em testes de sistema e testes lógicos. Os **testes de sistema** englobam testes end-to-end, que avaliam o sistema completo para garantir que todos os componentes funcionam corretamente em conjunto, testes de integração que verificam a comunicação e a interação entre diferentes módulos e sistemas externos, e testes de regressão que asseguram que alterações no sistema não introduzem novos problemas em funcionalidades previamente corretas. Os **testes lógicos**, por sua vez, incluem testes de unidade que avaliam partes individuais do código ou componentes para garantir que cada um funciona corretamente isoladamente, testes de controlo de fluxo que verificam a lógica e a sequência das operações dentro do sistema, e testes de condição e decisão que validam que todas as condições e decisões lógicas no código são tratadas adequadamente, cobrindo todos os caminhos e cenários possíveis. Deste modo, a quantidade de testes para cada tipo de testes realizado está descrita na Tabela 5.2.

Tipo de testes	**Quantidade de testes**
Testes de sistema	90
Testes lógicos	42

Tabela 5.2: Quantidade de testes e tipo de testes realizados

5.5 Gestão de defeitos, alterações e reteste

Durante todo o processo de desenvolvimento dos testes automáticos do projeto, houve momentos em que alguns testes falhavam, tanto localmente quanto na pipeline do Azure DevOps. Estes problemas ocorreram devido a diversos fatores, como erros no código de desenvolvimento, erros no próprio teste, problemas de configuração, dependências entre as diferentes componentes do projeto e outras inconsistências que surgiam no decorrer do desenvolvimento.
Assim, e sempre que se encontrava algum problema no resultado da execução dos testes automáti-

cos, o processo de gestão de defeitos era iniciado. Esse processo começa com a deteção do defeito pelo *tester* da equipa, neste caso a autora do trabalho de projeto.

Após a deteção, o defeito é registado e descrito de forma clara. No caso deste projeto, este registo é feito através da criação de um *bug* no *board* de desenvolvimento da equipa, fornecido pelo Azure DevOps. Neste é descrito o problema encontrado, quais os passos para reproduzir o defeito, a prioridade e severidade do defeito e informações sobre o ambiente de teste em que o defeito foi encontrado.

Em seguida, recorria-se a análise e ao diagnóstico do defeito. Nesta etapa, os *devolpers*, em conjunto com o *tester*, analisam o defeito de modo a entender a sua causa. Isso pode envolver a análise do código, uso de ferramentas de *debug* para rastrear a origem do problema e a análise de *logs* de sistema para obter mais informações sobre o defeito.

Depois de diagnosticado, os *developers* implementam uma correção para o defeito. Isso pode envolver a modificação do código-fonte de desenvolvimento e/ou de teste, ajustes nas configurações do projeto ou de ambiente, entre outras.

Uma vez que a correção é implementada, a correção é validada através do reteste. Isto é, o(s) teste(s) com defeito são testados novamente, localmente e na pipeline, para garantir que problema foi corrigido, e testes de regressão adicionais são realizados para garantir que a correção não introduziu novos defeitos em outras partes do sistema.

Se o reteste for bem-sucedido, o defeito é marcado como resolvido no board do Azure. Esta ação inclui a atualização do estado do defeito para "resolvido"ou "fechado", e a documentação das ações tomadas e resultados dos testes para referência futura.

É de notar que este é um processo iterativo e que vai ser repetido várias vezes até que o projeto esteja livre de defeitos conhecidos e a funcionar conforme esperado.

Capítulo 6

Avaliação

Neste capítulo será avaliado o sucesso e a eficácia do trabalho realizado no projeto. Primeiro serão descritas as métricas de avaliação e desempenho utilizadas para medir o cumprimento dos objetivos do projeto. Em seguida, irão ser apresentados os resultados das mesmas, oferecendo *insights* quantitativos sobre as realizações do projeto e áreas que necessitam de melhorias. Finalmente, será discutido a importância da automatização no processo de avaliação, destacando como ela melhora a precisão e a eficiência das avaliações de desempenho.
Sucintamente, este capítulo vai fornecer uma visão geral concisa do processo de avaliação, focando-se nas métricas, nos resultados dos KPIs e no papel crucial da automatização.

