Instituto Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Sofia Inês Morais Trindade

# Caso de estudo sobre automação de testes de software

Outubro de 2021

Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu

Sofia Inês Morais Trindade

# Caso de estudo sobre automação de testes de software

**Tese de Mestrado**
Sistemas e Tecnologias da Informação para as Organizações

Professora Doutora Ana Cristina Wanzeller Guedes de Lacerda
Professor Doutor Carlos Augusto da Silva Cunha

Outubro de 2021

*"A primeira regra de qualquer tecnologia utilizada nos negócios é que a automação aplicada a uma operação eficiente aumentará a eficiência. A segunda é que a automação aplicada a uma operação ineficiente aumentará a ineficiência."*

**Bill Gates**

# RESUMO

A qualidade de um sistema, atualmente é algo imprescindível para os utilizadores. Sem o mínimo de qualidade o produto pode não chegar ao consumidor final. Para assegurar a qualidade é necessário efetuar uma quantidade exaustiva de testes durante as fases do seu desenvolvimento de onde podem surgir defeitos. Quanto mais cedo os defeitos forem detetados menor é o seu custo de resolução. Atualmente, existe também a necessidade de validar os sistemas de forma mais rápida e segura devido ao conceito de entrega e integração contínua, cujo objetivo é colocar qualquer tipo de alteração no ambiente de produção. Neste sentido, recorre-se a estratégias de automação de testes para otimizar o seu processo de entrega.

Com a realização deste trabalho pretende-se aprofundar os fundamentos teóricos sobre a área da qualidade, mas também, propor uma estratégia de automação de testes de software, tendo em conta uma arquitetura básica e passível de aplicação na maioria de sistemas que a usem. Esta estratégia é aplicada num caso de estudo, tendo em conta as boas práticas e diretrizes do processo de teste: planeamento, controlo e monitorização, análise, desenho, implementação, execução e conclusão. A arquitetura adotada pela estratégia de testes proposta é baseada no padrão arquitetural de três camadas. O âmbito da estratégia passa pela implementação de testes de integração nos serviços da camada aplicacional, onde se validam as regras de negócio, o corpo e o código das respostas recebidas. Inclui ainda a implementação de testes *End-to-End* na camada de apresentação, onde se validam interações entre os componentes e as interfaces da aplicação. Todos os testes automatizados são executados numa *pipeline* de integração e entrega contínua, sempre que pretenda enviar qualquer alteração para o ambiente produtivo.

A aplicação da estratégia a um caso de uso permitiu concluir que a mesma é passível de utilização em projetos de várias dimensões, visto que são abordados os aspetos mais pertinentes da definição estratégias de teste. Qualquer pessoa que tenha interesse na área de automação de testes consegue obter as bases necessárias para proceder á implementação de testes automáticos, criar baterias de testes de regressão automáticas e investir o restante tempo em testes exploratórios manuais, cujo foco são situações nunca antes pensadas e de onde podem surgir problemas ainda não identificados, evitando que estes cheguem ao consumidor final.

# ABSTRACT

The quality of a system nowadays is something essential for users. Without a minimum of quality, the product may not reach the final consumer. To ensure quality it is necessary to carry out an exhaustive amount of tests during all phases of the development where defects may arise. The sooner defects are detected, the lower their resolution cost. In addition to this, there is also a need to validate systems faster and more securely due to the concept of continuous integration and delivery, whose objective is to place any type of change in the production environment. That way, test automation strategies are used to optimize the delivery process.

The purpose of this work is to deepen the theoretical foundations of the quality area, but also to propose a software test automation strategy, considering a basic architecture that can be applied in most systems that may use it. This strategy will be applied in a case study, considering the best practices and guidelines of the test process: planning, control and monitoring, analysis, design, implementation, execution, and conclusion. The architecture adopted by the proposed testing strategy is based on the three-tier architectural pattern. The scope of the strategy involves the implementation of integration tests in the application layer of the services, where the business rules, the body and the code of the responses received are validated. It also includes the implementation of end-to-end tests in the presentation layer, where interactions between application components and interfaces are validated. All automated tests run in a pipeline of continuous integration and delivery, whenever you want to send any changes to the production environment.

The application of the strategy to an use case allowed us to conclude that it can be used in projects of various dimensions, since the most pertinent aspects of defining test strategies are addressed. Anyone interested in the test automation area can obtain the necessary bases to implement automatic tests, create batteries of automated regression tests and invest the remaining time in manual exploratory tests, whose focus is situations never thought before and from where problems not yet unidentified may arise, preventing them from reaching the end users.

## PALAVRAS CHAVE

arquitetura de software
automatização de testes
estratégias de testes
entrega contínua
integração contínua
qualidade de software
tipos e níveis de testes

## KEY WORDS

automatic tests
continuous integration
continuous delivery
quality of software
software architecture
test strategies
types and levels of tests

# AGRADECIMENTOS

O desempenho no desenvolvimento desta dissertação não teria sido o mesmo sem a ajuda e apoio de algumas pessoas, às quais gostaria de aqui expressar o meu reconhecimento.

Aos meus pais, pelo incentivo, dedicação e confiança transmitidos na concretização deste projeto. A eles, agradeço-lhes os valores e educação que me transmitiram ao longo destes anos.

Aos meus orientadores, Prof. Dra. Cristina Wanzeller e Prof. Dr. Carlos Cunha, pela disponibilidade e dedicação neste projeto. Agradeço-lhes as respostas de todas as minhas dúvidas, o tempo despendido na leitura e retificação do mesmo e a contribuição para que o mesmo fosse possível.

Um agradecimento especial ao Prof. Dr. Paulo Tomé, diretor do mestrado, pelo apoio para que fosse possível a entrega deste projeto.

A todos, até mesmo os que não mencionei, um muito obrigado!

# ÍNDICE GERAL

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLEAS

# ABREVIATURAS E SIGLAS

| | |
|---|---|
| API | *Application Programming Interface* |
| BD | Base de Dados |
| BDD | *Behavior Driven Development* - Desenvolvimento Orientado por Comportamento |
| CD | *Continuous Delivery and/or Continuous Deployment* – Entrega Contínua e/ou Desenvolvimento Contínuo |
| CI | *Continuos Integration* – Integração Contínua |
| CSS | *Cascading Style Sheets* - Folha de Estilo em Cascatas |
| DEVs | *Devlopers* - Programadores |
| E2E | *End-To-End* |
| ESTGV | Escola Superior de Tecnologia e Gestão de Viseu |
| HTLM | *HyperText Markup Language* - Linguagem de Marcação de Hipertexto |
| HTTP | *Hypertext Transfer Protocol* - Protocolo de transferência de hipertexto/hipermédia |
| IP | *Internet Protocol* – Protocolo de Internet |
| IPV | Instituto Politécnico de Viseu |
| JS | *JavaScript* |
| JSON | *JavaScript Object Notation* - Notação de Objetos JavaScript |
| NPM | *Node Package Manager* |
| OAT | *Operational Acceptance Testing* |
| REST | *Representational State Transfer* - Transferência Representacional de Estado |
| STLC | *Software Testing Life Cicle* – Ciclo de Vida de Testes de Software |
| TCP | *Transmission Control Protocol* - Protocolo de Controlo de Transmissão |
| TDD | *Test Driven Development* - Desenvolvimento Orientado por Testes |
| UAT | *User Acceptance Testing* |
| UI | *User Interface* |
| URL | *Uniform Resource Locator* |
| US | *User Story* |

# 1. Introdução

Na sociedade atual, a tecnologia é omnipresente e quase omnipotente. As pessoas utilizam-na para quase tudo o que precisam de fazer no seu dia-a-dia. Esta dependência tecnológica necessita de uma enorme agilidade para constantes mudanças. Tal implica que o software esteja correto, funcional e pronto a usar.

A complexidade do software também está a crescer e, de modo a verificar todos os seus componentes e funcionalidades, é necessário efetuar uma quantidade exaustiva de testes durante o seu desenvolvimento [1].

Os testes de software são uma atividade que visa contribuir para a melhoria da qualidade do software. Segundo Emerson Rios na década de 70, os testes de software eram elaborados e executados pelos próprios programadores ou analistas de sistemas. Mas, nos anos seguintes, os utilizadores passaram a determinar quando um sistema poderia estar no ambiente de produção dependendo da quantidade problemas conhecidos que existiam no sistema [46].

O primeiro *bug* em computadores possivelmente ocorreu em 1947 [47], porque segundo os Engenheiros do Harvard Mark I, no primeiro computador digital automático de larga escala, foi encontrada uma traça nos seus circuitos que causou erros nos cálculos da máquina. Então eles prenderam-na e chamaram-lhe como o "primeiro bug" encontrado [45]. O termo bug – palavra inglesa que significa "inseto" – tem sido utilizado, desde então, pelos engenheiros para classificar falhas/defeitos nas máquinas [47].

A realização de testes de software é uma tarefa complexa, mas fundamental, pois, caso sejam encontradas as falhas antes do consumidor final utilizar o software, consegue-se poupar o seu tempo de resolução e, por consequência, reduzir custos.
O principal objetivo da atividade de teste é verificar que o comportamento do software é o expectável sem comportamentos indesejáveis, dentro de um conjunto de cenários possíveis. Esta verificação é muito importante para garantir a qualidade do software desenvolvido e reduzir, assim, a ocorrência de erros ao longo das várias fases do desenvolvimento.
Atualmente, as empresas de software precisam de validar os seus produtos de forma mais rápida. As aplicações passaram a ter ciclos mais curtos devido ao conceito de entrega contínua (*Continuous Delivery - CD*), cujo objetivo é colocar qualquer tipo de alteração (por exemplo, novas funcionalidades, falhas, configurações, entre outras) no ambiente de produção de forma segura, rápida e sustentável [34]. Deste modo, as empresas começaram a recorrer a estratégias de automação para tentar reduzir o tempo de execução de testes que eram feitos de forma manual.

## 1.1 Motivação e objetivos

Atualmente, a autora deste trabalho encontra-se a trabalhar na área da qualidade de software, tendo iniciado esta atividade há cerca de quatro anos. Inicialmente começou pelos casos de teste e execução de testes manuais. Posteriormente, iniciou pequenas experiências com testes automatizados a interfaces de aplicações. Atualmente, é responsável pelo planeamento e implementação de estratégias de testes a partir da arquitetura do software.
Pelo facto de querer evoluir na área da automatização, decidiu, no âmbito da unidade curricular de Introdução à Dissertação/Projeto/Estágio do Mestrado em Sistemas e Tecnologias da Informação para as Organizações, conceber uma dissertação acerca de um "Caso de estudo de automação de testes de software".
Durante a sua experiência profissional, a autora teve contacto com alguns profissionais da área e apercebeu-se que a evolução da realização de testes manuais para testes automatizados era uma dificuldade. Com a realização deste trabalho, pretende-se contribuir para que qualquer pessoa que tenha interesse na área da qualidade consiga obter as bases necessárias dos fundamentos teóricos e práticos de todo o processo de testes de software, tendo em conta uma

arquitetura básica de um sistema que possibilita a aplicação da estratégia que irá ser proposta na generalidade dos sistemas.

Através da automatização é possível conduzir a atividade de testes de software de forma mais expedita. Deste modo, este trabalho tem os seguintes objetivos:

- Efetuar uma pesquisa bibliográfica para identificação do estado da arte do tema dos testes de software, visando a fundamentação teórica do trabalho a realizar e para justificar limites e contribuições da revisão que será realizada;
- Proporcionar um resumo sistemático do tema dos testes de software que possa contribuir para a aquisição rápida de conhecimentos acerca deste tema por pessoas que o desconhecem;
- Propor uma estratégia de automação de testes de software tendo em conta uma arquitetura básica e passível de aplicação na maioria de sistemas que a usem.

## 1.2 Metodologia de investigação

Como metodologia de investigação vai-se usar a técnica dos "seis P", proposta por Oates [48], [49]. Esta técnica é composta pela definição das seguintes etapas: Propósito, Produtos, Processo, Participantes, Paradigma, Apresentação. A técnica será aplicada nesta dissertação do seguinte modo:

- **Propósito** - Estratégia de automação de testes de software.
- **Produtos** - Implementação de uma estratégia de automação de testes, tendo em conta uma arquitetura de uma aplicação de software e um caso de uso do sistema.
- **Processo** - vai-se ter em conta as seguintes etapas:
    a) Rever a literatura na área de testes de software;
    b) Investigar acerca de ferramentas e *frameworks open source* utilizadas para a realização de cada tipo ou nível de testes;
    c) Definir uma arquitetura para a demonstração de uma estratégia de automação de testes para cada nível de testes;
    d) Definir o caso de uso para a exposição de exemplos práticos de cada tipo de teste;

e) Para cada nível de teste, demonstrar como se procede à instalação das ferramentas/*frameworks* necessárias, como é feita a escrita dos testes e como se executam os mesmos.

- **Participantes** - No decorrer da investigação ter-se-á a colaboração de diversos agentes, colegas de trabalho, investigação e orientadores.
- **Paradigma** - Vai-se usar como paradigma o interpretativismo [51], uma vez que se pretende analisar diversos processos para depois se proceder à definição/implementação de um.
- **Apresentação** - Os resultados da investigação serão apresentados na forma de uma dissertação.

## 1.3 Estrutura do documento

O presente documento está organizado em seis capítulos, incluindo o presente capítulo de Introdução. Os restantes capítulos encontram-se estruturados da seguinte forma.

O capítulo 2, Fundamentos no âmbito dos testes, apresenta o estudo que aborda os vários conceitos teóricos sobre testes de software.

O capítulo 3, Padrões ideais para a definição testes de software, apresenta padrões e diretrizes a seguir para a escrita automatizada dos casos de teste.

O capítulo 4, Estratégia de testes proposta com base numa arquitetura de software, apresenta e descreve a arquitetura base de um sistema, bem como o planeamento da estratégia de testes, de acordo com as boas práticas e diretrizes do processo de testes.

O capítulo 5, Caso de estudo, descreve um caso de uso para o qual vai ser aplicada a estratégia delineada no capítulo anterior.

E finalmente, o capítulo 6, Conclusão, apresenta conclusões e limitações tiradas da realização deste trabalho e serão também sugeridos caminhos futuros para a continuação deste trabalho.

# 2. Fundamentos de testes de software

A exigência de elevados padrões de qualidade tem motivado a definição de novos métodos e técnicas a aplicar durante as fases de desenvolvimento de software. Consequentemente, o interesse pela atividade de testes tem vindo a aumentar nos últimos anos [7]. Uma das principais causas é o processo de *Continuous Delivery*, motivado pelas adaptações constantes ao sistema num curto período de tempo, garantindo que todos os seus componentes se mantêm como expectável. Para o efeito, passou-se a recorrer à automatização de testes, com o propósito de acompanhar de forma eficiente a entrega contínua.

Este capítulo apresenta uma revisão da literatura atual na área de testes de software. Primeiramente, aborda a importância de se investir em testes e qualidade. Seguidamente analisa o porquê de existirem defeitos ou falhas no software. Posteriormente apresenta-se o processo de testes e conceitos técnicos sobre os mesmos, como, os princípios e os conceitos utilizados na realização de testes, e os diferentes tipos e níveis de testes de software. Finalmente, explica-se a diferença entre testes manuais e testes automatizados, tendo em conta alguns aspetos que devem estar presentes antes da realização da automação.

## 2.1 Importância da qualidade de software

Hoje em dia, a garantia da qualidade de software tem mais importância, porque a evolução tecnológica trouxe mais complexidade para os sistemas. A adoção de novos métodos e padrões

de desenvolvimento tem sido uma necessidade constante para conseguir entregar e melhorar o software o mais rápido possível.
Os testes ajudam na obtenção de métricas, através dos erros encontrados nos sistemas, tanto ao nível de características funcionais e não funcionais ou até mesmo ao nível de requisitos [4].
Ao investir em testes, investimos em prevenção de falhas e defeitos no software entregue ao cliente. Ao fazermos isto, investimos também em garantias de que o produto que se quer entregar está de acordo com a especificação e necessidades do cliente. Por consequência objetivam-se os seguintes aspetos [8]:

- Maior satisfação do utilizador final;
- Melhoria da imagem da empresa;
- Maior redução das incertezas que rodeiam o software;
- Redução do custo de manutenção do produto entregue.

Nem sempre o desenvolvimento do software corre como estava planeado. Frequentemente, aparecem situações inesperadas. São exemplos situações como algum defeito no software, atraso nos prazos previstos de entrega por causa de défice de conhecimento/capacidade na equipa de desenvolvimento, entre outras. Estas situações podem ser utilizadas como conhecimento para evitar problemas semelhantes no futuro. Ao compreender as origens e causas dos defeitos encontrados, conseguimos que os processos possam ser melhorados, eliminando, assim, a recorrência desses defeitos e, por consequência, melhorando a qualidade dos futuros sistemas. Este é um aspeto fundamental da garantia de qualidade [4].

## 2.2 Causas de erros em software

Como errar é humano, qualquer ser humano pode cometer um erro (ou engano), mas esse erro pode produzir um defeito (ou *bug*) o que, por sua vez, pode dar origem a uma falha [4]. Por exemplo, um erro na perceção dos requisitos pode levar a um defeito nos requisitos, que resulta num erro de programação e leva a um defeito no código. Deste modo, os defeitos ocorrem normalmente, nos procedimentos mais estáticos, como por exemplo, no código de um programa ou na documentação/especificação técnica [4].
Se o defeito no código for executado, o sistema pode deixar de fazer o que era suposto ou fazer algo que não deveria, causando, assim, uma falha. Ou seja, as falhas estão associadas a aspetos mais dinâmicos onde haja execução de algo [4].

Os defeitos ocorrem principalmente devido a falhas humanas porque o ser humano está sujeito a vários fatores, como por exemplo, a pressão do cumprimento de *deadlines* (datas de entrega), a complexidade do código do programa/sistema/infraestrutura e a evolução tecnológica (que tem por consequência a curva de aprendizagem) entre outros [4].

As falhas, para além de serem causadas por defeitos no código, também podem ser causadas por questões ambientais, como, a radiação, o magnetismo, a sujidade, entre outras, podendo causar defeitos no *firmware*[1] ou influenciar a correta execução do software por causa de alguma mudança de condições do *hardware* [4].

## 2.3 O processo de testes

Todos os sistemas passam por um ciclo de desenvolvimento. O processo de testes em particular considera um conjunto de fases que devem ser efetuadas desde o início do desenvolvimento do sistema, tais como [15]:

- **Planeamento** - fase em que se define ou planeia os objetivos dos testes e quais as abordagens a seguir, tendo em conta o seu contexto;
- **Controlo e Monitorização** - Trata-se de uma fase contínua entre todas as outras, porque envolve a comparação entre o progresso atual e o progresso planeado ou esperado, usando métricas de monitorização definidas no plano de teste;
- **Análise** - nesta fase examina-se o que se vai testar e definem-se as condições de teste;
- **Desenho** - Envolve especificar como testar, transformando as condições de teste definidas na fase anterior, em casos de teste ou conjuntos de casos teste de alto nível;
- **Implementação** - De seguida, vai-se definir a sequência dos casos de teste num procedimento de teste e, também, definir todos os processos necessários para a execução dos testes;
- **Execução** – Durante esta fase, os conjuntos de testes são executados (manualmente ou recorrendo a ferramentas de execução de testes) de acordo com o seu planeamento. Faz-se a comparação dos resultados atuais com os resultados esperados, reporta-se os

[1] *Firmware* - refere-se ao software que foi instalado permanentemente numa máquina, dispositivo ou *microchip*, geralmente pelo fabricante. Tem como objetivo controlar o hardware em segundo plano e sem interação humana [102].

defeitos baseados nas falhas observadas e regista-se o resultado da execução dos testes (por exemplo, passou, falhou, bloqueado);

- **Conclusão** - As atividades de conclusão de testes guardam os dados e outras informações relevantes sobre as atividades de teste finalizadas para consolidar experiência.

## 2.4 Conceitos sobre testes de software

Nesta secção, pretende-se abordar os princípios que devem ser aplicados à realização de testes, dar a conhecer os vários conceitos técnicos utilizados no dia a dia no processo de testes de software e, de forma resumida, apresentar e descrever os diferentes tipos e níveis de testes.

### 2.4.1 Princípios na realização de testes

Ao longo dos últimos 40 anos, foi definido o seguinte conjunto de princípios de teste que oferecem linhas de orientação gerais e comuns a todos os tipos de testes [15]:

1) **Os testes mostram a presença de defeitos e não a sua ausência -** Os testes podem mostrar a presença de defeitos, mas não provam a sua inexistência. Os testes reduzem a probabilidade de os defeitos não detetados permanecerem no software.
2) **Testes exaustivos são impossíveis -** Testar tudo e considerar todas as combinações possíveis não é viável. Em vez de testes exaustivos, a análise de risco e as prioridades devem ser utilizadas para focar os esforços de teste.
3) **Testar cedo salva tempo e dinheiro -** Para tentar evitar que os defeitos sejam mantidos até uma fase de desenvolvimento mais avançada, as atividades de teste devem ser iniciadas o mais cedo possível no ciclo de desenvolvimento do software ou sistema e devem estar focadas nos objetivos definidos.
4) **Agrupamento de defeitos -** O esforço de testes deve ser proporcional à densidade de defeitos esperada por módulo. Normalmente, um número reduzido de módulos apresenta a maioria dos defeitos detetados, ou é responsável pela maioria das falhas. Portanto, deve-se perceber quais os módulos do sistema que são mais críticos ou propícios a defeitos ou falhas para se investir com mais rigor na qualidade desses módulos.

5) **Paradoxo do pesticida -** A repetição exaustiva dos mesmos casos de teste pode levar à não deteção de novos defeitos. Para superar este "paradoxo do pesticida", os casos de teste devem ser regularmente analisados, revistos e mudados para executar diferentes partes do software ou sistema.
6) **Os testes dependem do seu contexto -** Os testes são efetuados de forma diferente em diferentes contextos. Por exemplo, o software crítico em segurança (*safety-critical software*) é testado de forma diferente de um website de comércio eletrónico, pois exige diferentes tipos de cuidados.
7) **Falácia da ausência de erros -** Detetar e corrigir defeitos por si só não chega se o sistema construído não satisfizer as necessidades e expectativas dos seus utilizadores.

### 2.4.2 Níveis de testes de software

Níveis de testes referem-se a uma instanciação específica de um processo de teste. Tratam-se de grupos de atividades de testes que são organizados e geridos em conjunto, tendo estes uma orientação horizontal ou transversal. Na Tabela 2-1 aborda-se, de forma resumida, alguns tipos de testes de software [15] [16]:

Tabela 2-1 : Níveis de testes de software [15] [16]

| Níveis de Testes | | Descrição |
|---|---|---|
| **Testes de Unitários ou de Componente** | | O foco é testar os componentes isolados do sistema, ou seja, testa-se cada classe e cada componente do sistema isoladamente. |
| **Testes de Integração** | Sistemas | Os testes de integração de sistemas focam-se nas interações entre as classes, pacotes e microsserviços. Também se focam em interações providenciadas por organizações externas (por exemplo, *web services*[2]). |
| | Componentes | Os testes de integração de componentes focam-se nas interações entre as interfaces e os componentes. |
| **Testes de Sistema** | | Os testes de sistema focam-se no comportamento de um sistema ou produto, considerando as tarefas *end-to-end* que o sistema pode executar. |
| **Testes de Aceitação** | UAT | O *User Acceptance Testing* (UAT) geralmente foca-se na validação do uso adequado do sistema pelos utilizadores finais num ambiente real ou simulado. O |

[2] W*eb services* - são utilizados para transferir dados através de protocolos de comunicação para diferentes plataformas, independentemente das linguagens de programação utilizadas nessas plataformas [64].

| | | |
|---|---|---|
| Focam-se no comportamento e nas capacidades de um sistema ou produto como um todo | | principal objetivo é dar confiança aos utilizadores mostrando que conseguem usar o sistema para atender às suas necessidades. |
| | OAT | O *Operational Acceptance Testing* (OAT) é feito pela equipa de operações ou administração de sistemas, num ambiente de produção (simulado). O principal objetivo é criar a confiança de que os operadores ou administradores de sistemas conseguem manter o sistema a funcionar corretamente, mesmo sob condições excecionais. |
| | Alpha & Beta | *Alpha* - Teste realizado no ambiente de teste do programador, por pessoas externas à organização de desenvolvimento;<br>*Beta* - Teste realizado num site externo, por pessoas externas à organização de desenvolvimento. |
| | Contrato e Regulatória | Teste de Aceitação de Contrato é realizado para verificar se um sistema atende aos seus requisitos contratuais.<br>Teste de Aceitação de Regulatória é realizado para verificar se um sistema está em conformidade com as leis, políticas e regulamentos relevantes. |

### 2.4.3 Tipos de testes de software

Os tipos de testes são grupos de atividades de teste, definidas com base em objetivos específicos direcionados para características específicas de um componente ou sistema, tendo estes uma orientação vertical. Na Tabela 2-2 apresenta-se, de forma resumida, alguns tipos de testes de software [15] [16]*:*

Tabela 2-2 : Tipos de testes de software [15] [16]

| Tipos de Testes | Descrição |
|---|---|
| **Funcionais** | Testes realizados para avaliar a conformidade de um componente ou sistema com os requisitos funcionais. Estes tipos de testes podem ser feitos em todos os níveis de testes. |
| **Não-Funcionais** | Testes realizados para avaliar a conformidade de um componente ou sistema com requisitos não funcionais. Este tipo de teste também pode ser feito em todos os níveis de testes. Existem vários tipos de testes não-funcionais, tais como: Performance, Carga, Stress, Portabilidade, Usabilidade, Segurança, Confiabilidade e Manutenção. |

| | | |
|---|---|---|
| ***Black-Box*** | | Trata-se de um procedimento para definir casos de teste com base numa análise da especificação, funcional ou não-funcional, de um componente ou sistema (sem referência à sua estrutura interna). |
| ***White-Box*** | | Corresponde a um procedimento para definir casos de teste com base numa análise da estrutura interna de um componente ou sistema. |
| **Relacionados à Mudança** | Confirmação | Teste realizado após a correção de defeitos com o objetivo de confirmar que as falhas causadas por esses defeitos já não ocorrem mais. |
| | Regressão | Teste a um componente ou sistema que já foi testado anteriormente, mas que teve uma modificação. Assim sendo, é necessário garantir que não tenham sido introduzidos defeitos ou não sejam descobertos defeitos em áreas inalteradas do software. |

## 2.5 Testes manuais e testes automatizados

A realização dos testes de software pode ser efetuada de duas formas [12]:

- **Manual -** consiste na reprodução das tarefas por parte de uma pessoa previamente definida;
- **Automatizada -** consiste na automação do processo do teste manual, isto é, utiliza-se um software que imita a interação do ser humano com o sistema, onde são desenvolvidos *scripts* que simulam as tarefas manuais e que são executados automaticamente.

Ambas as formas anteriores possuem as suas vantagens e desvantagens. Atualmente os testes manuais são menos vantajosos devido ao tempo de execução e também ao número de vezes que podem ser executados num curto espaço de tempo devido ao processo de *continuous delivery*. Já a execução de testes automatizados é uma mais-valia, pois possibilita uma redução do tempo de execução, uma maior abrangência dos testes e, consequentemente, uma melhoria da sua qualidade. Isto permite também que os *Testers* apliquem o seu esforço noutro tipo de testes ou naqueles que não possam ser automatizados [12].

O principal motivo do crescimento da automação de testes de software é a redução do tempo de execução dos mesmos. Adicionalmente, as aplicações ficaram mais complexas, o que requer maior controlo da qualidade do produto entregue [8].

A automatização de testes para ser bem-sucedida requer uma análise fundamentada do sistema a automatizar. Consequentemente, deve-se ter em conta as seguintes questões [8]:

1) **Porquê automatizar?**

   É necessário perceber a importância de automatizar, definindo os motivos e objetivos. Pois, caso contrário pouco ou nenhum retorno poderá trazer para a empresa.

2) **O que automatizar?**

   O sistema alvo de testes tem de ser identificado, bem como os módulos ou áreas que precisam de ser testadas.

3) **Quando automatizar?**

   Dependendo do tipo de testes a automatizar, o ideal seria, após a definição dos casos de teste, para se conseguir obter uma pilha de regressão. Mas, também, tem que se ter em conta a arquitetura, o negócio, o tempo do projeto, os recursos, a priorização de tarefas, etc.

4) **Onde automatizar?**

   Idealmente, o ambiente de testes deve simular a realidade ou, pode aproveitar partes dos ambientes de desenvolvimento ou produção.

5) **Como automatizar?**

   Também depende do tipo de teste que se pretende implementar de forma automatizada. Deve-se ter em conta as boas práticas e diretrizes da escrita automatizada de cada tipo de teste, bem como selecionar as ferramentas/*frameworks* mais adequadas para se proceder à automatização dos diferentes tipos de testes.

6) **Quem vai automatizar?**

   É muito importante definir quem vai participar na automatização de testes e quem vai realizá-la. Tal, depende também, do tipo de testes a serem automatizados porque são requeridas capacidades específicas.

7) **Quanto custa automatizar?**

   O preço investido na automação tem que ser recuperado na medida em que novas falhas são descobertas (ou antigas são evitadas) antes que a aplicação entre em produção.

# 3. Padrões ideais para a definição de testes de software

Com a evolução da tecnologia, surgiram padrões que tornam mais fácil a reutilização de soluções e arquiteturas bem-sucedidas, para construir sistemas de forma flexível e fácil de manter [58].

Esta secção descreve os padrões que representam onde e como se deve investir no controlo de qualidade no ciclo de desenvolvimento de um software. De seguida, apresenta metodologias de escrita de testes automatizados.

## 3.1 Pirâmide dos níveis de testes

Martin Fowler, argumenta que os testes das camadas mais baixas (testes unitários) devem servir como a primeira linha de defesa, garantindo que os *bugs* previamente encontrados não voltem a ocorrer [22]. Já os testes de camadas mais altas (testes *end-to-end*) devem servir como uma segunda linha de defesa, porque se ocorrer alguma falha, pressupõe-se que possa existir algum defeito no código funcional e/ou ter um teste unitário ausente ou incorreto [22].

Na Figura 3-1, observam-se três níveis de testes (UI, *Service*, *Unit)*, mostrando à direita a relação com o custo e à esquerda a relação com a velocidade de desenvolvimento.

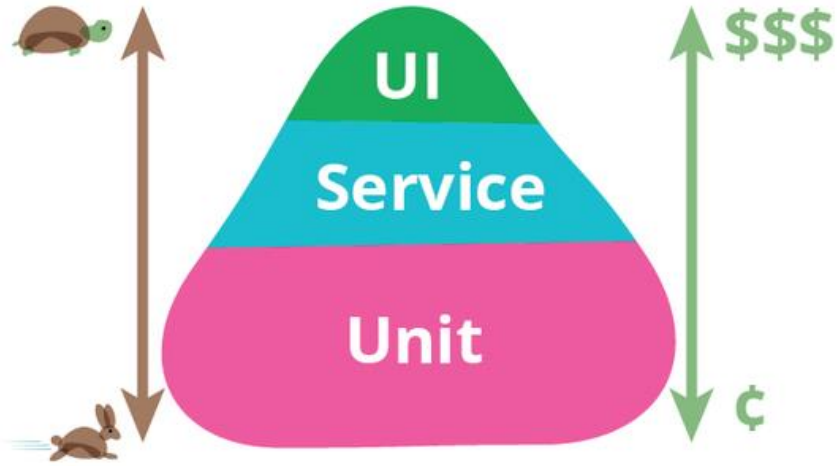



Figura 3-1: Pirâmide de testes segundo Martin Fowler [17]

Assim, da Figura 3-1 retira-se que é muito mais rápido e barato fazer testes de baixo nível do que testes de alto nível. Logo, para poupar tempo e dinheiro, a quantidade de testes deve aumentar, à medida que descemos para os níveis inferiores [17].

## 3.2 Pirâmide ideal de testes de software e os cones anti-padrão

Uma outra representação da pirâmide da Figura 3-1, são as pirâmides dos cones anti-padrão e ideal como se pode observar na Figura 3-2.

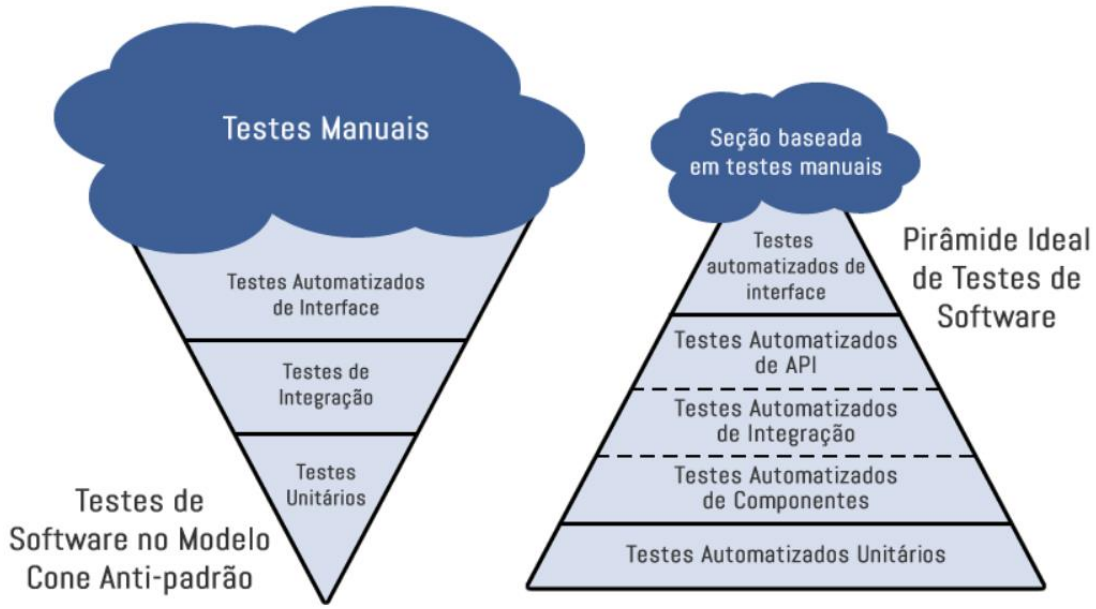



Figura 3-2: Pirâmide dos cones anti-padrão e ideal de testes de software [18]

Por vezes, algumas organizações usam a pirâmide ideal de testes invertida, formando assim o cone anti-padrão. Tal acontece quando a quantidade de testes automatizados de baixo nível (testes unitários, de componente ou de integração) é baixa, a quantidade de testes automatizados de interface é alta e há um número ainda maior de testes manuais, tal como é ilustrado na Figura 3-2. Mas, como a automação de testes, atualmente é algo que ganhou relevância no ciclo de desenvolvimento de software, o "cone anti-padrão" é cada vez menos comum, porque os ciclos tornaram-se muito mais rápidos para poder entregar pequenas atualizações num curto espaço de tempo, mesmo em sistemas de grande complexidade [50]. Então, para que os testes acompanhem estas entregas contínuas frequentes, é necessário recorrer a maiores quantidades de testes nos níveis mais baixos e ir diminuindo à medida que subimos para os níveis mais altos porque, como ilustrado na Figura 3-1, é mais rápido e barato fazer testes de baixo nível do que testes de alto nível. O maior benefício de diminuir drasticamente o volume de testes, à medida que subimos na pirâmide ideal de testes, é o de reduzir-se a incidência de bugs [21] [22].
Para uma melhor otimização do esforço na escrita dos testes, pode-se recorrer a metodologias de desenvolvimento já existentes, tais como *Test-Driven Development* para testes de níveis mais baixos e *Behavior-Driven Development* para testes de níveis mais altos, ajudando assim, na garantia de qualidade do software.

## 3.3 *Test-Driven Development*

O *Test-Driven Development* (TDD) é uma metodologia indicada para a escrita de testes de níveis mais baixos (testes unitários) introduzida por Kent Beck's [65]. Como se pode observar na Figura 3-3, a metodologia TDD baseia-se na execução de ciclos com as seguintes etapas [22]:

1. Escreve-se primeiramente um teste e observa-se o porquê da sua falha;
2. Escreve-se uma quantidade mínima de código para fazê-lo passar;
3. Simplifica-se o código para melhorar pontos como legibilidade (se necessário).

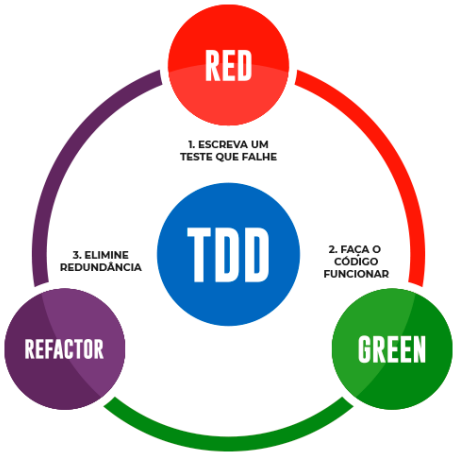



Figura 3-3 : Ciclo de TDD [66]

Esta metodologia é aplicada no nível de testes unitários. Um dos benefícios de escrever testes antes da implementação da funcionalidade é que se vai saber antecipadamente o que o código precisa fazer, evitando escrever código demasiadamente complexo ou que não vá de encontro aos requisitos do negócio [66].

## 3.4 *Arrange, Act, Assert pattern*

*Arrange, Act, Assert* (AAA) *pattern* é uma forma comum de escrever testes unitários para os métodos em teste [24].

- Na secção de *Arrange* inicializam-se os objetos e define-se o valor dos dados que vão ser passados para o método em teste;
- Na secção de *Act* invoca-se o método em teste com os parâmetros organizados na secção de *Arrange*;
- Na secção de *Assert* verifica-se se a ação do método em teste tem o comportamento esperado.

Na Figura 3-4 podemos observar um exemplo da aplicação deste padrão [67].

```
01  [TestMethod]
02  public void TestMySum()
03  {
04      // Arrange
05      int a = 5;
06      int b = 7;
07
08      // Act
09      int result = MySum(a, b);
10
11      // Assert
12      Assert.AreEquals(12, result);
13  }
```

Figura 3-4 : Exemplo do padrão AAA [67]

## 3.5 *Behavior-Driven Development*

O *Behavior-Driven Development* (BDD) também é uma metodologia de testes. Foi introduzida por Dan North [42] que se inspirou na metodologia apresentada anteriormente (TDD). O BDD é uma metodologia em que o desenvolvimento é orientado ao comportamento de um produto. Usa uma linguagem natural, adicionada numa camada acima do *script* automatizado, para que qualquer pessoa envolvida num projeto consiga entender o que os *scripts* de testes automatizados estão a fazer e quais os resultados expectáveis. Esta metodologia pode ser aplicada em qualquer nível de testes.

O BDD usa uma linguagem específica (*Domain Specific Language* - DSL) [43] de fácil entendimento para qualquer pessoa do projeto/negócio chamada de Gherkin [44]. Descreve os comportamentos do software, escondendo os seus detalhes de implementação numa escrita explícita que vai ao encontro dos cenários de testes de aceitação e utiliza a seguinte sintaxe [41]:

- ***Given*** (Dado) algum contexto inicial,
- ***When*** (Quando) um evento acontecer,
- ***Then*** (Então deve-se) garantir alguns resultados.

No BDD, os requisitos ou comportamentos de software são inicialmente definidos em cenários com o formato acima apresentado, onde as linhas com as palavras-chave "*Given*", "*When*" e "*Then*" são conhecidas como "***Steps***". Cada um desses "*Steps*" são analisados e executados por uma *framework* de testes BDD para verificar a expectativa do software. Estes cenários servem como um meio de comunicação entre programadores, *testers* e analistas para melhorar a qualidade do software, tendo por base um melhor entendimento entre todos [41] [22].

## 3.6 *Page object pattern*

O *Page Object* foi introduzido por Martin Fowler [33] e permite-nos criar um repositório de objetos com os elementos HTML de uma página Web. Para cada página, deve haver uma classe correspondente. Esta classe obtém e classifica os *WebElements* da página e também pode conter métodos que executam operações nesses *WebElements*. Ou seja, este padrão faz-nos abstrair da página HTML (ou parte dela) através da utilização da API específica da aplicação, permitindo manipular os elementos da página sem ter que se aceder diretamente ao HTML [68], como se pode observar na Figura 3-5. Com a utilização deste padrão podemos melhorar a organização do código dos testes tendo como benefícios [68]:

- Reaproveitamento de código;
- Código mais limpo;
- Facilidade na manutenção;
- Maior independência dos testes.

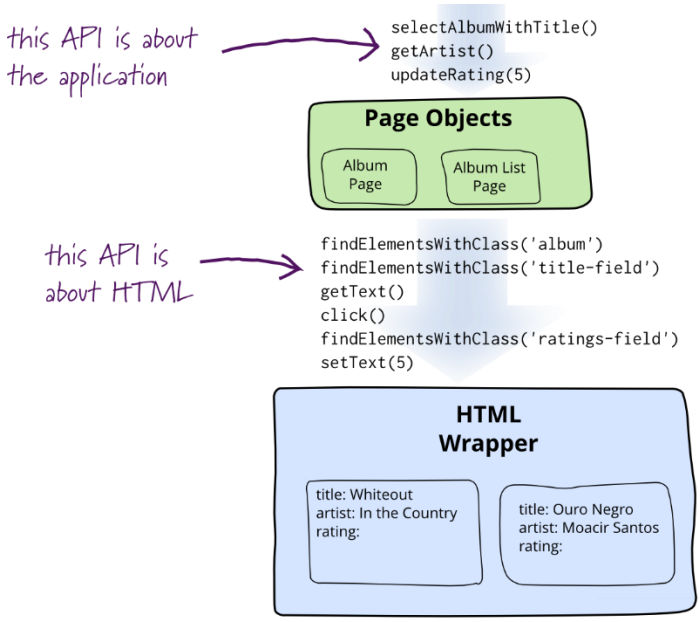



Figura 3-5 : *Page Object Pattern [33]*

Finalizam-se assim os fundamentos teóricos acerca do estado da arte do tema de testes de software**.** Todos os conceitos teóricos apresentados ao longo dos capítulos 2 e 3 servem de base para estruturação e definição da estratégia de testes proposta no próximo capítulo, sendo alguns deles aplicados na integra no âmbito e abordagem da mesma.

# 4. Estratégia de testes proposta

Para se conseguir a garantia da qualidade de qualquer componente dentro de uma arquitetura de um software, requer-se a preparação de uma estratégia bem estruturada por parte do(s) *tester(s)*, tanto de forma manual como automática. Uma estratégia de testes define uma abordagem a seguir no Ciclo de Vida de Testes de Software (STLC) [63].

Uma estratégia de testes deve ser definida no início do ciclo de desenvolvimento do software, mais concretamente na fase de análise de requisitos. Como se referiu no capítulo anterior, um dos princípios do processo de testes é testar o mais cedo possível, porque o custo de um defeito no software aumenta à medida que a sua deteção se afasta do momento em que ele é gerado [104].

Para além disso, o pensamento e planeamento da atividade de testes permite criar uma estrutura de trabalho comum e ajuda uma equipa a definir o âmbito e a abordagem de testes. Isto ajuda, também, qualquer elemento da equipa a obter a qualquer momento e de forma clara o estado atual da qualidade do projeto [75].

Este capítulo, começa por apresentar a arquitetura de software, as ferramentas e linguagens de desenvolvimento adotadas para o caso de estudo que irá ser descrito no próximo capítulo. De seguida, descreve o âmbito e a abordagem da estratégia de testes proposta, tendo em conta as boas práticas e diretrizes no processo de testes, bem como os ambientes e ferramentas de teste. Por fim, descreve o processo automatizado da *pipeline* de *Continuos Integrantion/Continuous Delivery* (CI/CD).

## 4.1 Arquitetura de software

A Figura 4-1 ilustra a arquitetura adotada pela estratégia de testes proposta. Esta arquitetura é baseada no padrão arquitetural de três camadas (3-*Tiers*) [52]. As *tiers* descrevem a distribuição física entre diferentes elementos computacionais.

- ***Tier* 1 – UI (camada de apresentação)** – agrega as classes do sistema com as quais os utilizadores interagem.
- ***Tier* 2 – Negócio (camada de aplicação)** – mantém as classes do sistema responsáveis pelos serviços e regras do negócio.
- ***Tier* 3 – Dados (camada de dados)** – camada responsável pelo armazenamento e recuperação dos dados persistentes do sistema.

Para além destas três camadas, é também importante referir a parte da comunicação que é responsável pela distribuição do sistema em várias máquinas.

Escolheu-se a arquitetura de três camadas porque é um dos padrões arquiteturais mais usados em aplicações web e *web services*. Neste padrão o cliente interage com a aplicação e o servidor aplicacional interage com uma base de dados [53] [74].

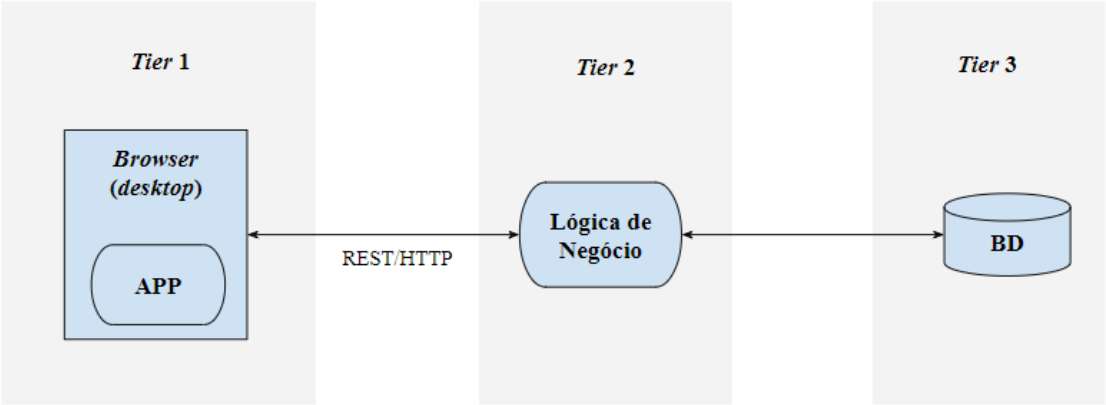



Figura 4-1: Esquema da arquitetura de três camadas

A comunicação é bidirecional entre o *browser (desktop)* e o servidor Web, utilizando o protocolo HTTP para a obtenção, alteração, adição ou remoção de recursos dinâmicos (e.g. os ficheiros JSON gerados pelos *requests*) e obtenção de recursos estáticos (e.g. os documentos HTML, JS, CSS, entre outros) [54].