6.1 Métricas de Avaliação e Desempenho

De forma a medir o impacto da implementação dos testes automáticos no projeto, alguns Key Performance Indicators (KPIs) foram selecionados.
Os Key Performance Indicators (KPIs) são indicadores críticos quantificáveis de progresso em direção a um resultado pretendido. Os KPIs fornecem um foco para melhorias estratégicas e operacionais, criam uma base analítica para a tomada de decisões e ajudam a concentrar a atenção no que é mais importante [24].
Um bom KPI pode fornecer diversas vantagens. Dentro destas, as vantagens mais relevantes são [24]:

- Evidências objetivas do progresso, de modo a alcançar um resultado desejado;
- Mede o que é necessário avaliar para ajudar numa melhor tomada de decisão;
- Oferece uma comparação que avalia o grau de mudança do desempenho ao longo do tempo;
- Pode monitorizar a eficiência, a eficácia, a qualidade, a pontualidade, a governança, a conformidade, os comportamentos, a economia, o desempenho do projeto, o desempenho da equipa ou a utilização de recursos.

Desde modo, os KPIs selecionados para medir o impacto e a eficácia da implementação de testes automáticos em vez de testes manuais foram os seguintes:

1. **Cobertura dos testes automáticos [13]:** Mete até que ponto o conjunto de testes automáticos cobrem as funcionalidades do software em comparação com o conjunto de funcionalidades totais. Este é uma percentagem calculada através da divisão do número de funcionalidades testadas automaticamente pelo número total de funcionalidades do software. Uma cobertura mais alta indica que um conjunto mais amplo de funcionalidades está a ser testado automaticamente, levando a uma melhor qualidade do software.

2. **Tempo de execução dos testes [26]:** Refere-se à duração necessária para executar um conjunto de cenários de teste. Com testes automáticos, este KPI provavelmente irá diminuir significativamente em comparação com os testes manuais. O tempo de execução mais curto permite ciclos de testes mais frequentes, *feedback* mais rápido sobre alterações de código e resolução de problemas mais rápida.

3. **Esforço de manutenção dos testes [15]:** Quantifica o tempo e o esforço gastos na manutenção dos testes automáticos em comparação com os testes manuais. Como os testes automáticos são projetados para serem reutilizados e de fácil manutenção, um menor esforço de manutenção indica que a automatização levou a uma maior eficiência e à redução de despesas gerais na manutenção de testes.

4. **Eficiência dos testes de regressão [23]:** Garante que novas alterações no código não afetam negativamente as funcionalidades já existentes. Os testes automáticos são excelentes para a realização de testes de regressão rápidos e repetitivos. Ou seja, este KPI mede quanto tempo é economizado em ciclos de testes de regressão ao automatizar casos de testes repetitivos, permitindo que as equipas de desenvolvimento entreguem mudanças com mais confiança e rapidez.

5. **Consistência dos testes [15]:** Os testes automáticos são altamente consistentes na sua execução, reduzindo a chance de erros ou desvios humanos. Este KPI mede a confiabilidade e a repetibilidade de testes automáticos comparando os resultados de várias execuções de testes. Os resultados consistentes aumentam a confiança na estabilidade do software.

6. **Utilização de recursos [15]:** Mede até que ponto os testes automáticos reduzem o esforço manual exigido dos *testers*. Ao automatizar tarefas realizadas diariamente e repetitivas, os *testers* podem concentrar-se em testes exploratórios (*exploratory testing*, em inglês), design de testes estratégicos e outras atividades de valor agregado que melhoram a qualidade geral do software.

6.2 Resultados dos KPIs

Sendo que os KPIs escolhidos já foram apresentados e descritos, agora serão analisados os resultados dos mesmos, comparando as métricas antes e após a implementação dos testes automáticos. Os resultados estão apresentados na Tabela 6.1, e oferecem uma visão detalhada sobre os impactos da automatização dos testes em diversos aspetos do processo de desenvolvimento.

KPI		Antes da automatização	Depois da automatização	Melhoria percentual
Cobertura dos testes		60%	85%	41.7%
Tempo de execução		20 horas	4 horas	80%
Esforço de manutenção	Reutilização do casos de teste	50%	80%	60%
	Custo dos testes	5.000€ por lançamento	3.000€ por lançamento	40%
	Tempo médio para atualizar um teste	5 horas por teste	2 horas por teste	60%
Eficiência dos testes de regressão	Frequência do teste de regressão	uma vez por lançamento	diariamente	aumento da frequência
	Tempo do ciclo de lançamento	8 semanas	6 semanas	25%
Consistência dos testes	Fuga de defeitos (*defect leakage*)	15%	5%	66.7%
	Tempo de deteção de defeitos	5 dias	1 dia	80%
Utilização de recursos	Tempo para testes exploratórios (*exploratory testing*)	5 horas por sprint	20 horas por sprint	30%
	Redução do esforço manual	30 horas por semana	10 horas por semana	66.7%
	Produtividade da equipa	70%	90%	28.6%

Tabela 6.1: Resultado dos KPIs selecionados

Como é possível observar na tabela, a **cobertura dos testes** aumentou significativamente após a automatização. Inicialmente, a cobertura era de 60%, indicando que apenas 60% das funcionalidades do software eram testadas. Com a implementação dos testes automáticos, essa cobertura subiu para 85%, resultando numa melhoria percentual de 41,7%. Essa melhoria pode ser atribuída à capacidade dos testes automáticos de abrangerem um número maior de cenários, incluindo aqueles que seriam impraticáveis ou extremamente demorados para testes manuais. Uma cobertura mais ampla é crucial para identificar e corrigir falhas de maneira mais eficaz, assegurando uma maior qualidade do software.

Além disso, o **tempo necessário para executar o conjunto de testes** foi reduzido de 20 horas para 4 horas, marcando uma diminuição de 80%. Esse resultado evidencia a eficiência dos testes automáticos em comparação com os manuais, pois estes são executados mais rapidamente e podem ser realizados em paralelo, reduzindo significativamente o tempo total de execução. A redução de tempo permite que os testes sejam realizados com maior frequência, oferecendo *feedback* mais rápido e facilitando a deteção precoce de problemas.

A **manutenção dos testes automáticos** também apresentou melhorias notáveis em vários aspetos. A **reutilização dos casos de teste** aumentou de 50% para 80%, refletindo um crescimento de 60%. A automatização facilita a reutilização dos testes, diminuindo a necessidade de criar novos testes para cada versão do projeto e tornando o processo de manutenção mais ágil. Ademais, os **custos de manutenção dos testes** diminuíram de 5.000€ por lançamento para 3.000€ por lançamento, representando uma redução de 40%. Essa diminuição deve-se à menor necessidade de trabalho manual contínuo, resultando em despesas operacionais menores. O **tempo médio para atualizar um teste** também desceu de 5 horas para 2 horas, mostrando uma melhoria de 60%. A facilidade de modificar e atualizar os testes contribui para um processo de manutenção mais eficiente e menos demorado.

No que diz respeito à **eficiência dos testes de regressão**, houve uma melhoria significativa. Anteriormente, os testes de regressão eram realizados apenas uma vez por lançamento, mas agora

são executados diariamente. Esse aumento na frequência dos testes permite identificar regressões e problemas causados por alterações recentes de forma mais rápida. O **tempo necessário para o ciclo de lançamento** também foi reduzido de 8 semanas para 6 semanas, uma melhoria de 25%. Isso deve-se à capacidade dos testes automáticos de realizar testes de regressão de maneira mais eficiente, contribuindo para ciclos de desenvolvimento mais curtos e para uma entrega mais ágil das novas funcionalidades.

Outra área de melhoria significativa foi a **consistência dos testes**. A **taxa de defeitos que passaram para a produção** foi reduzida de 15% para 5%, resultando numa melhoria de 66,7%. Essa redução pode ser atribuída à execução mais consistente dos testes, que minimiza erros humanos e aumenta a confiabilidade dos resultados. Além disso, o **tempo médio para detetar defeitos** diminuiu de 5 dias para 1 dia, uma redução de 80%, dado que a automatização permite uma deteção mais rápida de problemas, possibilitando uma correção mais ágil e reduzindo o impacto dos defeitos na produção.

Por fim, a **utilização de recursos** também apresentou melhorias significativas. O **tempo dedicado a testes exploratórios** aumentou de 5 horas para 20 horas, o que pode parecer contra-intuitivo, mas reflete uma maior dedicação a testes exploratórios em vez de testes repetitivos e manuais, que foram automatizados. Além disso, o **esforço manual** foi reduzido de 30 horas por semana para 10 horas por semana, uma diminuição de 66,7%. Essa redução no esforço manual permite que a equipa se concentre em atividades mais estratégicas e de maior valor agregado. Ademais, a **produtividade da equipa** aumentou de 70% para 90%, o que representa um crescimento de 28,6%. A automatização dos testes proporciona à equipa o tempo necessário para se focarem em tarefas mais complexas e produtivas, resultando numa melhoria geral na produtividade.