As mensagens enviadas pelo cliente, geralmente de um navegador da Web, são chamadas *requests*, e as mensagens enviadas pelo servidor como resposta são chamadas de *responses* [54]. Toda a lógica de negócio está construída na camada aplicacional.

O REST é utilizado como padrão arquitetural que proporciona o acesso a funcionalidades que estão no servidor aplicacional e permite a troca de dados entre o servidor aplicacional e uma aplicação cliente.

## 4.2 Ferramentas e linguagens de desenvolvimento

Para a arquitetura apresentada anteriormente podem ser usados vários tipos de ferramentas, servidores e linguagens de programação. Na Figura 4-2 são apresentadas as ferramentas utilizadas na arquitetura proposta [56].

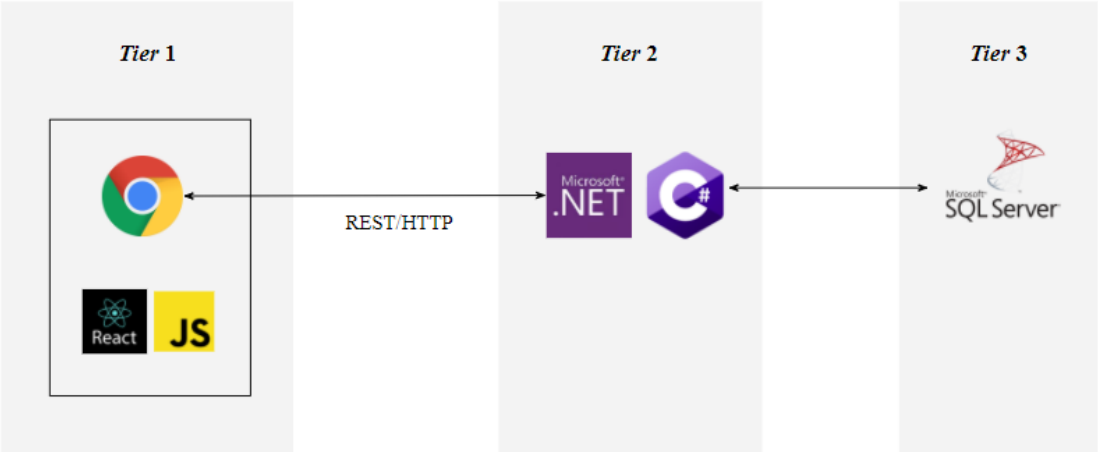



Figura 4-2 : Ferramentas utilizadas na arquitetura apresentada

Na camada de apresentação é usado *React* como biblioteca de *JavaScript* para construir aplicações Web ou componentes de UI [60].

Na camada de aplicação é utilizado o Microsoft .NET framework para construir e suportar a execução dos serviços de aplicações no Windows [61]. Utiliza-se C# como linguagem de programação.

Na camada de dados é usado o sistema de gestão de bases de dados relacionais Microsoft SQLServer que usa a linguagem *Structured Query Language* (SQL) – Linguagem de Consulta Estruturada [62].

Foram escolhidas as ferramentas e linguagens apresentadas anteriormente por serem recentes e representativas do tipo de ferramentas que os programadores utilizam atualmente e, são as mais utilizadas segundo vários tipos de blogs e sites de tecnologias [69] [70] [71] [72] [73].

## 4.3 Âmbito da estratégia

O âmbito da estratégia define o que se vai testar, quais os objetivos dos testes, quais as atividades de teste e quais os seus responsáveis.

Tem-se como objetivos dos testes investir na automação para realizar regressões automatizadas e ganhar tempo para fazer testes manuais exploratórios (testes que ainda não estão pensados nem documentados) e de onde podem surgir problemas ainda não identificados. Quando a automação de testes não existe, todos os casos de teste existentes vão ter de ser reproduzidos manualmente pelos *testers*, o que pode reduzir o tempo para fazer testes exploratórios.

Pretende-se que todos os testes automatizados sejam executados na *pipeline* de CI/CD para que se possa colocar o mais rapidamente possível as alterações no ambiente de produção.

Esta secção descreve os papéis e responsabilidades presentes na estratégia, a metodologia de desenvolvimento aplicada e as fases e atividades do processo de testes.

### 4.3.1 Papéis e Responsabilidades

Os papéis presentes nesta estratégia são os clientes, um líder de equipa, um responsável do produto, quatro programadores (dois de *frontend*[3] e dois de *backend*[4]), um gestor de testes (*test manager)* e dois *testers*, ilustrados na Figura 4-3. Esta estrutura de papéis é muito utilizada nas metodologias ágeis [84] [85].

[3] *Frontend* – é a parte visual de um sistema, a parte com que o *end-user* consegue interagir [100].

[4] *Backend* – como o próprio nome sugere, é o que "está por trás" de um sistema que, na maior parte dos casos, faz o processamento e a ponte dos dados entre o *browser* e a base de dados, aplicando sempre as devidas regras de negócio e validações [100].

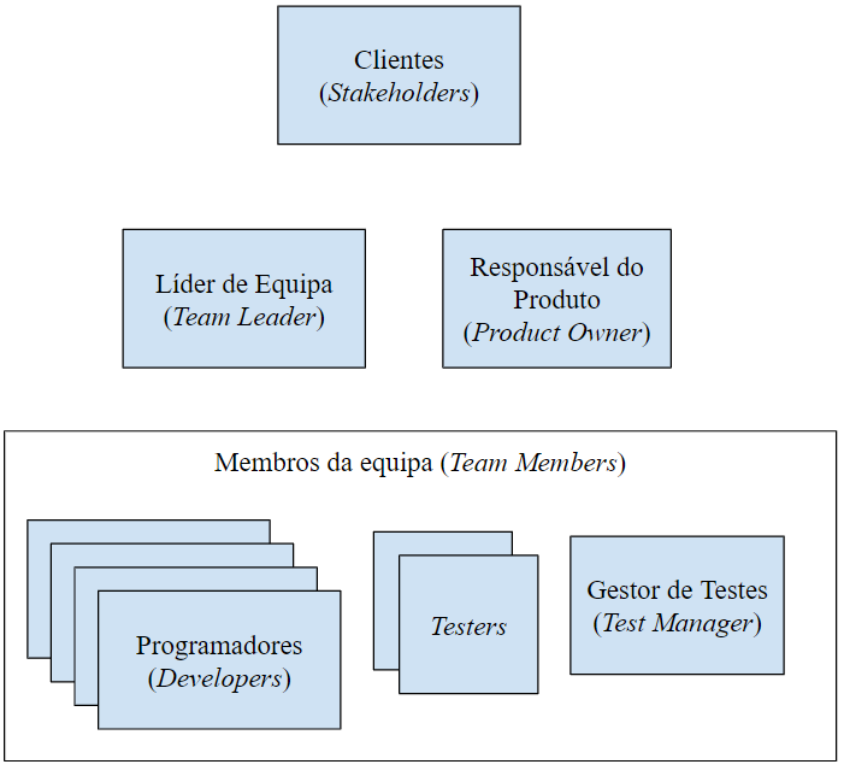



Figura 4-3 : Membros da equipa

Um breve resumo de cada função é exposto de seguida:

- O **Líder de Equipa** é um facilitador dentro da equipa, gere os elementos da equipa e protege-os de problemas. Esta função abrange as capacidades básicas de gestão de projetos, tais como planear e controlar a execução do projeto, conduzir o projeto para o sucesso através das considerações dadas pelos *stakeholders*[5] e clientes [79] [84].
- O **Responsável do Produto** é responsável por documentar os critérios de aceitação e documentar as conversações à volta das *user stories*, bem como a sua priorização. É o principal ponto de contato para o esclarecimento dos requisitos durante o desenvolvimento [83].
- O **Gestor de Testes** tem como responsabilidades orientar as atividades de teste, incluindo a defesa da qualidade e dos testes, planeamento e gestão de recursos e resolução de problemas que representam um obstáculo para as atividades de teste [80].
- Os ***Testers*** são responsáveis pela criação, implementação e execução dos cenários de teste.

[5] *Stakeholders* - é qualquer pessoa ou organização que tenha interesse ou seja afetado pelo projeto. (*Stake*: interesse, participação, risco / *Holder*: aquele que possui) [52].

- Por fim, os **Programadores** são responsáveis pela escrita, manutenção e execução do código de desenvolvimento de um sistema.

### 4.3.2 Metodologia de desenvolvimento

A frequente entrega contínua existente nos ambientes ágeis implica respostas rápidas acerca da qualidade. Deste modo, para reduzir o risco de entregar funcionalidades novas e/ou alteradas ou, até mesmo, correções de falhas sem comprometer a data de entrega, recorre-se à automação de testes. Pois, quantos mais testes estiverem automatizados, mais rápida vai ser a resposta acerca da garantia da qualidade e mais rápido vai ser possível entregar com confiança.
A estratégia proposta tem por base uma metodologia ágil de desenvolvimento – *Kanban* [86] – usada para gestão de testes e porque representa um fluxo de trabalho visual, cujo principal objetivo é controlar o trabalho em progresso. Uma vez definidos todos os papéis e responsabilidades de todos os membros da equipa na secção anterior, existia a necessidade de ter "algo" para organizar os itens em *backlog*[6] por iniciar, em progresso e já concluídos. Por este mesmo motivo, foi escolhida esta metodologia que se foca na gestão de itens de trabalho. Esta metodologia recorre a um quadro (físico ou digital) para planear e acompanhar cada item de trabalho (que pode representar uma tarefa, uma *user story*[7] ou um defeito). São utilizados cartões para cada item de trabalho e cada estado do fluxo, até o item ser finalizado, está representado em colunas, como se pode observar na Figura 4-4. A organização de cada item nas colunas é por prioridade, colocando os itens mais prioritários no topo da coluna. Existe uma limitação de itens em progresso para se poder identificar facilmente quais os itens que precisam de ajuda extra ou de tempo adicional para serem concluídos [81].

[6] *Backlog* - refere-se a uma lista de acumulação de trabalho num determinado intervalo de tempo. Ou seja, é uma "fila de pedidos" em espera. [91].
[7] *User Story* - As *user stories* são um padrão de escrita de alto nível dos requisitos funcionais do negócio [35].

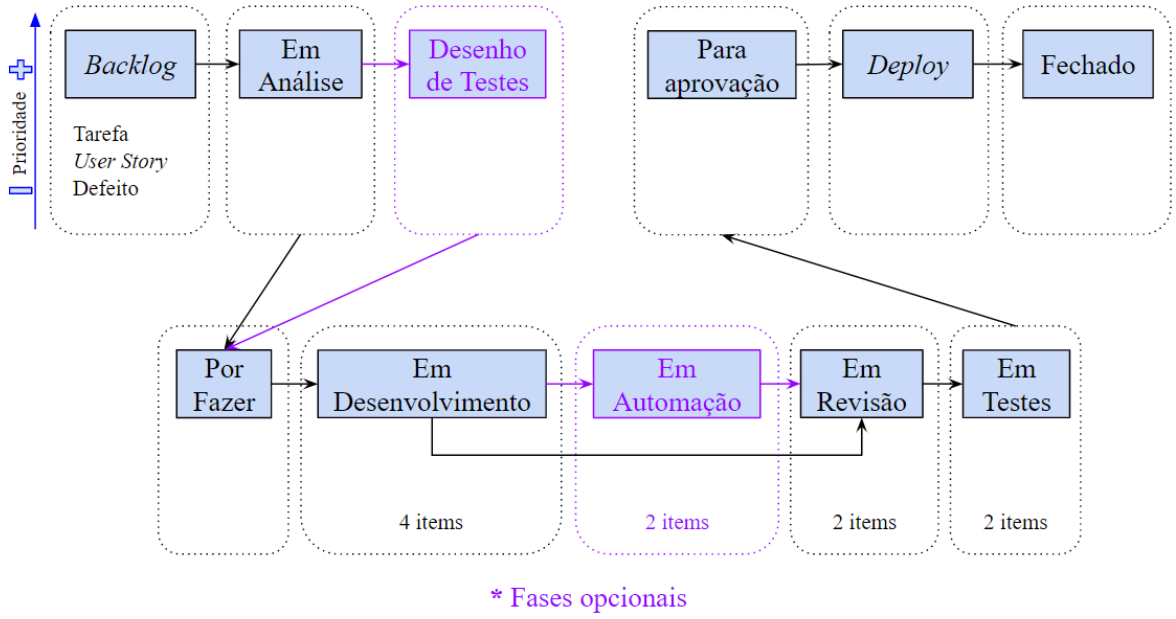



Figura 4-4 : Quadro de *Kanban*

A Figura 4-4 contém os estados que um item terá de percorrer ao longo do seu desenvolvimento, nesta estratégia. O detalhe de cada coluna do quadro *Kanban* é descrito de seguida, bem como a referência aos papéis e responsabilidades definidos na seção anterior:

- ***Backlog***: contém a lista de itens em cartões individuais. A lista também tem os itens que a equipa pode querer trabalhar no futuro, mas que ainda estão a ser planeados.
- **Em Análise**: contém a lista de itens do *backlog* que estão a ser analisados em maior detalhe. É neste estado que vão ser escritos/analisados os critérios de aceitação pelo responsável do produto.
- **Desenho de Testes**: uma vez que o item foi completamente analisado vai passar para a fase de desenho dos casos de testes.
- **Por Fazer**: depois de escritos os casos de teste o item é movido para este estado para mostrar à equipa que a tarefa já está pronta para ser implementada.
- **Em Desenvolvimento**: fase em que os programadores associam itens a si próprios e começam a trabalhar neles. Apenas quando os testes unitários ou de componentes forem concluídos é que se move o cartão para a próxima fase (se for possível de implementar estes testes para o respetivo item).

- **Em Automação**: fase em que os *testers* associam itens a si próprios e começam a trabalhar na implementação da automação dos testes de integração ou dos *testes end-to-end*.
- **Em Revisão**: quando o item estiver desenvolvido e com os respetivos testes unitários, de componente, de integração e de *end-to-end* implementados, é o momento de mover o item para a fase de revisão de código. Nesta fase, pelo menos duas pessoas vão analisar o código desenvolvido e sugerir melhorias ou identificar possíveis erros.
- **Em Teste/Reteste**: depois do item ter completado o processo de revisão de código é movido para a fase de testes onde o *tester* executa os casos testes e faz alguns testes manuais exploratórios.
- **Para Aprovação**: após os testes estarem executados, o item vai ser movido para aprovação por parte do *test manager* e pelo responsável do produto.
- ***Deploy*:** depois da tarefa ser aprovada, é movida para a próxima fase que é o *deploy*, onde se vai proceder às instalações das alterações no ambiente de produção.
- **Fechado**: Quando a tarefa estiver no ambiente de produção.

Alguns destes estados são comuns em metodologias de desenvolvimento ágil (*backlog*, em análise, por fazer, em desenvolvimento, em revisão, em testes, para aprovação, *deploy* e fechado) [81], à exceção dos estados "Desenho de testes" e "Em automação" que foram acrescentados no quadro de tarefas *Kanban* para dar mais visibilidade das atividades de testes. Mas, também, com o objetivo de retirar métricas e conclusões acerca do tempo dedicado a estas atividades de modo a conseguir obter estimativas de trabalho mais concretas. Estes dois estados ("Desenho de testes" e "Em automação") estão assinalados como opcional porque nem todos os itens do *backlog* vão ter casos de teste associados.

### 4.3.3 Fases e atividades do processo de testes

As atividades e fases do processo de testes bem como o(s) responsável(eis) por elas estão ilustradas na Figura 4-5. Estas fases são as usadas normalmente num processo de testes [15] [78], sendo também referidas nos fundamentos teóricos (Capítulo 2) da presente dissertação.
A primeira fase, planeamento dos testes (responsabilidade do *test manager*), tem por base a documentação de informações relevantes para o plano de testes, a seleção dos requisitos que serão testados, a definição dos recursos e do cronograma de atividades de teste. Na segunda fase, desenho de casos de teste, com a ajuda dos *testers*, identificam-se e descrevem-se os casos

de teste, de forma a estruturar os procedimentos de teste. De seguida, na terceira fase, os *testers* e os programadores começam a implementação, automatizando os procedimentos de teste. Após esta fase, vem a execução, como quarta fase, onde os *testers* vão executar os testes e fazer o registo de defeitos. Na quinta e última fase, o *test manager* faz a avaliação dos resultados, mede quantitativamente o progresso dos testes e gera relatórios para posterior avaliação [78].
O documento da estratégia de testes é criado e atualizado pelos *testers* e *test managers* porque são as pessoas mais responsáveis pela gestão da qualidade, mas não significa que os restantes elementos não possam sugerir alterações. Faz sentido, visto que o documento é aprovado pelo gestor de projeto e partilhado com todos os restantes membros da equipa para seu conhecimento.

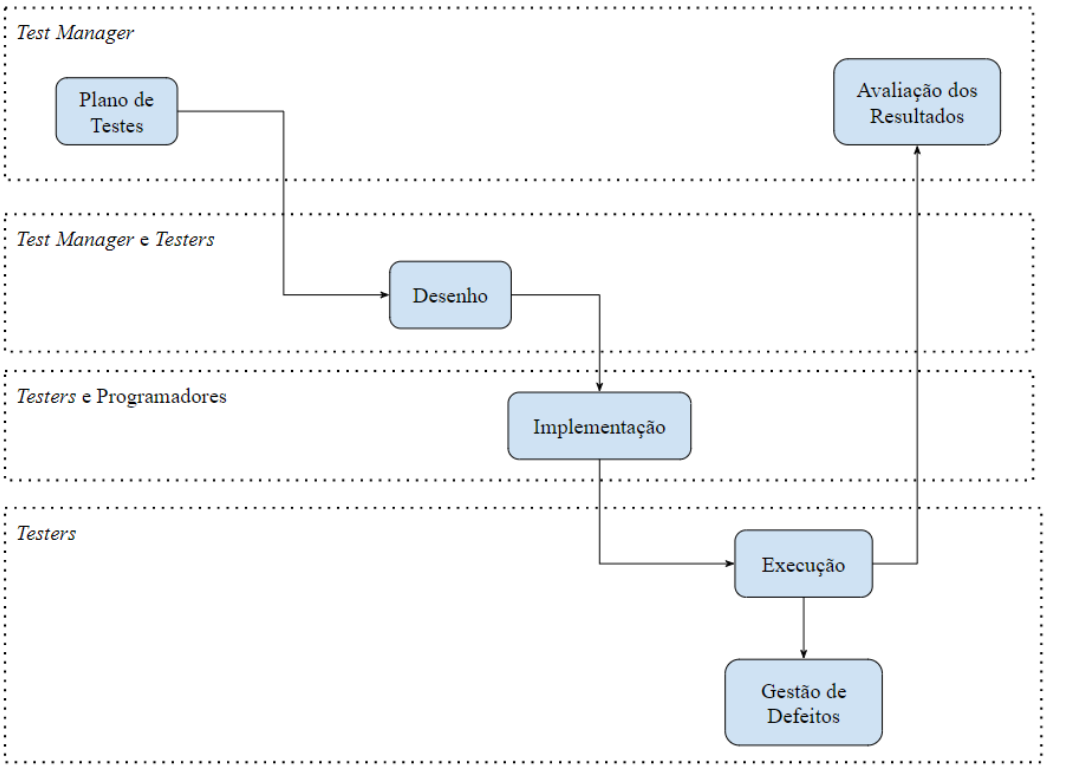



Figura 4-5 : Atividades de teste

## 4.4 Abordagem

A abordagem descreve o que vai ser testado e define que tipos de testes vão ser feitos. Em todas as partes da arquitetura, a abordagem segue todas as atividades de testes mencionadas no âmbito

da estratégia, descrevendo todos os processos e etapas de cada uma das seguintes atividades: plano de testes, desenho, implementação, execução, reteste, gestão de defeitos, gestão de alterações, revisões, aprovações, regressões, monitorização e métricas.

### 4.4.1 Plano de testes

No plano de testes deve ser definido um cronograma com as datas de início e fim de cada atividade de teste e este deve ser partilhado com toda a equipa. É sempre importante definir prazos para cada atividade, pois é necessário alocar recursos no tempo e, para além disto, consegue-se analisar os atrasos das atividades e tomar medidas para não impactar o objetivo final.

O *test manager* vai documentar informações relevantes para a fase de testes e selecionar os requisitos que serão testados. Para isso, o responsável do produto partilha com o *test manager* a lista de requisitos em forma de *user stories*, por área funcional. Cada *user story* vai ter uma prioridade associada (P1 a P3, sendo P1 mais prioritário e P3 menos prioritário), como se pode observar no exemplo apresentado na Tabela 4-1. A estratégia para selecionar e priorizar os casos de teste necessários vai ter em conta a prioridade da *user story*. Por exemplo, todas as *user stories* de prioridade 1 e 2 devem ter casos de teste associados e as de prioridade 3 podem ou não ter casos de teste associados.

Tabela 4-1 : Matriz de prioridades de requisitos

| Área Funcional | USID | Sumário | Descrição | Prioridade |
|---|---|---|---|---|
| Relatórios | US2 | Relatório de Stock | ENQUANTO vendedor, QUERO consultar o *stock* de um determinado produto PARA QUE possa enviar a um cliente [82]. | P1 |
| | US3 | Relatório de Vendas | ENQUANTO diretor, QUERO consultar o volume de vendas do mês PARA QUE possa acompanhar os objetivos [82]. | P2 |
| | US4 | Relatório de Planos | ENQUANTO cliente, QUERO consultar os planos existentes PARA QUE possa decidir qual plano comprar [82]. | P3 |
| *Login/Logout* | US1 | *Login* | ENQUANTO utilizador, QUERO aceder ao sistema PARA QUE possa utilizar as suas funcionalidades. | P1 |
| | US2 | *Logout* | ENQUANTO utilizador, QUERO poder fazer *logout* PARA QUE possa terminar a minha sessão no sistema. | P1 |

Após a priorização e seleção dos casos de teste, é necessário definir juntamente com o gestor de projetos os critérios da fase de testes de um projeto. Os critérios de entrada (conhecidos, também, como *definition of ready*) definem as pré-condições para realizar uma determinada atividade de teste e os critérios de saída (conhecidos, também, como *definition of done*) definem quais condições devem ser alcançadas para se assumir que a atividade de testes está concluída [15]. Ambos os critérios variam de projeto para projeto. É aconselhável ter esta definição de critérios de entrada e saída para se saber quando uma determinada atividade de teste deve começar e acabar [15]. Na Tabela 4-2 estão representados exemplos de critérios de entrada e saída do processo de testes da presente estratégia.

Tabela 4-2 : Critérios de entrada e saída da fase de testes

| Projeto | Critério | Descrição | Progresso | Data de conclusão |
|---|---|---|---|---|
| Projeto 1 | Entrada | Disponibilização da lista de requisitos | Concluído | 07/12/2020 |
| | | Disponibilização do ambiente de testes | Em progresso | |
| | | Disponibilização dos dados de teste | Em progresso | |
| | | Disponibilização dos utilizadores de teste | Em progresso | |
| | | Priorização e seleção de casos de testes para os requisitos recebidos | Não iniciado | |
| | Saída | $x$% dos testes planeados devem ser executados | Não iniciado | |
| | | $y$% de cobertura de testes para as *user stories* de prioridade 1. | Não iniciado | |
| | | $z$% de cobertura de testes para as *user stories* de prioridade 2. | Não iniciado | |
| | | Número máximo de defeitos abertos com máxima severidade (S1) deve ser igual a $x$. | Não iniciado | |
| | | Número máximo de defeitos abertos com severidade média (S2) deve ser igual a $y$. | Não iniciado | |
| | | Número máximo de defeitos abertos com severidade média (S3) deve ser igual a $y$. | Não iniciado | |
| | | Número máximo de defeitos abertos de severidade mínima (S4) deve ser inferior a $z$. | Não iniciado | |

Após a definição dos critérios de entrada e saída procede-se à construção da matriz de severidade de defeitos, porque nos critérios de saída tem-se em conta a severidade dos defeitos reportados durante a fase de testes para se conseguir planear a sua correção. A severidade de

um defeito define a relação entre o impacto que ele tem no funcionamento do sistema e a urgência da sua resolução [87] [88]. Com uma matriz de severidade de defeitos definida, todos os elementos da equipa vão saber exatamente quais as implicações que um defeito tem no sistema. Uma classificação incorreta da severidade de um defeito tem impacto no agendamento das correções dos defeitos, que, por consequência, origina uma má gestão do tempo que os programadores vão gastar na correção dos mesmos [87] [88].

### 4.4.2 Desenho

Durante a fase do desenho, os *testers* vão escrever todos os cenários e casos de teste, com a ajuda do *test manager*. O modelo de escrita dos testes *end-to-end* está representado na Tabela 4-3. E o modelo de escrita dos testes de integração está representado na Tabela 4-4. Cada caso de teste está ligado a pelo menos uma *user story* e tem um conjunto de ações, dados e resultados esperados.

Tabela 4-3 : Modelo de escrita de testes *end-to-end*

| USID | Caso de Teste | Ações | Dados | Resultados Esperados |
|---|---|---|---|---|
| US1 | Login com credenciais válidas | Aceder ao sistema | URL=www.___.pt | Abre a página de login. |
| | | Preencher o *username* | *username*=user143 | |
| | | Preencher a *password* | *password*=pass123 | |
| | | Clicar no botão de login | | Somos redirecionados para a *homepage*. |
| | Login com credenciais inválidas | Aceder ao sistema | URL=www.___.pt | Abre a página de login. |
| | | Preencher o *username* | *username*=user143 | |
| | | Preencher a *password* | *password*=pass167 | |
| | | Clicar no botão de login | | Aparece a seguinte mensagem: "O *username*/*password* são inválidos." |

Tabela 4-4 : Modelo de escrita dos testes de integração

| USID | Casos de Teste | *Endpoint*[8] | Dados de Teste | Resultados Esperados |
|---|---|---|---|---|
| US1 | Login com credenciais válidas | POST login | { username: user143, password: pass123 } | *Status Code*: 200 |
| | Login com credenciais inválidas | POST login | { username: user143, password: pass167 } | *Status Code*: 401 Mensagem: Acesso não autorizado. |

### 4.4.3 Implementação

Durante a fase de implementação, os *testers* e os programadores vão automatizar os procedimentos de teste associados aos casos de teste criados na fase de desenho. Deste modo, para cada nível de teste tem-se em conta as seguintes etapas:

a) Definir a sequência dos casos de teste num procedimento/*script* de teste;
b) Definir como vai ser feita a manipulação dos dados de teste para garantir independência dos testes. Consegue-se manipular os dados de duas formas:
    i. Inserir todos os dados de teste que se vai precisar antes de se começarem a executar todos os testes e após o seu término repor o estado inicial da camada aplicacional e da base de dados;
    ii. Inserir os dados de teste antes de cada teste ser executado e depois de cada execução apagá-los, para melhor perceção do que o teste pressupõe fazer.

A manipulação de dados, para além de ser necessária na execução de testes com dados específicos, modulariza a intenção do teste, promovendo a sua documentação. Na segunda abordagem o tempo de execução global dos testes pode ser maior, uma vez que obriga a executar as *scripts* de alteração e reposição dos dados para cada teste.

[8] *Endpoint* – é uma extremidade de um canal de comunicação. Quando uma API interage com outro sistema, os pontos de contato dessa comunicação são considerados *endpoints* [101].

A Figura 4-6 e a Tabela 4-5 apresentam o tipo de testes realizados nas diferentes partes da arquitetura apresentada no inicio do presente capítulo e, também, quem vai ter a responsabilidade de os criar.

Tabela 4-5 : Abordagem de testes

| Nível de teste | Tipo de Teste | Sistema | Responsabilidade |
|---|---|---|---|
| Testes de Componentes/Unitários | Funcional | APP/API | Programador |
| Testes de Integração | Funcional | API | *Tester* |
| Testes *End-to-End* | Funcional | APP | *Tester* |

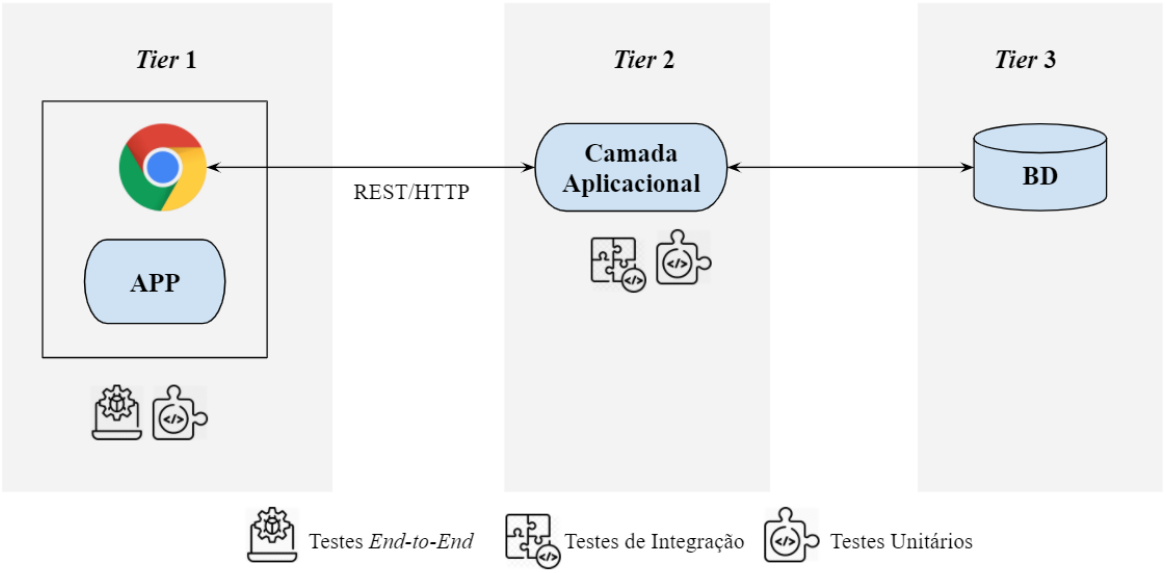



Figura 4-6 : Distribuição da tipologia de testes na arquitetura

Os testes unitários no *backend* (camada aplicacional) e no *frontend* (*web client* - APP) vão ser elaborados pelos programadores, porque foram eles que desenvolveram as classes/componentes do sistema. Por conseguinte, terão mais conhecimento para o fazerem. Os programadores podem ter a colaboração dos *testers* na parte de planeamento e definição dos cenários de teste. Os testes unitários nas classes dos serviços da API utilizam o paradigma TDD (*Test-Driven Development*) e o padrão de escrita AAA (*Arrange, Act, Assert*) explicados no Capítulo 3.

Os testes de integração na API, verificam se os pedidos dos clientes têm o comportamento expectável e verificam se os requisitos do negócio estão a ser cumpridos. Para isso, os testes de integração na API vão ser elaborados pelos *testers*, porque é necessário validar se os componentes que foram desenvolvidos e testados isoladamente pelos programadores quando interligados continuam a cumprir com os requisitos do negócio.

Os testes *end-to-end* no *Web Client* (APP) são a ponta final da arquitetura. Estes testes vão ser elaborados pelos *testers* numa perspetiva funcional, pois vão conter todos os passos para reproduzir uma funcionalidade do sistema. Os testes *end-to-end* utilizam o paradigma BBD (*Behavior-Driven Development*) porque se focam no comportamento do sistema. Normalmente, descrevem as funcionalidades, de modo que sejam de fácil leitura e compreensão. Para isso, é utilizada uma linguagem natural, numa camada acima do código, chamada de *Gerkin*. Estes testes estão organizados segundo o padrão *Page Object* para se ter código mais estruturado e de fácil de reutilização.

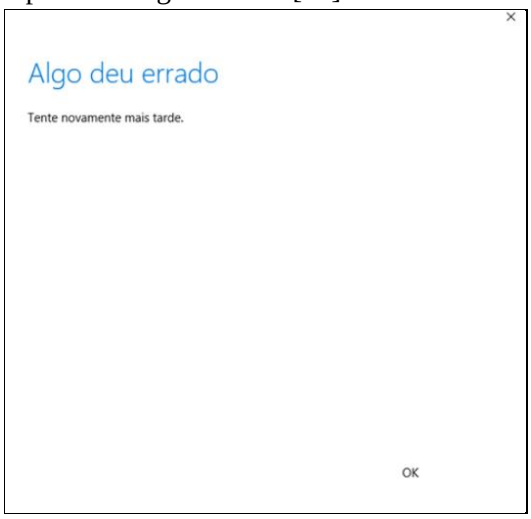
### 4.4.4 Execução, reteste e gestão de defeitos

A gestão de defeitos é uma atividade fulcral da fase de execução de testes. Existe um conjunto específico de estados pelos quais o defeito passa durante todo o seu ciclo de vida, cujo objetivo é coordenar e comunicar facilmente o estado atual do defeito e tornar o seu processo de correção sistemático e eficiente [76].

Quando um defeito é encontrado, deve ser adicionado um cartão do tipo "*bug*" no quadro *Kanban.* Um defeito deve ter todos os detalhes necessários para que outra pessoa o consiga reproduzir, como a identificação da *user story* e caso de teste, passo/ação onde falhou a execução, resultado obtido e resultado esperado, severidade (de acordo com a matriz de severidade definida no plano de testes), ambiente, bem como evidencias (*screenshoots*, documentos, *logs*, gravações, etc) [15] [16]. Para tal, deve ser descrito de acordo com o seguinte modelo apresentado na Tabela 4-6.

Tabela 4-6 : Modelo para descrição de defeitos [103]

| **Título:** | Erro ao efetuar login | | |
|---|---|---|---|
| **Descrição:** | Ao tentar efetuar login com as minhas credenciais aparece um erro não tratado | | |
| **Severidade** | S1 | | |
| **USID** | US1 | | |
| ***Browser*** | Chrome/Mozilla/Edge | | |
| **Caso de Teste** | Login com credenciais válidas | | |
| **Ação** | **Dados** | **Resultado Esperado** | **Resultado obtido (com evidencias)** |
| Aceder ao sistema | URL=www.___.pt | Abre a página de login. | |
| Preencher o *username* | *username*=user143 | | |

| | | | |
|---|---|---|---|
| Preencher a *password* | *password*=pass123 | | |
| Clicar no botão de login | | Somos redirecionados para a *homepage*. | Aparece o seguinte erro [96]:   |
| | | | |

Os estados pelos quais um defeito passa variam de projeto para projeto. Na Figura 4-7 é apresentado o ciclo de vida dos defeitos para a presente estratégia. Todos os estados representados no ciclo de vida dos defeitos estão, também, presentes no quadro *Kanban* apresentado no âmbito da estratégia. Estes estados são os que normalmente são usados no âmbito de gestão de defeitos [76] [77] [89] [90]. Apenas se renomearam alguns estados para fazer a respetiva correspondência aos estados do quadro *Kanban* para que os defeitos estejam incluídos nele.

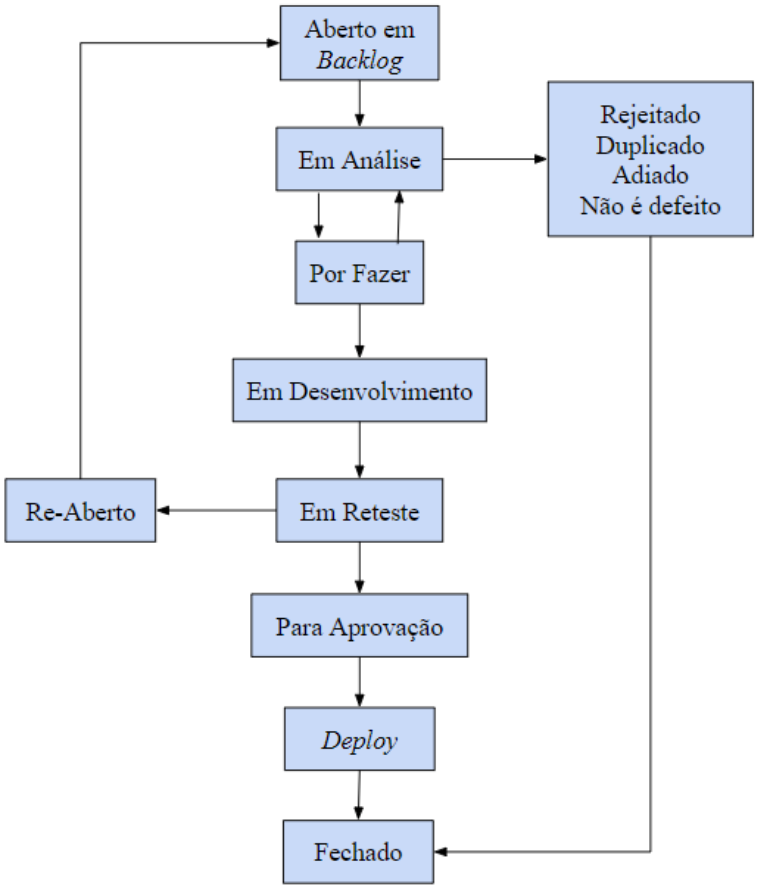



Figura 4-7 : Ciclo de vida de defeitos

Os estados do ciclo de vida de defeitos são descritos da seguinte forma:

- **Aberto em *Backlog* -** quando um novo defeito é registado pelos *testers*.
- **Em Análise -** depois de o defeito ser aberto, o *test manager* começa a sua análise, da qual resulta a sua aceitação e um dos seguintes estados:
    - **Duplicado -** se o defeito se repetir mais do que uma vez ou se corresponder ao mesmo conceito de outro defeito.
    - **Rejeitado -** se o programador achar que não é um verdadeiro defeito, ele volta com o defeito para "em análise" e o *test manager* procede à sua rejeição (se concordar).
    - **Adiado -** se o defeito não tiver grande prioridade no momento atual e se esperar que seja corrigido na próxima versão.
    - **Não é defeito -** se não afetar as funcionalidades da aplicação.

- **Em Desenvolvimento** - o programador começa a analisar e trabalhar na correção do defeito.
- **Em Reteste** - após o programador fazer a correção, o defeito para o estado de reteste e o *tester* executa de novo os casos testes para verificar se o defeito já não acontece mais.
    - **Re-Aberto** - Se o defeito persistir, o *tester* altera o defeito para reaberto e o defeito vai passar novamente pelo seu ciclo de vida.
    - **Para Aprovação** - Se o defeito foi corrigido pelo programador, o *tester* passa o defeito "Para Aprovação" do *test manager*.
- **Deploy** - depois de o defeito ser aprovado, procede-se à instalação das correções no ambiente de produção.
- **Fechado** - Se o defeito não existir mais, o *test manager* atribui o status "Fechado".

Sempre que seja encontrado um defeito no software, deve ser criado um cartão do tipo "tarefa" no quadro de *Kanban* para se cobrir o cenário em falha adicionando/atualizando os casos de teste e, se possível, adicionar/atualizar os scripts dos testes automatizados para se ter garantia de que não volta a acontecer sempre que se executar a regressão dos mesmos.

### 4.4.5 Gestão de alterações

Para além da gestão de defeitos, é também necessário gerir as alterações de funcionalidades que já passaram por todo o processo de desenvolvimento e já estão em ambiente de produção. Na Figura 4-8 podemos observar todas as etapas da gestão de alterações no quadro de *kanban.*

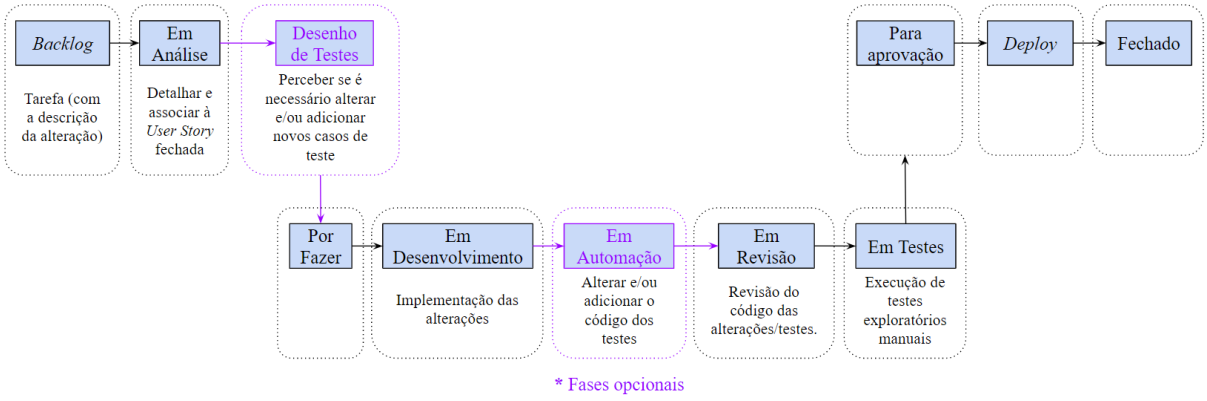



Figura 4-8 : Gestão de alterações

Sempre que for necessário fazer alterações a *user stories* já fechadas é necessário criar um cartão do tipo "tarefa" no quadro *kanban* com o detalhe das alterações e associar esse cartão à

*user story* fechada. Depois é necessário perceber os impactos dessa alteração no que já está implementado e perceber se é necessário alterar ou criar novos casos de teste. Após essa análise a alteração está pronta a ser implementada. Depois da implementação, é necessário alterar/criar os *scripts* dos testes automatizados relativos a essa alteração.

Seguidamente, faz-se a revisão do código de desenvolvimento e dos *scripts* dos testes automatizados. Concluída a revisão do código, procede-se à execução dos testes automatizados e faz-se alguns testes exploratórios manuais. Por fim, a alteração vai para aprovação. Assim que estiver aprovada, faz-se o *deploy* para o ambiente de produção e fecha-se a tarefa. Tal como os defeitos, as alterações também são geridas no quadro de *kanban*.

Na secção 4.6 Pipeline de CI/CD pode-se consultar mais pormenorizadamente todo o processo de gestão de alterações.

### 4.4.6 Revisões, aprovações e regressões

De forma a reduzir a probabilidade de um erro humano, é importante que todos os casos de teste e *scripts* de automação sejam revistos por dois membros da equipa: um programador, porque conhece bem a lógica da aplicação; e um *tester*, porque é especializado no processo de testes. Só depois da sua aprovação é que se pode avançar para a próxima fase. Nesta revisão eles podem sugerir melhorias no código ou identificar possíveis erros.

Quando todas as atividades de teste estiverem definidas, o plano da estratégia de testes tem de ser revisto e aprovado por todas as entidades envolvidas na gestão do projeto. Todas as alterações provenientes da revisão devem ser rastreadas no início do documento, juntamente com o nome do aprovador, a data e um comentário. Além disso, é um documento que deve ser continuamente revisto e atualizado com os desenvolvimentos do processo de teste [75].

Como o objetivo desta estratégia é ter as regressões automatizadas, isto faz com que sejam sempre executadas na *pipeline* de CI/CD quando se procede a alguma alteração no código aplicacional.

### 4.4.7 Monitorização e métricas

Durante todas as etapas mencionadas anteriormente, deve-se fazer uma constante monitorização do estado dos testes. Para tal, vão ser gerados vários tipos de relatórios. Alguns são gerados automaticamente, após a execução dos testes automatizados. Estes dão logo visibilidade da quantidade de testes, do seu estado (passou, falhou ou não executado) e do seu tempo de execução. Existem outros que nos permitam saber o estado atual dos testes dos

requisitos do sistema. Estes, podem ser construídos manualmente, mas, atualmente já existem ferramentas de gestão de testes que geram automaticamente este tipo de relatórios. São exemplos Klaros-Testmanagement [92], Test Rail [93], Xray [94] [95]. Os relatórios que devem ser gerados são os seguintes:

- O relatório de rastreabilidade entre requisitos, casos de teste, execuções e falhas encontradas nas execuções (Tabela 4-7). Este relatório disponibiliza várias métricas, tais como:
    - Qual a *user story* com mais defeitos reportados;
    - Quem executou que testes e que defeitos reportou;
    - Os casos de teste de cada *user story*;
    - E que tipo de teste está associados a cada *user story.*
- O relatório de cobertura de requisitos, mostra a quantidade de requisitos com e sem cobertura de testes (Figura 4-9). Uma baixa cobertura de testes pode indicar baixa qualidade, porque existem requisitos sem casos de teste associados. O ideal seria ter 100% de cobertura de testes, visto que cada requisito teria de ter pelo menos um caso de teste associado.
- O relatório de defeitos reportados por severidade e por estado (Tabela 4-8). Deste relatório pode-se retirar o total de defeitos reportados, o total de defeitos por severidade e o total de defeitos por estado.
- O relatório da quantidade de testes por tipo de teste (Figura 4-10).

Após as atividades de automação de testes serem finalizadas, guardam-se os dados e outras informações relevantes para consolidar experiência e analisar possíveis melhorias.