6.3 Automatização e a sua Importância

Como já referido, o teste é uma etapa muito importante no ciclo de vida de desenvolvimento de software, visto que uma boa amostra de testes garante que a maioria dos erros é encontrada, no entanto, um dos princípios na realização de testes é que nem todos os erros podem ser encontrados. Assim, os especialistas na área acreditam que o produto final funciona como deveria, se estiver o mais próximo possível de uma determinada funcionalidade, confiabilidade ou desempenho [41].

A grande maioria dos processos de teste conduz ao processo de automatização dos mesmos e atualmente, quase todas as empresas desejam implementar essa funcionalidade dos seus projetos de desenvolvimento de software [60, 41].

O processo de automatização ocorre tanto durante a fase de implementação como durante a fase de teste e consiste no uso de um software específico para realizar o processo de teste. Este realiza a comparação dos resultados reais com os resultados esperados, reduzindo também a necessidade dos testes manuais, que podem ser consideravelmente trabalhosos, através da economização de tempo e da exposição de uma grande quantidade de erros e defeitos, que não poderiam ser reconhecidos por meio do processo tradicional [20].

Em mais detalhe, e como base nos resultados descritos anteriormente, pode-se dizer que esta téc-

nica é eficaz nos seguintes aspetos [20, 41]:

- **Testes de regressão**: Os testes de regressão verificam se um software funciona corretamente após a correção de *bugs* ou erros e são um dos tipos de teste que requer mais tempo quando realizados manualmente, assim a automatização torna-se bastante comum nestes casos;
- **Velocidade de execução**: A automatização ajuda a fornecer um *feedback* rápido à equipa de desenvolvimento, pois apesar dos *scripts* de verificação automatizados demorarem algum tempo para serem executados, estes continuam a ser mais rápidos que o processo de verificações manual;
- **Economização de tempo (dos *testers*)**: As verificações automatizadas podem ser iniciadas manualmente ou automaticamente, recorrendo a alguma ou nenhuma supervisão por parte da equipa de testes. Por isso, a automatização de testes economiza o tempo dos *testers*, fazendo com que estes possam-se focar em explorar novos recursos;
- **Toda a equipa pode produzir testes automáticos**: Os testes automáticos geralmente são escritos na mesma linguagem que o software que está a ser testado. Assim, a responsabilidade de escrever, conduzir e realizar testes torna-se uma responsabilidade compartilhada entre toda a equipa de desenvolvimento, fazendo com que todos os membros, não apenas os *testers*, possam contribuir para a qualidade do software.

No entanto, a eficiência dos testes automáticos pode ser comprometida em vários cenários. Nas **fases iniciais de um projeto** caracterizado por uma evolução rápida de software, investir fortemente na automatização pode levar a um ciclo contínuo de atualizações de *scripts*, criando uma carga de manutenção substancial. Adicionalmente, quando a interface do utilizador de uma aplicação sofre alterações frequentes, os *scripts* automáticos vinculados a elementos específicos da interface do utilizador podem tornar-se frágeis e exigir ajustes contínuos, diminuindo os benefícios da automatização.
Projetos de curta duração, com ciclos de desenvolvimento rápidos ou implementações únicas, podem não justificar o tempo e os recursos necessários para o desenvolvimento e manutenção de *scripts* automatizados.
Além disso, **a automatização não é adequada para testes exploratórios**, pois estes dependem da intuição e da criatividade dos *testers* para descobrir problemas imprevistos. Os testes automáticos seguem etapas predefinidas, o que pode levar a uma potencial perda destes cenários inesperados.
Nos casos em que um **software é instável**, a automatização pode ser ineficiente, uma vez que os testes podem falhar devido a fatores externos, como o tempo de inatividade do servidor ou a interrupções na rede, e não a defeitos reais de software.
Cenários de teste complexos apresentam outro desafio para a automatização. Testes altamente complexos que envolvem uma lógica complexa ou interações de dados complexas podem resultar em esforços de criação de *scripts* demorados e num risco elevado de erros no processo de automatização.