Tabela 4-7 : Relatório de rastreabilidade

| USID | | Caso de Teste | Tipo de Teste | Executado por | Estado | Defeitos |
|---|---|---|---|---|---|---|
| US1 | 1 | Login com credenciais válidas | Manual | *Tester* 1 | Passou | |
| | | | Manual | *Tester* 2 | Falhou | Defeito 1 |
| | | | Manual | *Tester* 2 | Passou | |
| | | | Automatizado – E2E | Auto | Não Executou | |
| | 2 | Login com credenciais inválidas | Manual | *Tester* 1 | Falhou | Defeito 2 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | Manual | *Tester* 1 | Passou | |
| | | | Automatizado – E2E | Auto | Não Executou | |
| | | | Automatizado - API | Auto | Não Executou | |

Tabela 4-8 : Relatório de defeitos reportados

| | **S1** | **S2** | **S3** | **S4** | Total |
|---|---|---|---|---|---|
| Aberto | 1 | 1 | 2 | 3 | **7** |
| Em Análise | | | | | |
| Por fazer | | | | | |
| Em desenvolvimento | 1 | | | | **1** |
| Em Reteste | | 1 | | | **1** |
| Para Aprovação | | | | | |
| *Deploy* | | | 1 | | **1** |
| Fechado | | | | | |
| Total | **2** | **2** | **3** | **3** | **10** |

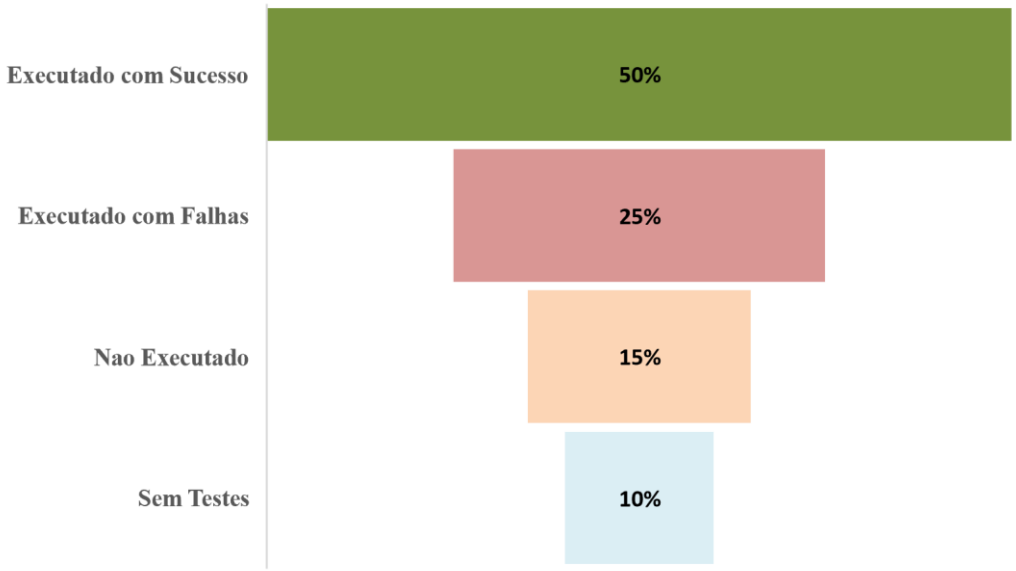



Figura 4-9 : Relatório de cobertura de testes

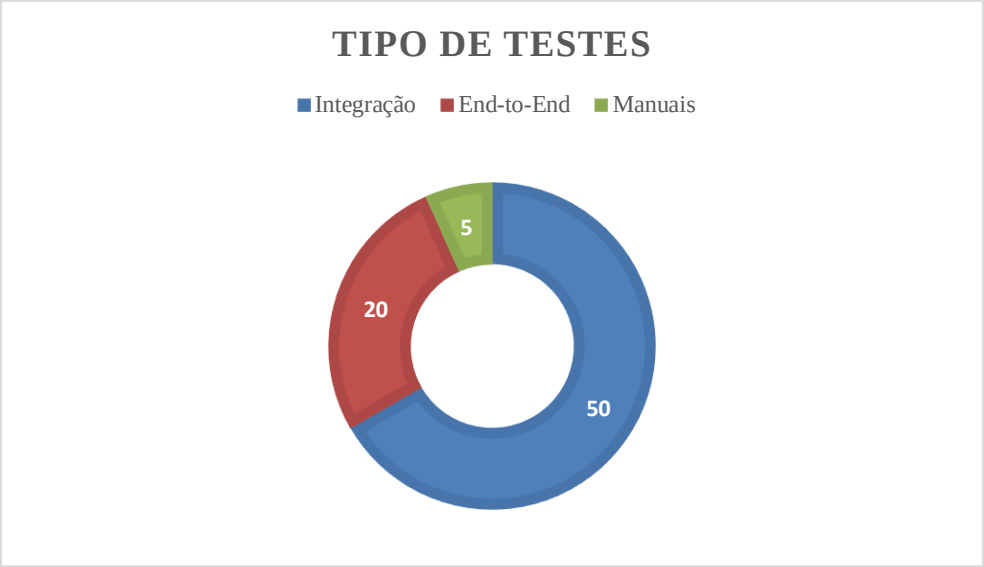



Figura 4-10 : Relatório de quantidades de testes

## 4.5 Ambientes e ferramentas de teste

Os ambientes de teste são muito importantes porque permitem simular ambientes idênticos ao de produção sempre que for preciso testar os sistemas. Necessitam de ser ambientes estáveis para que o *tester* tenha confiança nos resultados do teste. Nesta estratégia vai-se ter os seguintes ambientes:

- Ambiente local ou Ambiente de desenvolvimento – ambiente que é executado na própria máquina do programador;
- Ambiente de testes ou Ambiente de *Staging* – ambiente utilizado para testes manuais e/ou automatizados;
- Ambiente de produção – Ambiente final que serve os clientes/utilizadores finais.

Quanto às ferramentas de teste, é importante trabalhar com ferramentas de automação ideais para o software/arquitetura que está a ser desenvolvido/a. Deste modo, pesquisou-se por ferramentas de teste *open-source* em que a linguagem de programação fosse *Javascript* ou c#. Pretende-se manter consistência com as linguagens de desenvolvimento utilizadas na arquitetura apresentada anteriormente e que se pudessem adaptar a vários tipos de *frameworks*. Após a análise das ferramentas mais comuns e atuais para cada tipo de testes, as que foram selecionadas para escrever e executar cada nível de testes são:

- Para a automação dos testes de componentes é usado “JEST” com a biblioteca “React Testing Library”;
- Para a automação de testes unitários é usado “xUnit.net”;
- Para a automação de testes de integração é usado o “JEST” com a biblioteca “Supertest”;
- Para a automação de testes *end-to-end* é usado o “webdriverIO” com a *framework* “Cucumber”.

A Figura 4-11 representa na pirâmide ideal de testes as ferramentas selecionadas (xUnit.net, React Testing Library, Jest, Surpertest, WebdriverIO) por nível de teste. Pode-se observar que, de acordo com a pirâmide ideal de testes, se investe em maior quantidade, nos testes unitários e nos testes de componente, de seguida nos testes de integração, depois nos testes *end-to-end* e, por fim, nos testes manuais exploratórios. Pode-se, também, identificar de quem é a responsabilidade da elaboração dos testes de um determinado nível e, ainda, quais dos testes são manuais e/ou automatizados.

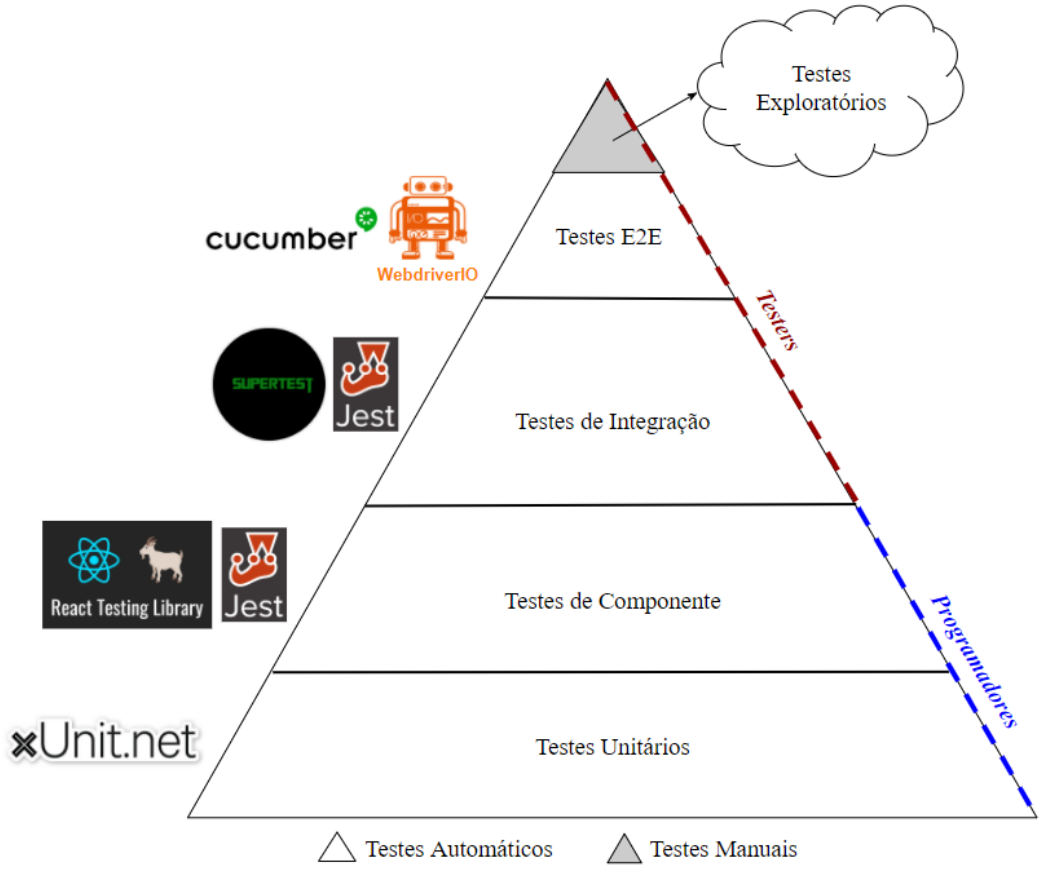



Figura 4-11 : Ferramentas de Teste

### 4.5.1 Instalação das ferramentas/*frameworks* para testes de integração

De acordo com a arquitetura apresentada, delineou-se fazer testes de integração na API. Para fazer estes testes, utiliza-se *Javascript* como linguagem de programação e as seguintes ferramentas/*frameworks open source*:

- ***Node.js*** - Plataforma para construir *Web Applications* escaláveis e de alta *performance* usando *JavaScript* [29] [59];
- **NPM** (*Node Package Manager*) - Repositório *online* e *open source* para a publicação de projetos *Node.js*. Funciona através da linha de comando e é assim que consegue interagir com o repositório, ajuda na instalação de *packages*, controlo de versões e a gerir dependências [30].
- ***Jest*** - *Framework* em *Javascript* para ajudar na execução de testes em que se foca na simplicidade, ou seja, é como se fosse um "*Test Runner*" [31];
- ***Supertest*** - Biblioteca que compreende facilmente pedidos HTTP [32].

Relativamente à instalação, primeiramente ter-se-á de instalar o *node.js*. Para isso basta fazer o *download* do executável (.exe) através da *internet*. Após isto, através da linha de comandos deve-se executar os seguintes comandos:

- ***npm init*** - para inicializarmos a utilização do *npm* e definir algumas informações sobre os testes/projeto, nomeadamente a parte em que se define qual o comando que vai fazer com que corram os testes, neste caso colocou-se "*test*". Após a execução deste comando vai-se ter na pasta selecionada um ficheiro cujo nome é "*package.json*" que serve para guardar algumas definições dos testes, como *scripts* de execução de comandos, caminhos de diretórias onde são guardados *outputs* dos resultados dos testes, dependências, etc;
- ***npm install jest supertest*** – para se instalar os *packages* do *jest* e *supertest*. Após a execução deste comando vai-se ter uma pasta chamada "*node_modules*" que é onde estão guardados os packages instalados. Também os ficheiros "*package.json*" e "*package-look.json*" são alterados para incluírem as informações dos novos *packages* instalados.

Após executar os dois comandos anteriores e caso se necessite de colocar o repositório dos testes noutra máquina, apenas é necessário efetuar ***npm install*** ou ***npm i***– para instalar o *npm*, todos os *packages* contidos nas dependências definidas no ficheiro "*package.json*". Não será necessário voltar a instalar cada *package* individualmente.

Na Figura 4-12 podemos observar os *outputs* da instalação. O ficheiro "*package.json*" deve ficar com as mesmas configurações ilustradas na Figura 4-13.

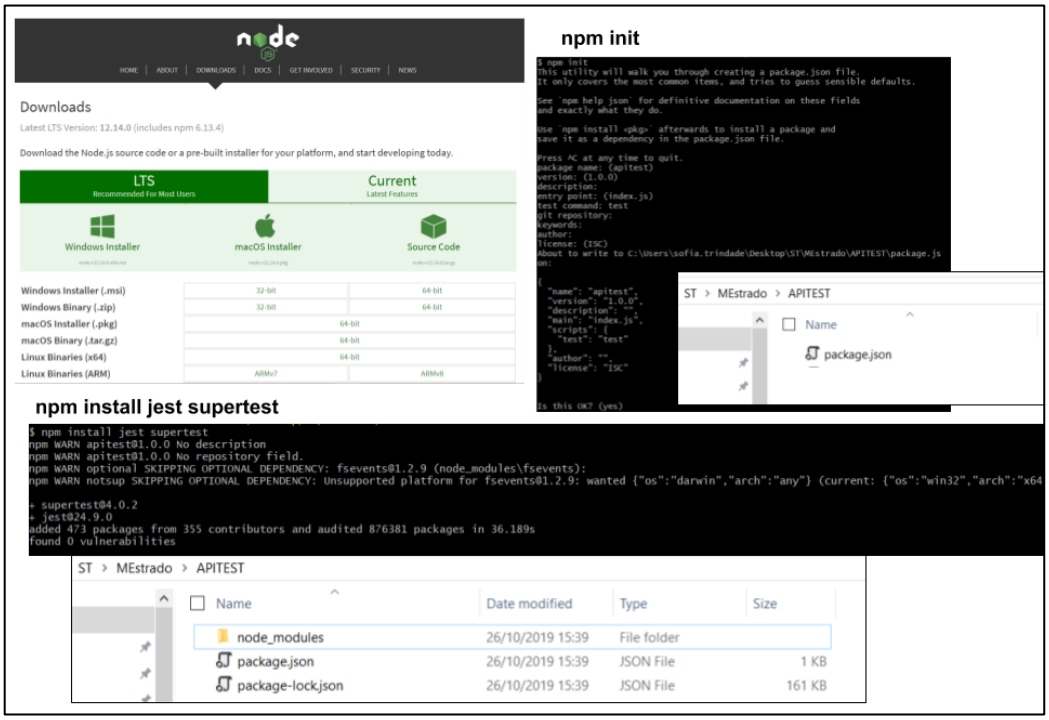



Figura 4-12 : *Outputs* da instalação das ferramentas/*frameworks* necessárias para os testes de integração

```
{
  "name": "apitest",
  "version": "1.0.0",
  "description": "Testes de Integração",
  "main": "index.js",
  "scripts": {
    "test": "test"
  },
  "author": "Sofia Trindade",
  "license": "ISC",
  "dependencies": {
    "jest": "^24.9.0",
    "supertest": "^4.0.2"
  }
}
```

Figura 4-13 : Package.json

### 4.5.2 Instalação das ferramentas/*frameworks* para os testes *end-to-end*

Para os testes end-to-end utiliza-se, também, *Javascript* como linguagem de programação, o *node.js* e o *npm.* Para além destes, utiliza-se:

- ***WebdriverIO*** – é uma *framework open source* de testes automatizados para node.js [97];
- ***ChromeDriver*** - é um servidor autónomo que usa como padrão W3C *WebDriver*[9]. O *ChromeDriver* está disponível para o *Chrome* em *Android*/*Desktop* (Mac, Linux, *Windows* e *ChromeOS*) [98].

Relativamente à instalação, primeiramente ter-se-á de instalar a versão do *chromedriver*, de acordo com a versão do *chrome* que estiver instalada na máquina pretendida. Para isso, basta fazer o *download* do executável (.exe) através da *internet*. Após o *download* estar concluído, deve-se copiar o ficheiro "chromediver.exe" para uma pasta no disco (C:), como se observa na Figura 4-14.

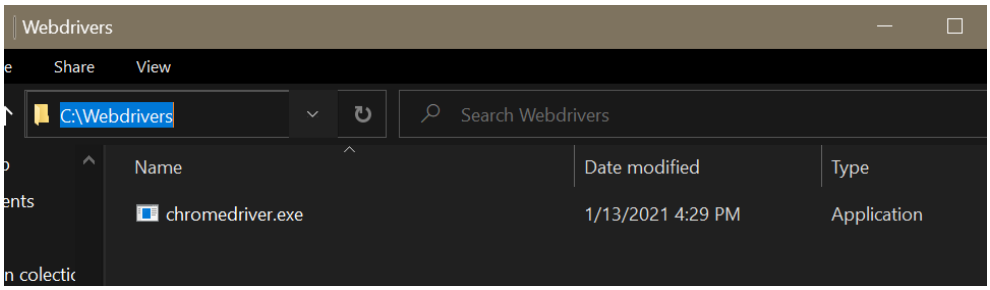



Figura 4-14 : Mover o *chromedriver* uma pasta do sistema

Seguidamente, é necessário incluir o caminho desde *webdriver* nas variáveis de ambiente do *Windows* representado na Figura 4-15.

[9] *W3C WebDriver* - *webdriver* é uma ferramenta *open source* para testes automatizados de aplicações *web* em vários *browsers*. Fornece recursos para navegar entre páginas *web*, elementos HTML, execução de *JavaScript* entre outros [98].

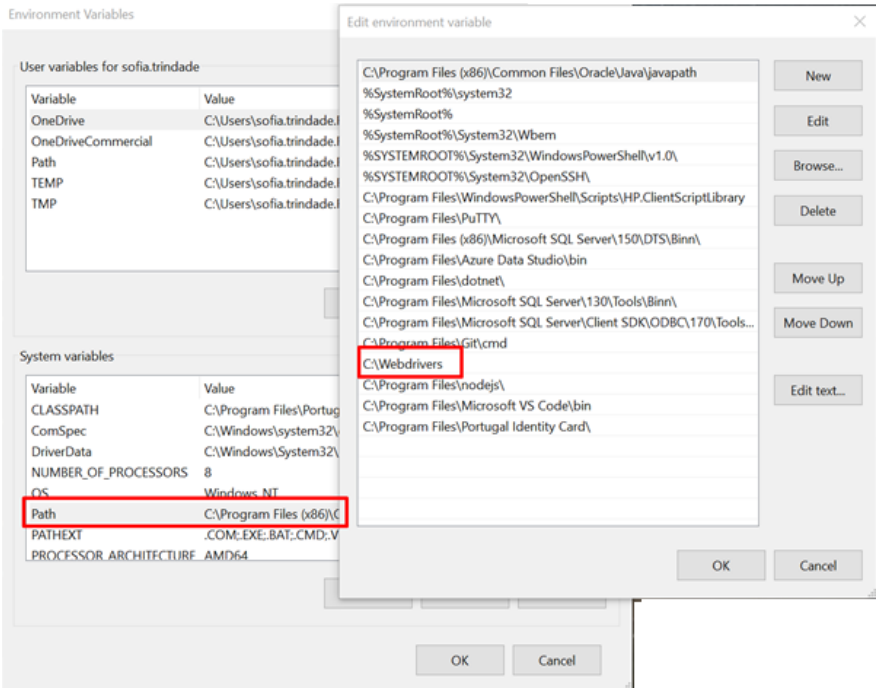



Figura 4-15 : Adicionar o *chromedriver* às variáveis de ambiente do *Windows*

De seguida, através da linha de comandos deve-se aceder a uma pasta onde se queira guardar os testes e executar os seguintes comandos:

- ***npm init -y*** – para inicializarmos a utilização do *npm* com as suas configurações por omissão;
- ***npm i --save-dev @wdio/cli*** – para se instalar o *test runner* do *webdriverIO*;
- **npx wdio config** – para gerar o ficheiro de configurações e instalar os seus respetivos *packages*. Não será utilizado o "*-y*" no final do comando porque não se vai utilizar as configurações por omissão de instalação. As configurações que vão ser utilizadas estão descritas a seguir, mas também, podem ser consultadas na Figura 4-17.
    - *@wdio/local-runner;*
    - *@wdio/cucumber-framework;*
    - *@wdio/spec-reporter;*
    - *@wdio/sync;*
    - *Chromedriver.*

Na Figura 4-16 podemos observar os *outputs* da instalação. Na Figura 4-18 podemos observar alguns resultados das configurações definidas, tais como, a criação das pastas "*features*", "*step-definitions*" e "*pageobjects*" cada uma delas com um ficheiro exemplo e também, a criação do ficheiro de configurações do *webdriverIO* "*wdio.conf.js*". Este ficheiro é muito importante, pois é nele que se vai definir todas as condições para executar os testes.

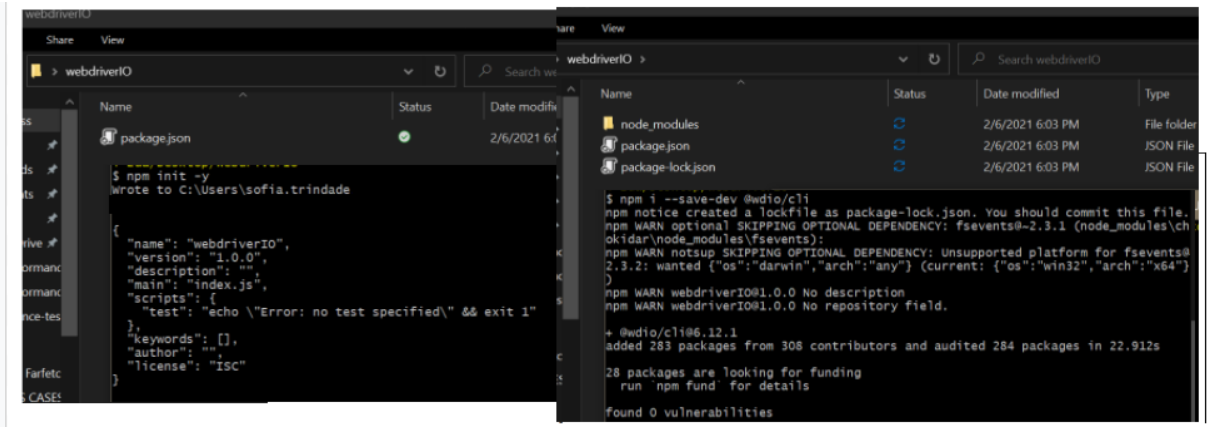



Figura 4-16 : *Outputs* da instalação das ferramentas/*frameworks* necessárias para os testes de *end-to-end*

```
$ npx wdio config

=========================
WDIO Configuration Helper
=========================

? Where is your automation backend located? On my local machine
? Which framework do you want to use? cucumber
? Do you want to run WebdriverIO commands synchronous or asynchronous? sync
? Where are your feature files located? ./features/**/*.feature
? Where are your step definitions located? ./features/step-definitions/steps.js
? Do you want WebdriverIO to autogenerate some test files? Yes
? Do you want to use page objects (https://martinfowler.com/bliki/PageObject.html)? Yes
? Where are your page objects located? ./features/pageobjects/**/*.js
? Are you using a compiler? No!
? Which reporter do you want to use? spec
? Do you want to add a service to your test setup? chromedriver
? What is the base url? http://localhost

Installing wdio packages:
- @wdio/local-runner
- @wdio/cucumber-framework
- @wdio/spec-reporter
- wdio-chromedriver-service
- @wdio/sync
- chromedriver
```

Figura 4-17 : Configurações do *webdriverIO*

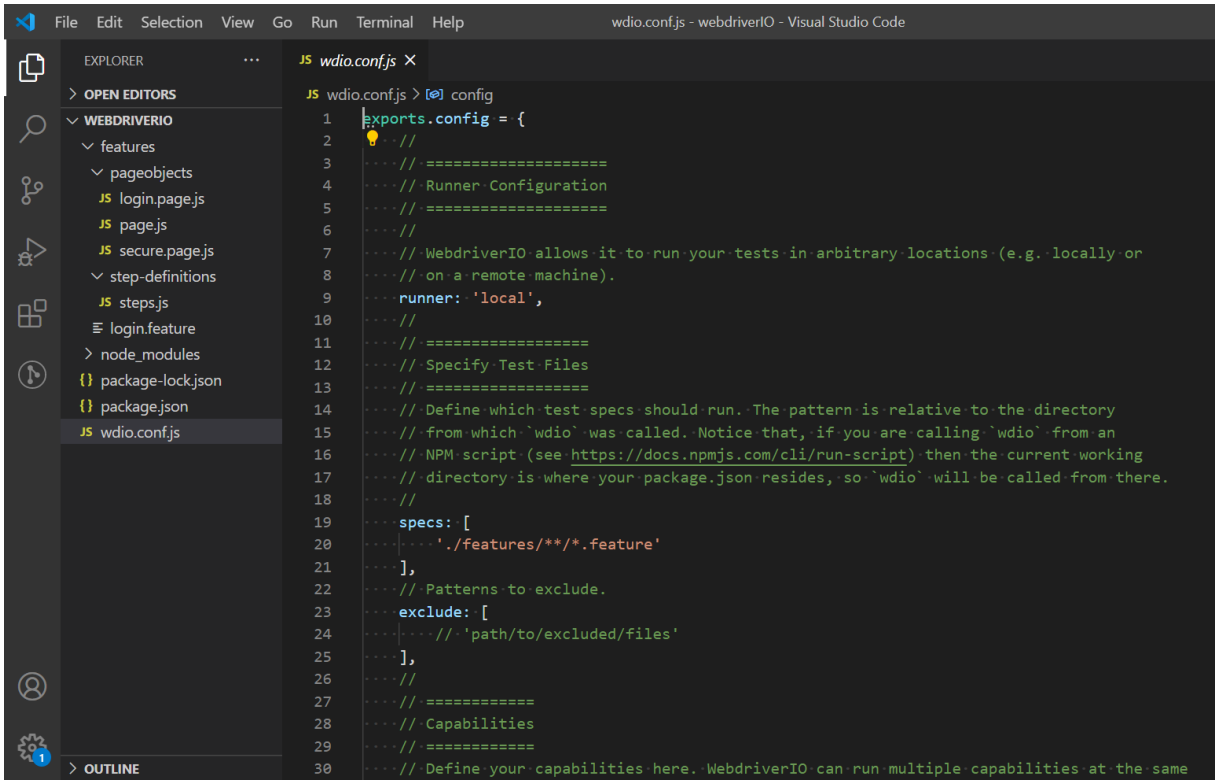



Figura 4-18 : Ficheiro de configurações do *webdriverIO*

## 4.6 *Pipeline* de CI/CD

Atualmente é muito comum recorrer-se a pipelines de CI/CD para evitar processos manuais, por parte da equipa de operações, no que diz respeito à atualização dos ambientes de teste e produção.

A integração contínua (*Continuous Integration* – CI) diz respeito à consolidação regular das alterações no código num repositório compartilhado. Já a entrega contínua/desenvolvimento contínuo (*Continuous Delivery/ Continuous Deployment* – CD) refere-se ao nível de automação das fases finais da *pipeline*. Geralmente, a entrega contínua refere-se ao *deploy* automático das alterações feitas pelos programadores num repositório compartilhado. O desenvolvimento contínuo, refere-se ao *deploy* automático das alterações feitas pelos programadores do repositório compartilhado para o ambiente de produção [57].

O processo automatizado de CI/CD proposto na presente dissertação pode ser observado na Figura 4-19 e a distinção das fases da *pipeline* por *continuous integration*/*continuous delivery*/ *continuous deployment* na Figura 4-20.

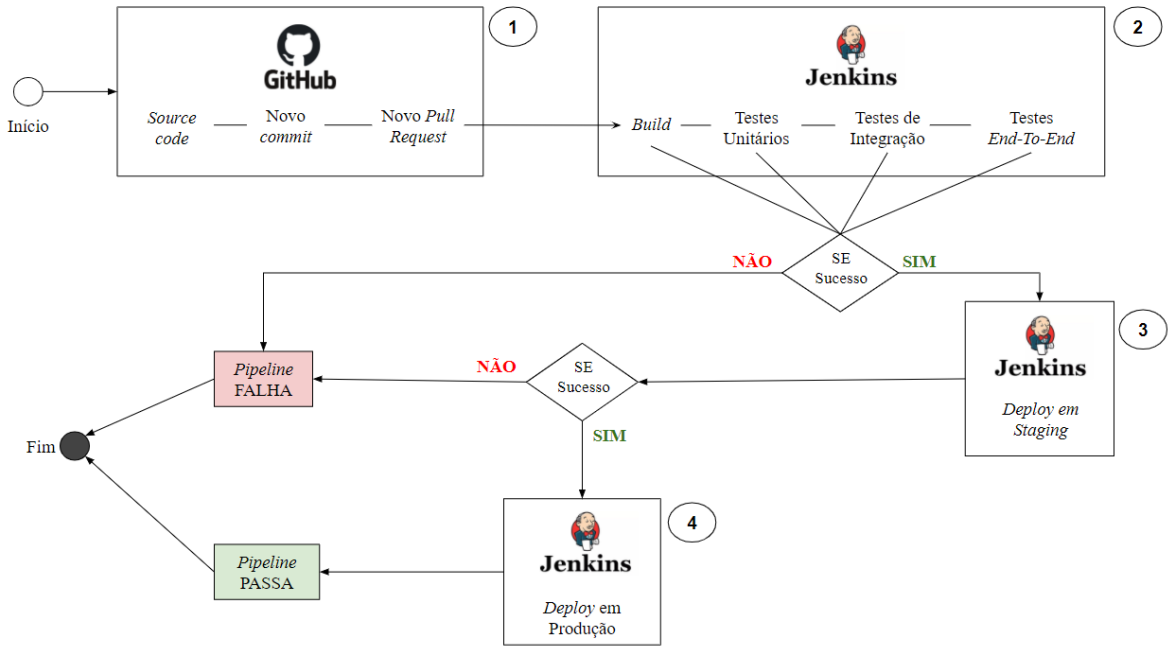



Figura 4-19 : Fluxograma da *pipeline* de CI/CD

O processo de CI/CD começa no código fonte de um sistema e deve estar num repositório compartilhado de versionamento de código como o GitHub. Posteriormente é necessário manter a integração contínua do código fonte e para isso utiliza-se como ferramenta o Jenkins.

A cada nova alteração no código é gerado um novo *commit* e aberto um novo *pull request* no GitHub para ser feita a revisão do código fonte. Cada *pull request* desencadeia uma ação no Jenkins para dar início à execução de todas as fases da *pipeline*.

A primeira fase é o *Build* onde se compila e executa o código fonte. De seguida, executam-se os testes unitários no caso da API e testes de componente no caso da APP. Nestes testes devem-se validar partes isoladas do código responsáveis pelas diferentes funcionalidades. Normalmente restringem-se a uma função ou módulo e são específicos dessa função ou módulo. Depois, na fase seguinte, executam-se os testes de integração nos serviços da API. Nestes testes validam-se as regras de negócio, o corpo (*body*) das respostas recebidas e o código da resposta (*status code*) expectável. Fazem-se pedidos diretos na API que, por sua vez, vão ler/escrever/alterar/eliminar dados diretamente à base de dados. Caso houvesse dependência de serviços externos seriam utilizados *mocks* para simular pedidos a outras APIs.

Posteriormente, executam-se os testes de *end-to-end* no UI da APP. Nestes testes validam-se as interações entre os componentes e as *interfaces* da aplicação. Estas interações despoletam pedidos diretamente na API.

Se todas as fases de testes passarem com sucesso, as alterações são instaladas automaticamente no repositório de *master* durante a fase de *Deploy em Staging*. E se esta fase também terminar com sucesso, as alterações são instaladas automaticamente no ambiente de produção durante a fase *Deploy em Produção* que é a última fase da pipeline de CI/CD.

Caso ocorram falhas em alguma fase da *pipeline*, terão de ser feitas as respetivas correções do(s) problema(s) num novo *commit* que voltar a desencadear uma ação no Jenkins e a executar novamente todas as fases da *pipeline*.

Como se optou por uma solução de *continuous deployment*, assim que todas as fases da *pipeline* forem executadas com sucesso é feito automaticamente o *deploy* com as alterações do último *commit* para o ambiente de produção.

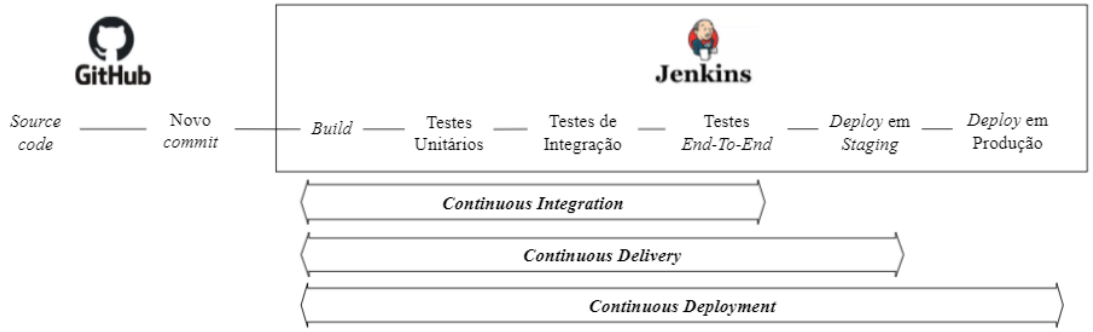



Figura 4-20 : *Pipeline* de CI/CD

# 5. Caso de Estudo

Tendo-se delineado a estratégia de testes no capítulo anterior, agora é necessário colocar em prática a sua implementação. Para isso, neste capítulo, descreve-se um caso de estudo sobre um requisito do negócio e foca-se apenas a implementação da estratégia de automação desse mesmo requisito. Para uma melhor perceção do requisito a automatizar vai, também, ser apresentado um caso de uso com as respetivas *user stories* e os seus critérios de aceitação. Posteriormente, vai ser aplicada a este caso de uso toda a estratégia apresentada no capítulo anterior, detalhando o plano, desenho, implementação, execução e avaliação dos resultados dos testes.

## 5.1 Caso de uso

Os diagramas de casos de uso demonstram as interações entre os utilizadores (atores) com o sistema [38]. Estes diagramas têm como propósito especificar o contexto do sistema, documentar os requisitos funcionais e derivar a criação/implementação dos casos de teste. Devem ser desenvolvidos por analistas em conjunto com os especialistas do negócio [37].
O âmbito do caso de uso escolhido é uma plataforma de comércio eletrónico (*ecommerce)* de produtos biológicos de diversas aldeias vinhateiras do Alto Douro em Portugal. No *frontoffice* deste sistema os *end-users* fazem os pedidos de encomendas dos produtos biológicos, cujos

fornecedores são agricultores das diversas aldeias do Douro. O caso de uso a automatizar encontra-se ilustrado na Figura 5-1.

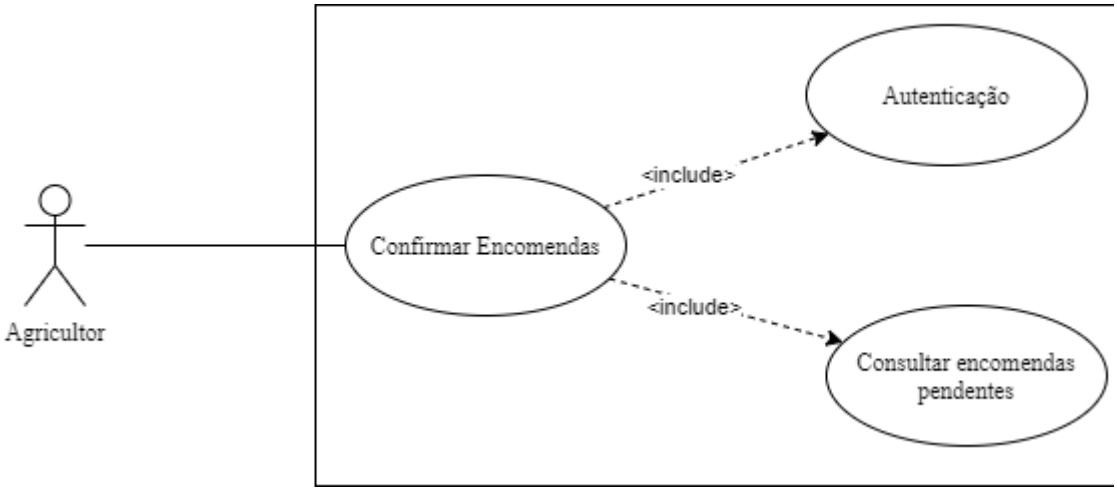



Figura 5-1 : Caso de uso

Temos como ator do caso de uso de confirmar encomendas os agricultores que, no âmbito do caso de uso "Confirmar Encomendas", recorrem às ações autenticação e consultar encomendas pendentes. Finalizado o desenho do caso de uso, o responsável do produto prossegue com a definição das *user stories* e critérios de aceitação do mesmo.

### 5.1.1 *User stories*

As *user stories* introduzidas em ambientes de metodologias ágeis têm como objetivo ajudar as equipas de desenvolvimento a entenderem a descrição dos requisitos do negócio na perspetiva do utilizador final. Já existem alguns formatos distintos para a escrita de *user stories*, mas, o mais comum, é o seguinte [36]:

**ENQUANTO** <determinado utilizador>,

**QUERO** <executar uma ação>,

**PARA QUE** <obtenha um benefício>.

A *user story* deve conseguir responder às seguintes questões [26]:

- Quem? Para quem é que vamos construir algo? Quem é o utilizador?
- O quê? O que é que nós vamos construir? Qual é que é a intenção?
- Porquê? Porque é que nós vamos construir? Qual é o valor que traz para o cliente?

De seguida, descrevem-se as *user stories* para o caso de uso anteriormente apresentado. É sobre estas que se vai delinear toda a estratégia de automação de testes:

***UserStoryID*** **- US1**

**ENQUANTO** agricultor

**EU QUERO** confirmar encomendas pendentes

**PARA QUE** possam ser processadas.

***UserStoryID*** **- US2**

**ENQUANTO** utilizador

**EU QUERO** validar as credenciais de acesso válidas

**PARA QUE** possa entrar no sistema.

***UserStoryID*** **- US3**

**ENQUANTO** utilizador

**EU QUERO** consultar encomendas pendentes

**PARA QUE** possa confirmá-las.

### 5.1.2 Critérios de aceitação

Os critérios de aceitação definem as condições que o sistema tem de cumprir para que seja aceite pelo *end-user*. Devem ser únicos por cada *user story* e definem o comportamento da funcionalidade de acordo com a perspetiva do *end-user*. A escrita destes critérios obriga à identificação de todas as condições de saída, evitando assim resultados inesperados e, por consequência, contribui para garantir a satisfação de todos os envolvidos [27]. [39].

Na Figura 5-2, estão ilustradas as interações entre os componentes da interface que vão responder aos critérios de aceitação abaixo definidos.

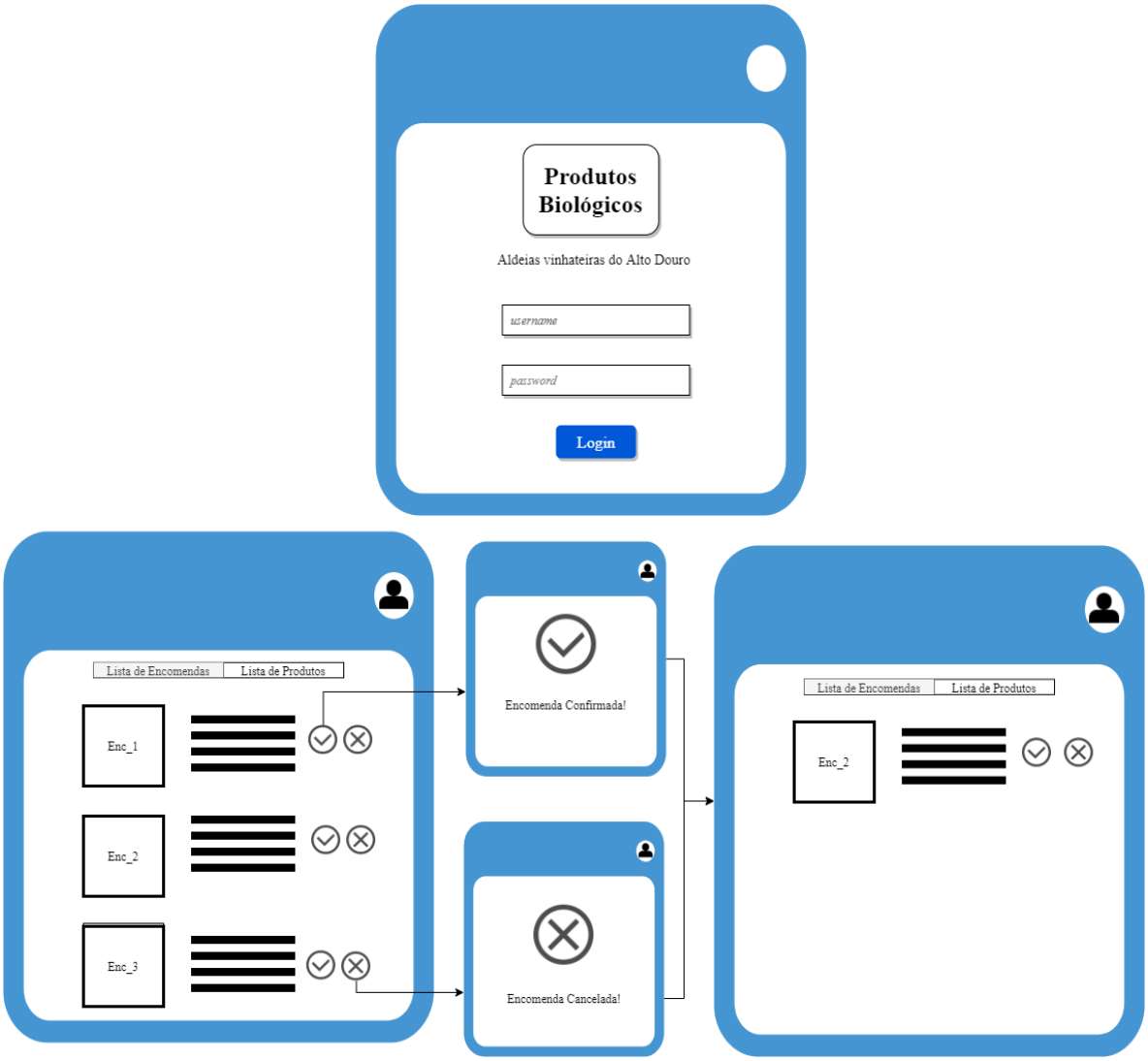



Figura 5-2 : Interações entre os componentes da interface

Para o caso de uso que está a ser apresentado, têm-se os seguintes critérios de aceitação descritos na Tabela 5-1. Todos os critérios de aceitação vão estar associados às *user stories* acima apresentadas.

Tabela 5-1 : Associação dos critérios de aceitação às *user stories*

| USID | Descrição | Critérios de Aceitação |
|---|---|---|
| **US1** | **ENQUANTO** agricultor **EU QUERO** confirmar encomendas pendentes **PARA QUE** possam ser processadas. | • Quando o utilizador efetua a confirmação/cancelamento da encomenda ela desaparece da listagem e deve aparecer uma informação sobre a ação que foi realizada.<br>• Quando o utilizador cancela/confirma uma encomenda não deve conseguir cancelá-la/confirmá-la novamente. |

| | | |
|---|---|---|
| **US2** | **ENQUANTO** utilizador<br>**EU QUERO** ter credenciais de acesso válidas<br>**PARA QUE** possa entrar no sistema. | • Quando efetuado *login* com credenciais inválidas o utilizador deve ser informado.<br>• Quando efetuado *login* com credenciais válidas, o sistema deve redirecionar para a lista de encomendas. |
| **US3** | **ENQUANTO** utilizador<br>**EU QUERO** consultar encomendas pendentes<br>**PARA QUE** possa confirmá-las. | • Quando existe uma encomenda registada na lista de encomendas pendentes de um agricultor deve ser possível:<br>o Ver a informação da encomenda.<br>o Confirmar ou cancelar uma encomenda. |

## 5.2 Plano de testes

Como já foi referido na secção da "Abordagem" do capítulo "Estratégia de testes proposta", o plano de testes deve incluir a matriz de prioridades de requisitos, os critérios de entrada e saída da fase de testes, a matriz de severidade de defeitos e o cronograma das atividades de teste.

### 5.2.1 Cronograma das atividades de teste

Após reunir com os vários elementos do projeto, para obter estimativas do esforço de cada atividade, o *test manager* cria o cronograma das atividades de teste, apresentado na Figura 5-3. Este cronograma contém as datas planeadas de início e fim de cada uma das atividades de testes. Está planeado cerca de 15 dias para o plano de testes, 1 mês para o desenho, implementação e execução e cerca de 5 dias para avaliação dos resultados, antes da data final de passagem para produção *go-live*.

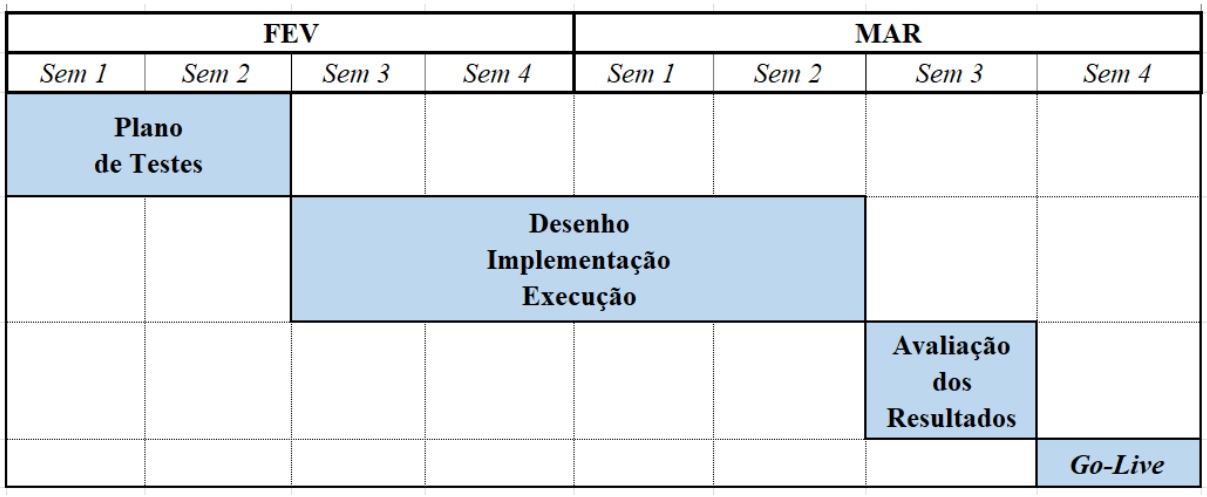



Figura 5-3 : Caso de Estudo - Cronograma das atividades de teste

### 5.2.2 Matriz de prioridades de requisitos

Após o responsável do produto partilhar a lista de requisitos sob forma de *user stories*, o *test manager* constrói a matriz de prioridades de requisitos com as *user stories* por área funcional e por prioridade, como se observa na Tabela 5-2. Para o presente caso de uso é necessário escrever casos de testes para todas as *user stories*, pois todas elas foram classificadas com alta prioridade (P1).