Orçamentos e recursos limitados também podem impactar a viabilidade da automatização, isto é, o investimento inicial necessário para a seleção de ferramentas, desenvolvimento de *scripts* e manutenção contínua pode superar os benefícios, especialmente quando os recursos são limitados. A **falta de pessoal qualificado** tanto em práticas de teste como em ferramentas de automatização pode prejudicar os esforços da automatização. Os *scripts* e a manutenção inadequados podem levar a resultados abaixo da média.
Da mesma forma, para **testes realizados com pouca frequência ou apenas uma vez**, o esforço investido na criação e na manutenção de *scripts* de automatização pode não proporcionar um retorno substancial do investimento. **Softwares fortemente dependentes de integrações de terceiros** também podem representar desafios para a automatização, uma vez que estas integrações estão sujeitas a fatores externos fora do controlo do processo de teste.
Por último, **tentar automatizar todos os testes**, incluindo aqueles que esporadicamente são alterados, pode levar a custos desnecessários de manutenção, sendo mais pragmático concentrar os esforços de automatização em testes críticos e executados com frequência.

Assim, encontrar um equilíbrio certo entre testes manuais e testes automáticos é essencial, e a decisão de automatizar deve ser baseada em fatores como a complexidade do projeto, a estabilidade do software, os recursos disponíveis e as implicações de manutenção a longo prazo.

Capítulo 7

Conclusão

O uso da *framework* Specflow, em conjunto com o paradigma BDD, para criar testes automáticos para um projeto organizacional de grande escala melhora e impulsiona significativamente o processo de desenvolvimento de software e garante de qualidade do mesmo. O estudo realizado neste trabalho de projeto explicou os diversos métodos, *framework* e software utilizados no processo de elaboração dos testes, descrevendo em detalhe todas as suas fases. Ainda, as descobertas apresentadas ilustram como esta abordagem efetivamente melhora a qualidade do software, promove a colaboração entre a equipa e aumenta a eficiência dos testes, bem como a aplicação dos mesmos num projeto industrial.

A implementação do BDD melhorou significativamente a comunicação e o alinhamento entre todos os membros da equipa (técnicos e não técnicos), pois utilizar a linguagem Gherkin para formular cenários de teste permitiu que todos pudessem entender e contribuir para refinar os requisitos. Essa abordagem colaborativa garantiu uma compreensão mais profunda das *user stories* e dos critérios de aceitação, alinhando o produto final de perto com os objetivos de negócios.

O SpecFlow provou ser fundamental na automatização de testes dentro do ambiente .NET, integrando-se perfeitamente com o Visual Studio para converter *scenarios* Gherkin em casos de teste executáveis. A automatização reduziu significativamente a duração do desenvolvimento e execução dos testes, transformando o que antes era um processo trabalhoso numa prática simplificada e repetível, o que economizou tempo e esforço consideráveis.

Adicionalmente, a escalabilidade e a manutenção foram considerações críticas durante todo o processo de teste. O design modular do SpecFlow facilitou a organização dos testes e dos *step definitions*, acomodando a complexidade de um projeto de grande escala. As práticas de integração contínua foram efetivamente suportadas, garantindo que os testes eram executados rotineiramente e perfeitamente integrados ao ciclo de desenvolvimento. A deteção precoce de defeitos por meio de testes automáticos minimizou os custos e interrupções associados à correção de bugs, contribuindo para a estabilidade do projeto.

No entanto, integrar o BDD num workflow de desenvolvimento já existente revelou-se árduo devido a complexidade do projeto, e garantir o desempenho do *test suite* foi um esforço contínuo, pois os testes foram otimizados e paralelizados para reduzir o tempo de execução, e melhorias de infraestrutura foram feitas para suportar simultaneidade.

Embora os resultados do trabalho de projeto sejam promissores, existem várias áreas identificadas para possíveis **trabalhos futuros**. Primeiramente, os esforços devem concentrar-se na expansão da cobertura de testes automáticos para abranger cenários de teste mais complexos e casos extremos, reforçando assim a robustez do processo de teste. Além disso, explorar a integração com outras ferramentas e frameworks de teste, como Selenium para testes de UI ou JMeter para testes de desempenho, pode oferecer uma solução de teste mais abrangente. Adicionalmente, desenvolver capacidade avançadas de *reporting* e análise, de modo a obter *insights* mais profundos sobre resultados de testes, tendências e integridade do sistema, pode ser vantajoso para a melhoria contínua. Por fim, continuar a investir numa aprendizagem constante e cultivar uma cultura de qualidade e colaboração será crucial para sustentar o sucesso da abordagem BDD.

Bibliografia

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2016.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[4] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 1st edition, 2002.