Tabela 5-2 : Caso de Estudo - Matriz de prioridades de requisitos

| Área Funcional | USID | Sumário | Descrição | Prioridade |
|---|---|---|---|---|
| Gestão Encomendas | US1 | Confirmar encomendas | ENQUANTO agricultor<br>EU QUERO confirmar encomendas pendentes<br>PARA QUE possam ser processadas. | P1 |
| | US3 | Consultar encomendas | ENQUANTO agricultor<br>EU QUERO confirmar encomendas pendentes<br>PARA QUE possam ser processadas. | P1 |
| *Login/Logout* | US2 | *Login* | ENQUANTO utilizador, QUERO ter credenciais de acesso válidas PARA QUE possa entrar no sistema. | P1 |

Após definida a matriz de prioridades de requisitos, as *user stories* vão ser adicionadas ao *Backlog* do quadro *Kanban* e, assim que possível, vão avançar para em análise, onde vão ser analisados os critérios de aceitação pelo responsável do produto. Esta transição pode ser observada na Figura 5-4.

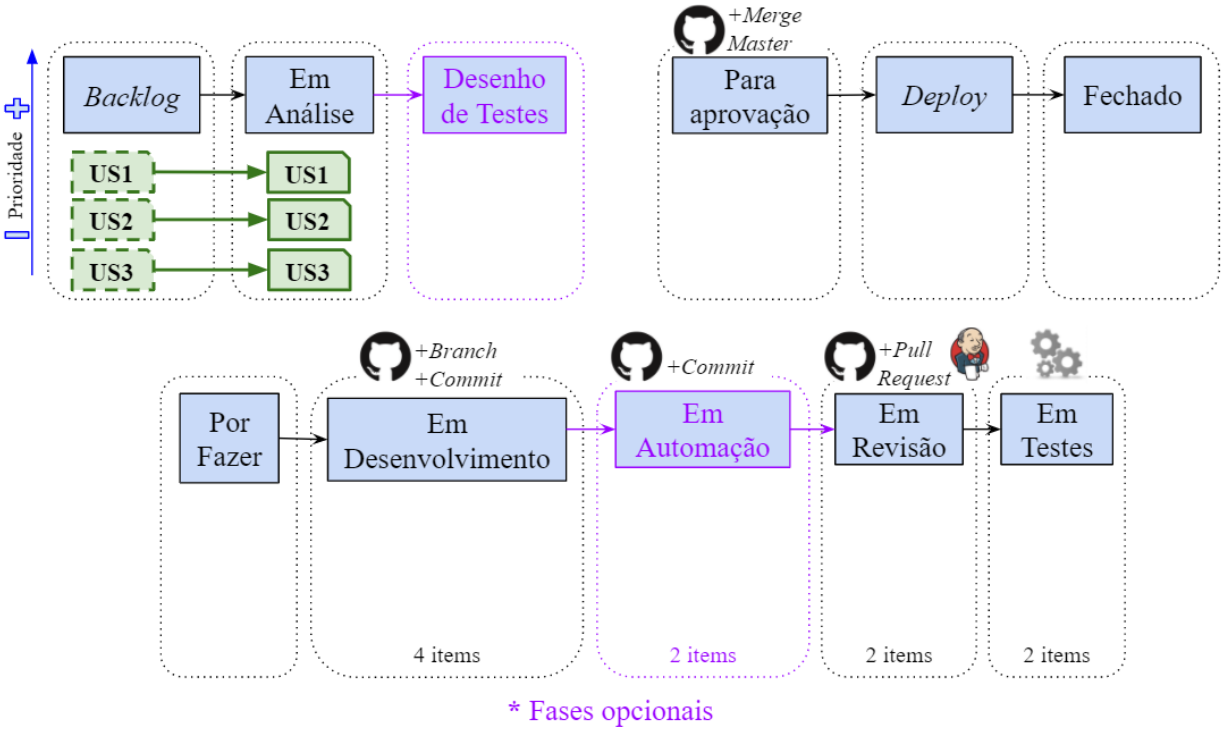



Figura 5-4 : Quadro de *Kanban* na fase de plano de testes

### 5.2.3 Critérios de entrada e saída da fase de testes

Juntamente com o gestor de projeto, o *test manager* inicia a definição dos critérios de entrada e saída da fase de testes. Nestes critérios devem constar todas as tarefas que é preciso alcançar para se iniciar e concluir a fase de testes. Essas tarefas estão representadas na Tabela 5-3. Pode-se observar que o primeiro critério de entrada "Disponibilização da lista de requisitos" já se encontra concluído, pois foi definida a lista de requisitos na seção anterior.

Tabela 5-3 : Caso de Estudo - Critérios de entrada e saída da fase de testes

| Projeto | Critério | Descrição | Progresso | Data de conclusão |
|---|---|---|---|---|
| Projeto 1 | Entrada | Disponibilização da lista de requisitos | Concluído | 20/01/2021 |
| | | Priorização e seleção de casos de testes para os requisitos recebidos | Em progresso | |
| | Saída | 100% dos testes planeados devem ser executados | Não iniciado | |
| | | 100% de cobertura de testes para as *user stories* de prioridade 1. | Não iniciado | |
| | | Número máximo de defeitos abertos com máxima severidade (S1, S2) deve ser igual a 0. | Não iniciado | |

| | | | | |
|---|---|---|---|---|
| | | Número máximo de defeitos abertos de média/baixa severidade (S3, S4) deve ser inferior a 2. | Não iniciado | |

### 5.2.4 Matriz de severidade de defeitos

Na Tabela 5-4 pode ser consultada a matriz de severidade de defeitos para este caso de estudo. Com uma matriz de severidade de defeitos definida, todos os elementos da equipa vão saber exatamente quais as implicações que um defeito tem no sistema. O defeito é avaliado segundo o seu impacto e urgência na resolução, ambos os critérios têm três níveis (Alto, Médio e Baixo) de avaliação cuja sua combinação vai originar a severidade do defeito [105] [106]. Deste modo, um defeito de impacto *vs* urgência:

- **alto/alta** vai ter severidade máxima (**S1**);
- **alto**/**média, alto/baixa** ou **médio/alta** vai ter severidade alta (**S2**);
- **médio/média, médio/baixa**, **baixo/média** ou **baixo/alta** vai ter severidade média (**S3**);
- **baixo/baixa** vai ter severidade mínima (**S4**).

Tabela 5-4 : Caso de Estudo - Matriz de severidade de defeitos

| | | | | |
|---|---|---|---|---|
| **Impacto** | **Alto**<br>Bloqueia completamente a utilização de uma funcionalidade ou até mesmo do sistema como um todo. | **S2** | **S2** | **S1** |
| | **Médio**<br>Bloqueia a utilização de uma funcionalidade, mas, existe outro caminho para se contornar a situação. | **S3** | **S3** | **S2** |
| | **Baixo**<br>Problemas que não afetam funcionalidades. | **S4** | **S3** | **S3** |
| **Severidade** | | **Baixa**<br>Defeito de baixa prioridade. | **Média**<br>O defeito deve ser corrigido assim que seja possível (até 3 dias úteis). | **Alta**<br>O defeito deve ser corrigido imediatamente. |
| | | **Urgência** | | |

## 5.3 Desenho dos casos de teste

O desenho de casos de teste começa assim que o responsável do produto passa as *user stories* para a coluna de desenho de casos de testes no quadro *Kanban*. Esta transição pode ser observada na Figura 5-5. É nesta fase que são escritos os testes de integração ao nível da API e os testes *end-to-end*.

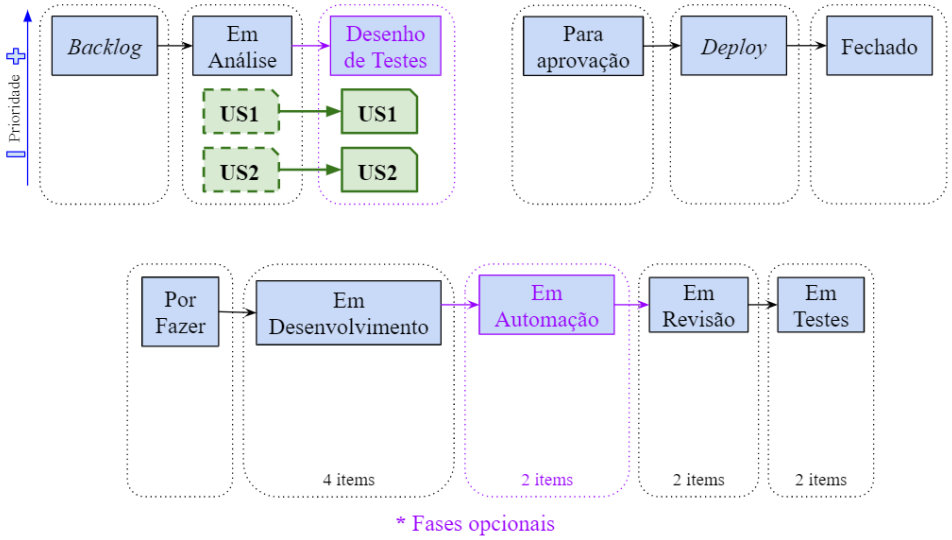



Figura 5-5 : Quadro de *Kanban* na fase de desenho de testes

### 5.3.1 Definição dos casos de teste de integração

Antes de começar o desenho dos casos de testes de integração, necessitamos de reunir algumas informações acerca da API, como os seus *endpoints* e possíveis códigos de erro (*status code*). Deste modo, na Tabela 5-5 estão apresentadas as rotas e verbos dos *endpoints*.

Tabela 5-5 : *Endpoints* disponíveis na API

| Ação | Verbo | Rota |
|---|---|---|
| Confirmar encomenda | PUT | api/Encomendas/{clienteId}/{encomendaId} |
| Eliminar encomenda | DELETE | api/Encomendas/{clienteId}/{encomendaId} |
| Filtrar encomendas | GET | api/Encomendas/{clienteId}/{encomendaId} |
| Visualizar todas as encomendas | GET | api/Encomendas |

O {clienteId} e {encomendaId} são parâmetros que vão ser recebidos aquando da realização do pedido. Todos os *endpoints* da tabela anterior retornam os seguintes códigos de erro:

- **200** em caso de sucesso;
- **400** quando os parâmetros enviados são inválidos ou omissos;

- **401** quando o utilizador não é autorizado;
- **404** no caso em que algum dos parâmetros não existir em sistema.

Conhecidos os *endpoints* da API é, também, necessário saber em que parte da *interface* vão ser chamados, porque são eles que vão popular a informação de alguns componentes do UI ou fazer operações de adições, edições, remoções, entre outras nos eventos despoletados por cliques na *interface*. Na Figura 5-6 ilustra-se o mapeamento entre os *endpoints* que referimos anteriormente e os componentes da *interface*.

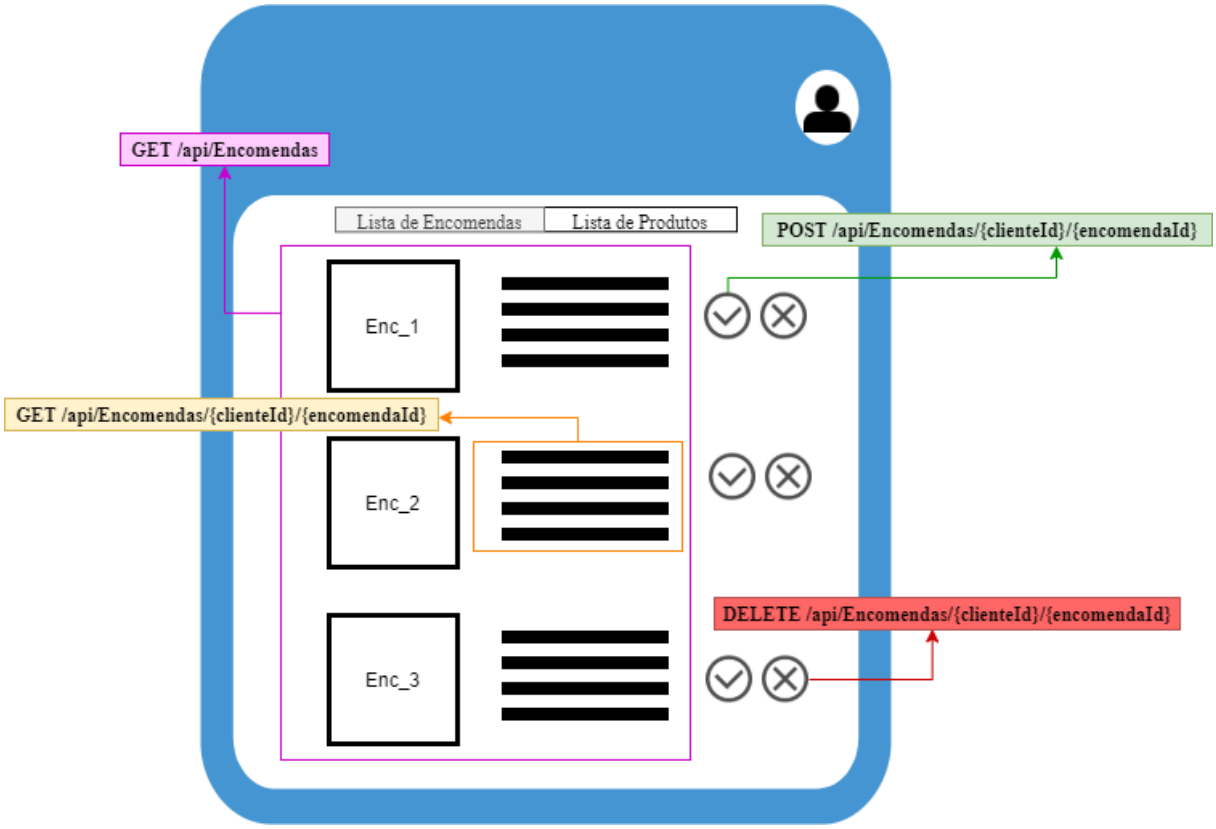



Figura 5-6 : Ligação entre os componentes e os métodos da API

Reunidas todas as informações necessárias acerca da API, pode-se iniciar o desenho dos casos de teste. Os critérios para a escrita dos testes são a total cobertura dos critérios de aceitação das *user stories* definidas no caso de uso apresentado no início do presente capítulo e a cobertura de todas as condições de erro apresentadas anteriormente. Deste modo, os casos de teste de integração que se vai automatizar estão descritos na Tabela 5-6.

Tabela 5-6 : Casos de testes de integração na API

| USID | Pré-Condições | Teste ID | Casos de Teste | *Endpoint* | Dados de Teste | Resultados Esperados |
|---|---|---|---|---|---|---|
| US1 | Ter a encomenda ‘34566’ registada. | TS1 | Confirmar uma encomenda | PUT /Encomendas | ClienteId: 123456789; EncomendaId: ‘34566’. | *Status Code*: 200 |
| | | TS2 | Tentar confirmar uma encomenda que não existe | PUT /Encomendas | ClienteId: 987654321; EncomendaId: ‘33566’. | *Status Code*: 404 { *errors*: [ id:123, *message*: “Encomenda não Encontrada.”]} |
| | | TS3 | Tentar confirmar uma encomenda com um cliente que não existe | PUT /Encomendas | ClienteId: 987654322; EncomendaId: ‘33567’. | *Status Code*: 404 { *errors*: [ id:123, *message*: Cliente não Encontrado. ]} |
| | Ter a encomenda ‘34566’ já confirmada. | TS4 | Tentar confirmar uma encomenda que já foi confirmada anteriormente | PUT /Encomendas | ClienteId: 123456789; EncomendaId: ‘34566’. | *Status Code*: 400 { *errors*: [ id:123, *message*:Encomenda já confirmada. ]} |
| | | TS5 | Tentar enviar uma *string* em vez de apenas números no ID do cliente | PUT /Encomendas | ClienteId: ‘a23456789’; EncomendaId: ‘34566’. | *Status Code*: 400 { *errors*: [ id:123, *message*: O campo do ID do Cliente apenas aceita números. ]} |
| | | TS6 | Tentar enviar um número de caracteres diferente de 9 no ID do cliente | PUT /Encomendas | ClienteId: 12345678; EncomendaId: ‘34566’. | *Status Code*: 400 { *errors*: [ id:123, *message*: O campo do Id do cliente tem que ter 9 caracteres. ]} |
| | | TS7 | Tentar não enviar o ID do cliente (campo obrigatório) | PUT /Encomendas | EncomendaId: ‘34566’. | *Status Code*: 400 { *errors*: [ id:123, |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | *message*: O Id do Cliente é um campo obrigatório. ]} |
| | | TS8 | Tentar não enviar o ID da encomenda (campo obrigatório) | PUT /Encomendas | ClienteId: 123456789.; | *Status Code*: 400<br>{ *errors*: [ id:123,<br>*message*: O Id da Encomenda é um campo obrigatório. ]} |
| | Ter a encomenda ‘34766’ registada. | TS9 | Cancelar uma encomenda | DELETE /Encomendas/ {clienteId}/{encomendaId} | ClienteId: 123321123; EncomendaId: 34766. | *Status Code*: 204 |
| | | TS10 | Tentar cancelar uma encomenda que não existe | DELETE /Encomendas/ {clienteId}/{encomendaId} | ClienteId: 123321123; EncomendaId: 34567. | *Status Code*: 404<br>{ *errors*: [ id:123,<br>*message*: Encomenda não Encontrada. ]} |
| | | TS11 | Tentar cancelar uma encomenda com um ID de cliente que não existe | DELETE /Encomendas/ {clienteId}/{encomendaId} | ClienteId: 456654456; EncomendaId: 34766. | *Status Code*: 404<br>{ *errors*: [ id:123,<br>*message*: Cliente não Encontrado. ]} |
| | Ter a encomenda ‘34766’ já cancelada. | TS12 | Tentar cancelar uma encomenda que já foi cancelada | DELETE /Encomendas/ {clienteId}/{encomendaId} | ClienteId: 123321123; EncomendaId: 34766. | *Status Code*: 400<br>{ *errors*: [ id:123<br>*message*: Encomenda já cancelada. ]} |
| US3 | Ter a encomenda ‘34596’ registada. | TS13 | Listar uma encomenda | GET /Encomendas/ {clienteId}/{encomendaId} | ClienteId: 269962269; EncomendaId: 34596. | *Status Code*: 200<br>Objeto com as informações da encomenda desse cliente:<br>{clienteId: 269962269,<br>encomendaId: 34596,<br>nomeCliente: Sofia Costa,<br>dataEncomenda: 2019/12/20,<br>lista de produtos: [ |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | {prodId: P8547, qnt: 2}, {prodId: P3254, qnt: 1}]} |
| | | TS14 | Tentar listar uma encomenda que não existe | GET /Encomendas/ {clienteId}/{encomendaId} | ClienteId: 269962269; EncomendaId: 35596. | *Status Code*: 404 { *errors*: [ id:123, *message*: Encomenda não Encontrada. ]} |
| | | TS15 | Tentar listar uma encomenda com um cliente que não existe | GET /Encomendas/ {clienteId}/{encomendaId} | ClienteId: 269962266; EncomendaId: 34596. | *Status Code*: 404 { *errors*: [ id:123 *message*: Cliente não Encontrado. ]} |
| | Ter a encomenda '32222' registada. | TS16 | Listar todas as encomendas para confirmação e verificar se alguma encomenda está na lista | GET /Encomendas | | *Status Code:* 200 Lista de todas as encomendas para confirmação. Nesta lista deve constar o/a Cliente: Marta Trindade, cujo Id, é 487487487; com a seguinte Encomenda: 32222. |
| US2 | | TS17 | Autenticação válida | Todos os *endpoints* | *username*: userTestes; *password*: pass1234. | *Status Code*: 200 |
| | | TS18 | Autenticação Inválida | | *username*: userTestes; *password*: pass1235. | *Status Code*: 401 Mensagem: Utilizador não autorizado. |

Como se pode verificar, todas as *user stories* (US1, US2, US3) e os seus respetivos critérios de aceitação foram abrangidas nos testes, bem como todas as condições de erro (200, 400, 401, 404). A cobertura dos critérios de aceitação e das condições de erro da API vai assegurar que existe pelo menos um caso de teste para cada critério de aceitação/condição de erro durante o processo de *deploy* automatizado na pipeline de CI/CD.

Assim, sempre que seja necessária alguma mudança ou correção, estes casos de teste vão ser executados e consegue-se logo ter a perceção se tudo continua a funcionar como expectável.

### 5.3.2 Definição dos casos de teste *end-to-end*

O critério para escrita dos testes *end-to-end* é a cobertura de todos os critérios de aceitação das *user stories* definidas no caso de uso apresentado no início do presente capítulo, tendo em conta que a *interface* já prevê alguns erros. Deste modo, os casos de teste de *end-to-end* a automatizar estão descritos na Tabela 5-7.

Tabela 5-7 : Casos de testes dos testes *end-to-end*

| USID | Teste ID | Caso de Teste | Ações | Dados | Resultados Esperados |
|---|---|---|---|---|---|
| US1 | TS19 | Confirmar uma encomenda | Fazer login | *Username*= userTestes<br>*Password*= pass1234 | |
| | | | Aceder à lista de encomendas pendentes | | |
| | | | Clicar no botão de confirmação de encomenda | ClienteId: 123456789<br>EncomendaId: 34566 | Ao clicar no botão de confirmar desta encomenda irá aparecer uma *pop-up* com o texto "Encomenda Confirmada!". |
| | | | Clicar no (x) da pop-up | | Verificar que a encomenda já não se encontra na listagem. |
| | TS20 | Cancelar uma encomenda | Fazer login | *Username*= userTestes<br>*Password*= pass1234 | |
| | | | Aceder à lista de encomendas pendentes | | |
| | | | Clicar no botão de cancelamento de encomenda | ClienteId: 987654321<br>EncomendaId: 34766 | Ao clicar no botão de cancelar desta encomenda irá aparecer uma *pop-up* com o texto "Encomenda Cancelada!". |

| | | | | | |
|---|---|---|---|---|---|
| | | | Clicar no (x) da pop-up | | Verificar que a encomenda já não se encontra na listagem. |
| US2 | TS21 | Login válido | Aceder à página de login | | |
| | | | Preencher o *username* | *Username* = userTestes | |
| | | | Preencher a *password* | *Password* = pass1234 | |
| | | | Clicar no botão Login | | Verificar que somos redirecionados para a lista de encomendas pendentes. |
| | TS22 | Login Inválido | Aceder à página de login | | |
| | | | Preencher o *username* | *Username* = userTestes | |
| | | | Preencher a *password* com um valor errado | *Password* = pass12345 | |
| | | | Clicar no botão Login | | Verificar que aparece a mensagem: “Credenciais inválidas, por favor tente novamente!” |
| US3 | TS23 | Consultar Encomendas pendentes | Fazer login | *Username*= userTestes<br>*Password*= pass1234 | |
| | | | Aceder à lista de encomendas pendentes | | Verificar que apresenta uma lista de encomendas. |
| | TS24 | Consultar Encomendas vazia | Fazer login | *Username*= userTestes1<br>*Password*= pass12345 | |
| | | | Aceder à lista de encomendas pendentes | | Verificar que aparece a mensagem “Não existem encomendas pendentes!”. |

Como se pode verificar, todas as *user stories* (US1, US2, US3) e seus respetivos critérios de aceitação foram abrangidas nos testes. Neste tipo de testes já foi possível reduzir bastante o número de casos de teste, em comparação com os casos de teste dos testes de integração. Era expectável porque, de acordo com a pirâmide de testes, a quantidade de testes diminui quando se passa de testes de integração para testes *end-to-end.*

Tal como referido nos testes anteriores, a cobertura dos critérios de aceitação, vai assegurar que existe pelo menos um caso de teste para cada critério durante o processo de *deploy* automatizado na pipeline de CI/CD. Assim, sempre que se seja necessária alguma mudança ou correção, estes casos de teste vão ser sempre executados e consegue-se logo ter a perceção se tudo continua a funcionar como expectável.