[5] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development. https://agilemanifesto.org/iso/en/manifesto.html, 2001. Acedido em: 20 de agosto, 2024.

[6] International Software Testing Qualifications Board. Certified tester foundation level syllabus. https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CTFL_Syllabus_2018_v3.1.1.pdf, 2018. Acedido em: 20 de agosto, 2024.

[7] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Transactions on Software Engineering*, 21(5):61–72, 1988.

[8] I. Burnstein. *Practical Software Testing: A Process-Oriented Approach*. Springer Professional Computing. Springer New York, 2003.

[9] M. G. Cavalcante and J. I. Sales. The behavior driven development applied to the software quality test: A case study applied to the promotion of sports financing in brazil. In *Proceedings of the 14th Iberian Conference on Information Systems and Technologies (CISTI 2019)*, pages 1–4. IEEE, 2019.

[10] CGI. It and business consulting services | cgi.com. https://www.cgi.com/en, 2024. Acedido em: 20 de agosto, 2024.

[11] L. A. Cisneros, C. I. Reis, M. Maximiano, and J. A. Quiña. An experimental evaluation of itl, tdd, and bdd. In *Proceedings of the Thirteenth International Conference on Software Engineering Advances (ICSEA 2018)*, pages 20–24. IARIA, 2018.

[12] M. Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 1st edition, 2004.

[13] S. Desikan and G. Ramesh. *Software Testing: Principles and Practice*. Pearson Education India, 1st edition, 2006.

[14] E. Dustin, T. Garrett, and B. Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Pearson Education, 1st edition, 2009.

[15] M. Fewster and D. Graham. *Software Test Automation*. Addison-Wesley, 1st edition, 1999.

[16] Martin Fowler. Business readable dsl. https://martinfowler.com/bliki/BusinessReadableDSL.html, 2008. Acedido em: 20 de agosto, 2024.

[17] G. Gurung, R. Shah, and D. Jaiswal. Software development life cycle models: A comparative study. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pages 30–37, 7 2020.

[18] L. A. C. Gómez. Analysis of the impact of test-based development techniques (tdd, bdd, and atdd) on the software life cycle. Master's thesis, Instituto Politécnico de Leiria - Escola Superior de Tecnologia e Gestão, 2018.

[19] M. L. Hutcheson. *Software Testing Fundamentals: Methods and Metrics*. John Wiley & Sons, 2003.

[20] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad. Software testing techniques: A literature review. In *Proceedings of the 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pages 177–182. IEEE, 2016.

[21] JBehave. What is jbehave. https://jbehave.org/, 2022. Acedido em: 20 de agosto, 2024.

[22] P. Jorgensen and B. DeVries. *Software Testing: A Craftsman's Approach*. Auerbach Publications, 5th edition, 2022.

[23] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing: A Context-Driven Approach*. John Wiley & Sons, 1st edition, 2011.

[24] KPI.org. What is a key performance indicator (kpi)? https://www.kpi.org/kpi-basics/, 2024. Acedido em: 20 de agosto, 2024.

[25] M. Krief. *Learning DevOps: The Complete Guide to Accelerate Collaboration with Jenkins, Kubernetes, Terraform, and Azure DevOps*. Packt Publishing, 2019.

[26] W. E. Lewis and G. Veerapillai. *Software Testing and Continuous Quality Improvement*. Auerbach Publications, 2nd edition, 2004.

[27] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 1st edition, 2013.

[28] A. P. Mathur. *Foundations of Software Testing: Fundamental Algorithms and Techniques*. Addison-Wesley Professional, 2nd edition, 2012.

[29] R. McLeod Jr. and G. Everett. *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley-IEEE Computer Society Press, 1st edition, 2007.

[30] Microsoft. Azure artifacts overview - azure artifacts | microsoft learn. https://learn.microsoft.com/en-us/azure/devops/artifacts/start-using-azure-artifacts, 2024. Acedido em: 20 de agosto, 2024.

[31] Microsoft. Collaborate on code - azure repos | microsoft learn. https://learn.microsoft.com/en-us/azure/devops/repos/get-started/what-is-repos, 2024. Acedido em: 20 de agosto, 2024.

[32] Microsoft. What is azure boards? - azure boards | microsoft learn. https://learn.microsoft.com/en-us/azure/devops/boards/get-started/what-is-azure-boards, 2024. Acedido em: 20 de agosto, 2024.

[33] Microsoft. What is azure devops? - azure devops | microsoft learn. https://learn.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops, 2024. Acedido em: 20 de agosto, 2024.

[34] Microsoft. What is azure pipelines? - azure pipelines | microsoft learn. https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines, 2024. Acedido em: 20 de agosto, 2024.

[35] Microsoft. What is azure test plans? manual, exploratory, and automated test tools - azure test plans. https://learn.microsoft.com/en-us/azure/devops/test/overview?view=azure-devops, 2024. Acedido em: 20 de agosto, 2024.

[36] Microsoft. What is scrum? - azure devops | microsoft learn. https://learn.microsoft.com/en-us/devops/plan/what-is-scrum, 2024. Acedido em: 20 de agosto, 2024.

[37] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2nd edition, 2004.

[38] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, Inc., 3rd edition, 2011.

[39] P Nikhil. *Learning Continuous Integration with Jenkins: A Beginner's Guide to Implementing Continuous Integration and Continuous Delivery Using Jenkins 2*. Packt Publishing Ltd, 2nd edition, 2017.

[40] D. North. Introducing bdd. https://dannorth.net/introducing-bdd/, 2022. Acedido em: 20 de agosto, 2024.

[41] B. Oliinyk and V. P. Oleksiuk. Automation in software testing: Can we automate anything we want? https://api.semanticscholar.org/CorpusID:211242106, 2019. Acedido em: 20 de agosto, 2024.

[42] R. W. Pate II. Behavior driven development kick-start. Master's thesis, The University of Texas at Austin, 2021.

[43] R. S. Pressman and B. Maxim. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 8th edition, 2014.

[44] RSpec. Rspec - behaviour driven development for ruby. https://rspec.info/, 2022. Acedido em: 20 de agosto, 2024.

[45] Z. Seremet and K. Rakic. Best approach to security in azure devops. *DAAAM International Scientific Book*, 2021.

[46] M. Shahin, M. A. Babar, and L. Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access*, 5:3909–3943, 2017.

[47] R. Sharda, D. Delen, and E. Turban. *Business Intelligence, Analytics, and Data Science: A Managerial Perspective*. Pearson, 4th edition, 2018.

[48] R. M. Sharma. Quantitative analysis of automation and manual testing. *International Journal of Engineering and Innovative Technology (IJEIT)*, 4(1), 2014.

[49] I. Shubinsky and H. Schabe. Errors, faults and failures. *Dependability*, 21(2):24–27, 2021.

[50] Dr. S. K. Singh and Dr. A Singh. *Software Testing*. Vandana Publications, 1st edition, 2019.

[51] Perforce Software. Quantum. https://www.perfecto.io/integrations/quantum, 2022. Acedido em: 20 de agosto, 2024.

[52] SmartBear Software. Gherkin reference. https://cucumber.io/docs/gherkin/reference/, 2019. Acedido em: 20 de agosto, 2024.

[53] SmartBear Software. Bdd testing & collaboration tools for teams | cucumber. https://cucumber.io/, 2022. Acedido em: 20 de agosto, 2024.

[54] Spec Solutions. Specsync. https://www.specsolutions.eu/specsync/. Acedido em: 20 de agosto, 2024.

[55] I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011.

[56] Tricentis. Bdd framework for .net - specflow - enhance your automated tests. https://specflow.org/, 2024. Acedido em: 20 de agosto, 2024.

[57] Tricentis. Bdd helps you find bugs before they find you. start learning bdd! https://specflow.org/learn/bdd/, 2024. Acedido em: 20 de agosto, 2024.

[58] Tricentis. Gherkin language - use your language to describe test cases. https://specflow.org/learn/gherkin/, 2024. Acedido em: 20 de agosto, 2024.

[59] Tricentis. Welcome to specflow's documentation! https://docs.specflow.org/projects/specflow/en/latest/, 2024. Acedido em: 20 de agosto, 2024.

[60] S. I. M.l Trindade. Caso de estudo sobre automação de testes de software. Master's thesis, Instituto Politécnico de Viseu, 2021.

[61] M. A. Umar. Comprehensive study of software testing: Categories, levels, techniques, and types. *TechRxiv*, June 2020.

[62] J. Virtanen. Comparing different ci/cd pipelines. *Journal of Software Engineering*, 15(4):234–245, 2021.

[63] J. A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70–79, 2000.

[64] F. Zampetti, S. Garzó, G. Bavota, and M. Di Penta. Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 471–482. IEEE, 2021.