## 5.4 Implementação e revisão dos testes

A fase de implementação está dependente da conclusão dos critérios de entrada da fase de testes. Como se pode observar na Tabela 5-8, o critério "Priorização e seleção de casos de testes para os requisitos recebidos" que ainda estava em progresso, já se encontra concluído porque já foi terminada a fase de desenho de testes, onde foi feita a priorização e seleção de casos de testes para os requisitos recebidos.

Tabela 5-8 : Caso de Estudo – Concretização dos critérios de entrada da fase de testes

| Projeto | Critério | Descrição | Progresso | Data de conclusão |
|---|---|---|---|---|
| Projeto 1 | Entrada | Disponibilização da lista de requisitos | Concluído | 20/01/2021 |
| | | Priorização e seleção de casos de testes para os requisitos recebidos | Concluído | 30/01/2021 |

Após a concretização dos critérios de entrada, os cartões das *user stories* vão passar para a coluna de por fazer do quadro *Kanban*. Assim, os programadores podem começar o seu desenvolvimento passando os cartões das *user stories* para a coluna em desenvolvimento, criam uma nova *branch* no repositório para fazer *commit* de todo o código desenvolvido e dos respetivos testes unitários. Depois disso, passam o cartão da *user story* para a coluna em automação para os *testers* procederem à escrita dos testes automatizados nessa mesma *branch*. Todas estas transições podem ser observadas na Figura 5-7.

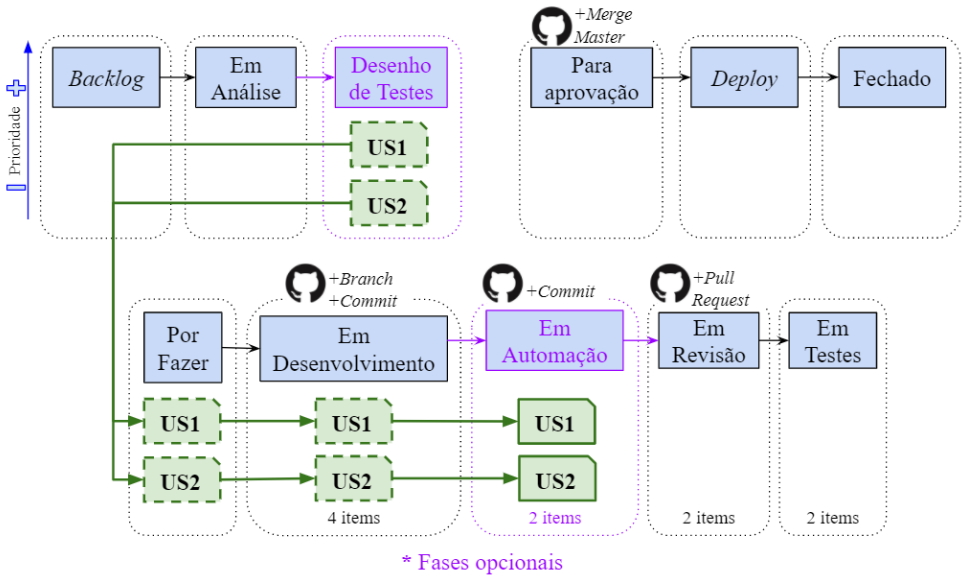



Figura 5-7 : Quadro de *Kanban* na fase de implementação de testes

### 5.4.1 Escrita automatizada dos testes de integração

Os testes de integração realizam vários pedidos à API. A definição de um pedido utilizando o *Supertest* tem a seguinte estrutura, também apresentada na Figura 5-8:

- URL da API
- Verbo HTTP: *Get/Post/Put/Delete*
- Atribuição de *headers* através do método *.set*
- Formação de *queries* através do método *.query*;
- Validação do *status code* esperado através do método *.expect*.

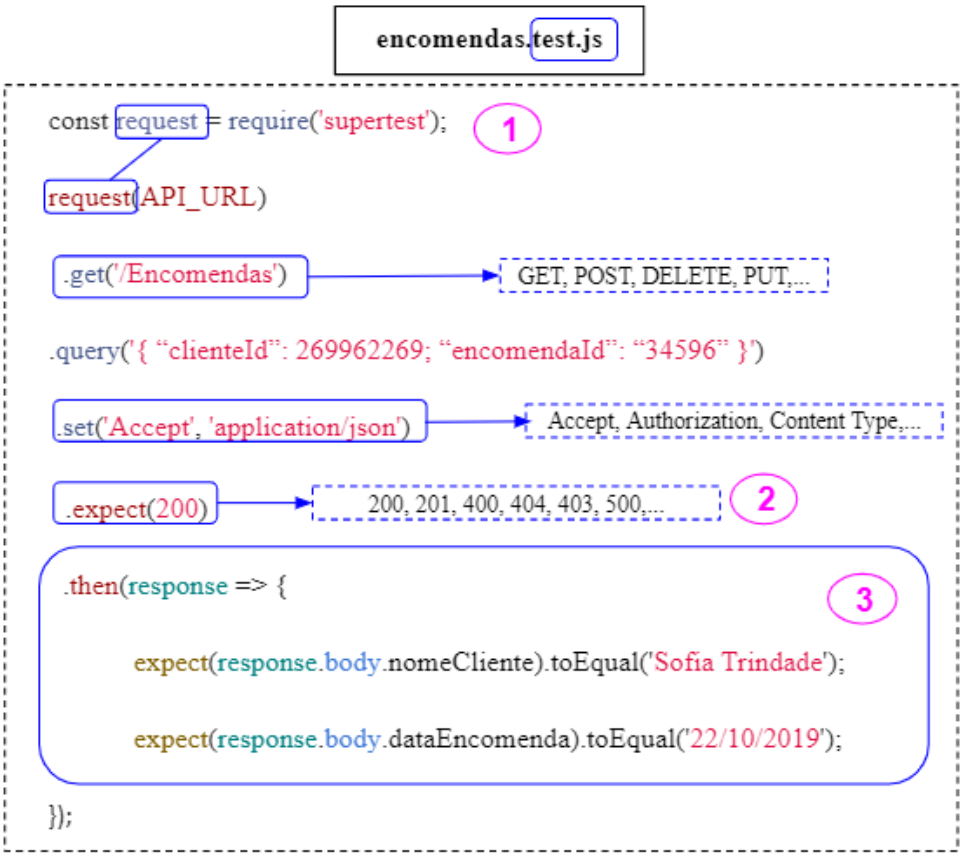



Figura 5-8 : Escrita dos testes de integração com o Supertest

Existem mais métodos que podemos usar. Para isso deveremos consultar a documentação do *supertest* [32] que contém todos os métodos disponíveis.

O *body* das respostas é validado através do *then* utilizando o *expect* do JEST que pode utilizar outros métodos que nos ajudam a fazer melhores validações, como definido no passo 3 da

Figura 5-8. Na documentação do JEST [31] podemos, também, consultar quais os métodos disponíveis.

Para além deste *expect*, o JEST tem mais alguns métodos que vão ser úteis na escrita dos testes, tais como:

- ***describe***: é como se fosse uma *suite* de testes, ou seja, um *describe* pode ter um conjunto de testes. Deste modo, faz sentido a este nível a identificação da *user story* para manter a rastreabilidade entre elas e os respetivos casos de testes. Dentro dos *describes* pode-se ainda colocar outros *describes* com outros conjuntos de testes como se pode visualizar na lateral direita da Figura 5-9;
- ***it* ou *test***: são os nossos testes, cada *it* é um teste, logo os *describes* aglomeram um conjunto de *it.* Cada *it* tem associado o respetivo *TestID*;
- ***beforeAll/beforeEach***: estes dois métodos conseguem executar ações antes de todos os testes (*beforeAll*) ou antes de cada teste (*beforeEach*). Exemplos práticos dos dois casos podem ser:
    - No *beforeAll* iniciar a conexão a uma base de dados;
    - E no *beforeEach* a execução de um *script* de inserção de algum dado requerido nos testes;
- ***afterAll/afterEach***: estes dois métodos (que são o contrário dos anteriores) conseguem executar instruções depois de todos os testes (*afterAll*) ou depois de cada teste (*afterEach*). Exemplos práticos dos dois casos podem ser:
    - No *afterAll* fechar a conexão a uma base de dados;
    - No *afterEach* a execução de um script de remoção de algum dado que apenas foi utilizado nos testes.

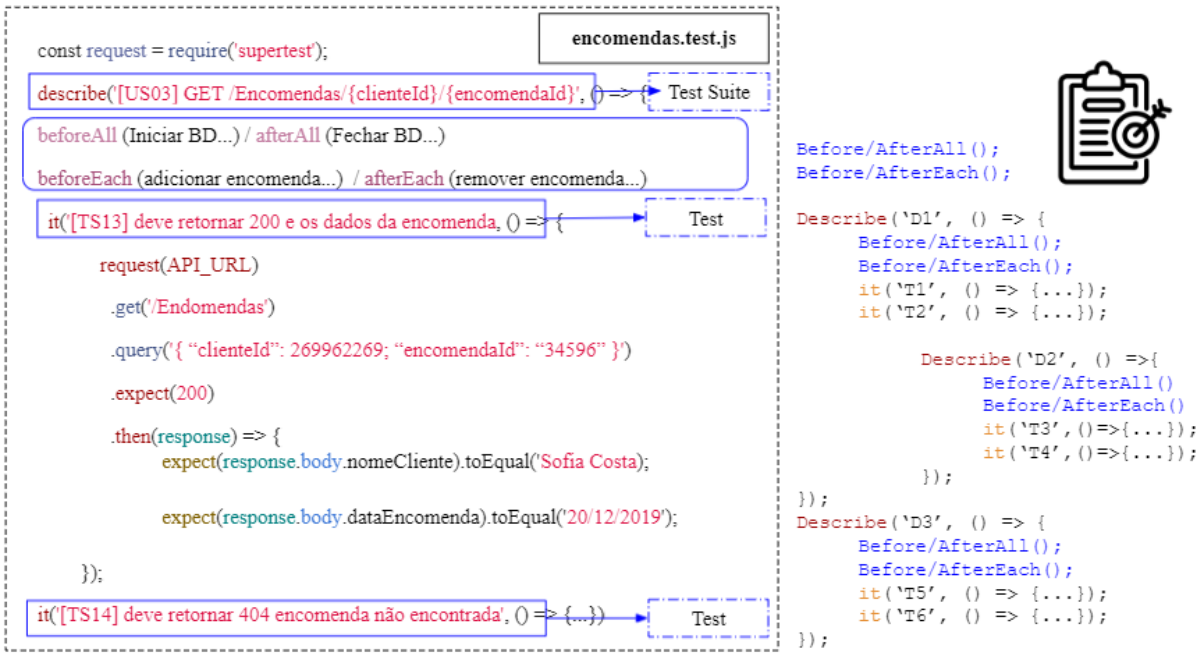



Figura 5-9 : Exemplo de utilização dos métodos do JEST

### 5.4.2 Escrita automatizada dos testes *end-to-end*

Para a escrita dos testes *end-to-end*, existem certos padrões que facilitam a sua interpretação. Estes testes utilizam o paradigma BBD, uma linguagem natural que providencia um nível de abstração do código, chamada de *Gerkin*, e estão organizados segundo o padrão *Page Object* como referido na secção 4.4.3.

Todos os elementos da *interface* onde sejam necessárias interações devem estar identificados com um atributo de marcação específico. Deste modo, consegue-se evitar a dependência dos testes relativamente à estrutura dos elementos HTML que, por vezes, é dinâmica [40], devido à alteração do valor dos atributos, à medida que faz *refresh* da aplicação ou devido à manutenção do código por parte dos programadores. Para identificar os elementos HTML com os quais se pretende interagir, vai ser usado o atributo "*data-test-id*", como ilustra a Figura 5-10.

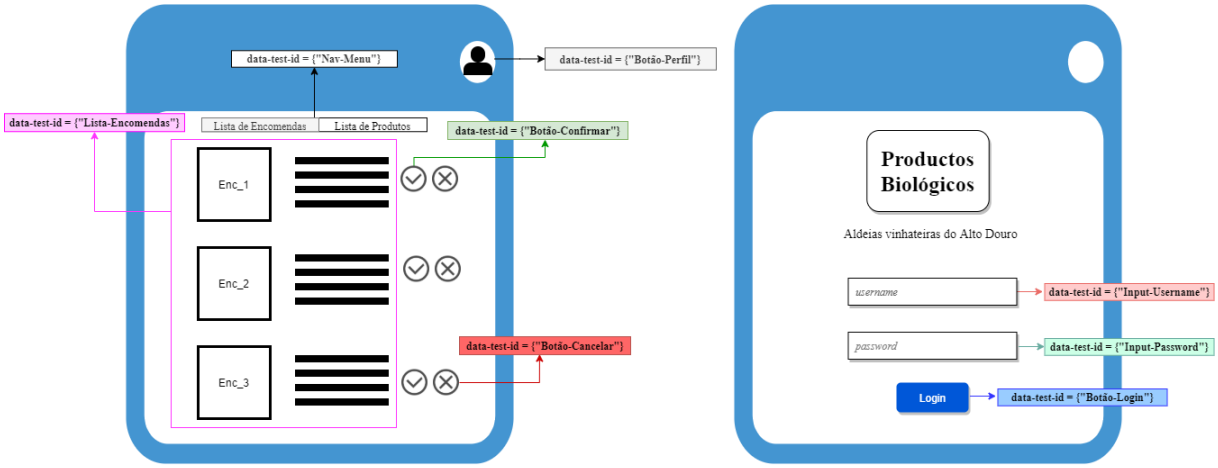



Figura 5-10 : Identificação dos elementos disponíveis na *interface*

Como a *framework* de execução destes testes é o *Cucumber*, para suportar o paradigma do BDD, estes testes vão ter a seguinte estrutura de pastas:

- *Features*: onde estão os ficheiros que vão conter os testes descritos na secção de desenho de casos de teste, mas escritos sob a forma de *Gerkin*;
- *Pages*: que contém os seletores para todos os elementos HTML com os quais se pretende interagir e as funções com os passos necessários a executar na *interface*;
- *Step-definitions:* onde se efetua a ligação entre todas as frases das *features* e as funções das *pages*.

Cada ficheiro de *feature* vai representar uma *user story* e os seus respetivos casos de teste. Para isso, é utilizada uma *tag* (@) com os respetivos IDs. A *tag* das *users sotries* é @REQ_ID e a *tag* dos casos de teste é @TEST_ID. Esta associação permite manter a rastreabilidade das *user stories* e casos de testes definidos na secção de desenho de casos de teste. A Figura 5-11 apresenta os casos de teste do *login* escritos na forma de *Gerkin*. A *feature* é a *user story* do *login*, os *scenarios* são os casos de teste dessa *user story* e os procedimentos do teste estão representados com *given/when/then*. O *scenario outline* define um caso de teste com várias opções de execução que se encontram descritas na zona dos *exemples* (Figura 5-11). Cada coluna dos exemplos vai mapear com as variáveis definidas entre < >.

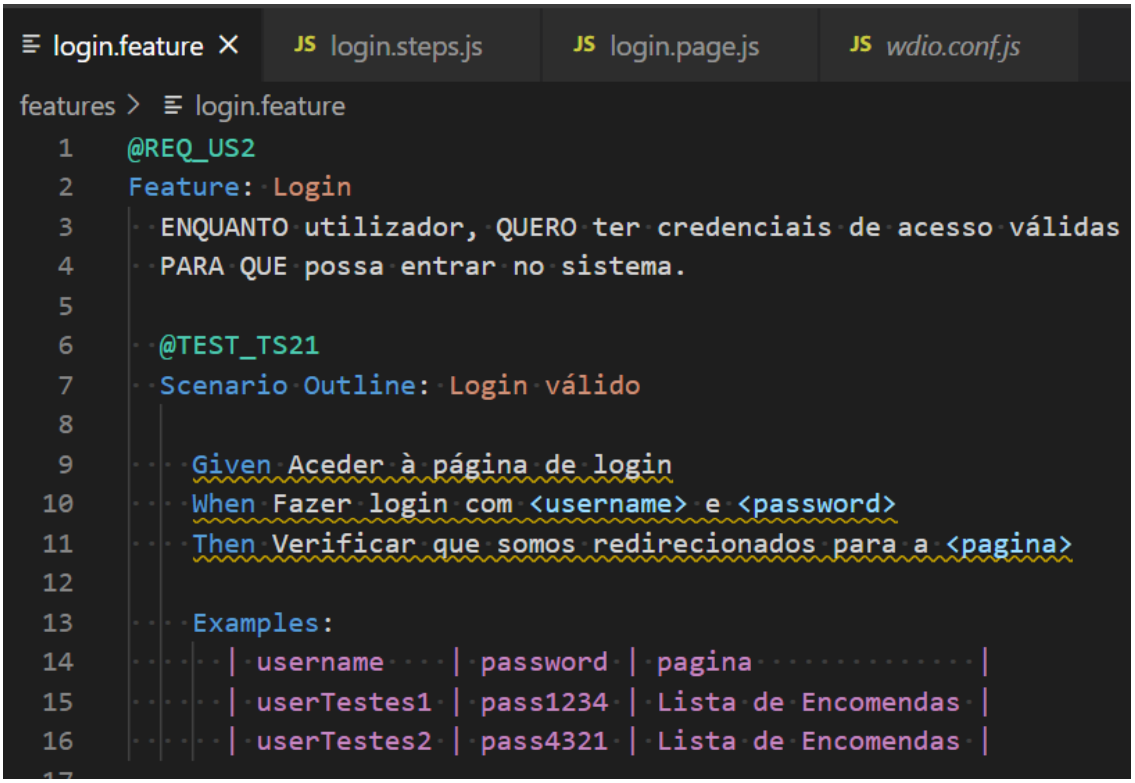



Figura 5-11 : Ficheiro da *feature*

Após a finalização do ficheiro das *features*, vai-se criar o ficheiro das *pages* relativo à página de *login*, identificar os seletores com base nos elementos que foram definidos com o atributo *data-test-id* e, por fim, criar as funções com as instruções dos passos necessários a executar na *interface*. Como por exemplo, na função *login* da Figura 5-12 é necessário preencher o *input* do *username*, o *input* da *password* e clicar no botão *login*. Pode-se ainda reutilizar os métodos de todos os ficheiros das *pages*, fazendo um *extend* do ficheiro requerido para evitar a repetição de código. Na Figura 5-12 é feito um *extend* da página *Page* para utilizar o seu método *open()* que serve para abrir uma determinada página.

login.feature | login.page.js

features > pageobjects > login.page.js > ...

```
6   class LoginPage extends Page {
7       /**
8        * define selectors using getter methods
9        */
10      get inputUsername () { return $('[data-test-id="Input-Username"]') }
11      get inputPassword () { return $('[data-test-id="Input-Password"]') }
12      get btnSubmit () { return $('[data-test-id="Botão-Login"]') }
13
14      /**
15       * a method to encapsule automation code to interact with the page
16       * e.g. to login using username and password
17       */
18      login (username, password) {
19          this.inputUsername.setValue(username);
20          this.inputPassword.setValue(password);
21          this.btnSubmit.click();
22      }
23
24      /**
25       * overwrite specifc options to adapt it to page object
26       */
27      open () {
28          return super.open('login');
29      }
30  }
```

Figura 5-12 : Ficheiro da *page*

Nos *step-definitions* faz-se a ligação entre as *features* e as *pages*. A cada frase definida como *Given*, *When* ou *Then* vai ter atribuída uma função das *pages*, como se pode observar na Figura 5-13. Os valores dos parâmetros a receber nas funções encontram-se no ficheiro de *features* e são identificados por expressões regulares – e.g., "(.*)" permite identificar o início e fim do valor do parâmetro através das aspas e, neste caso, como é utilizado o (.*) vai incluir qualquer tipo de caracter (letras, números, símbolos, caracteres especiais, etc).

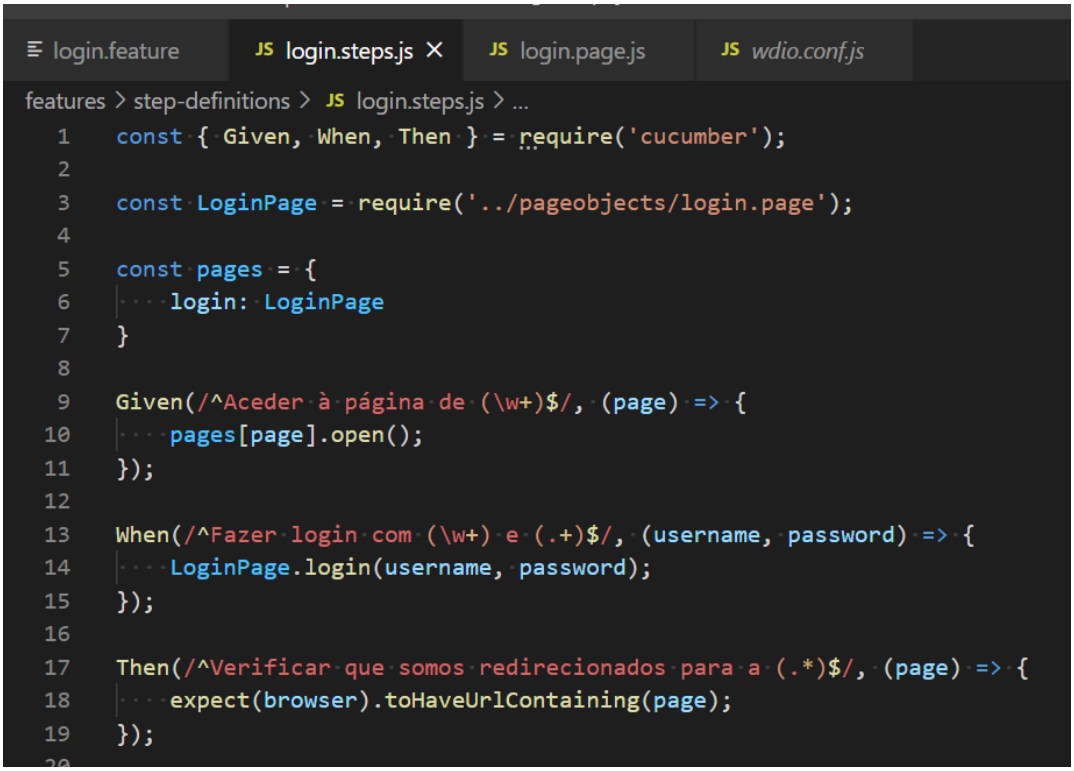



Figura 5-13 : Ficheiro do *step-definition*

Por fim, é necessário incluir o caminho do ficheiro com as *step-definitions* no *require* das *cucumberOpts* do ficheiro de configurações do *webdriverIO wdio.conf.js*, como se observa na Figura 5-14.

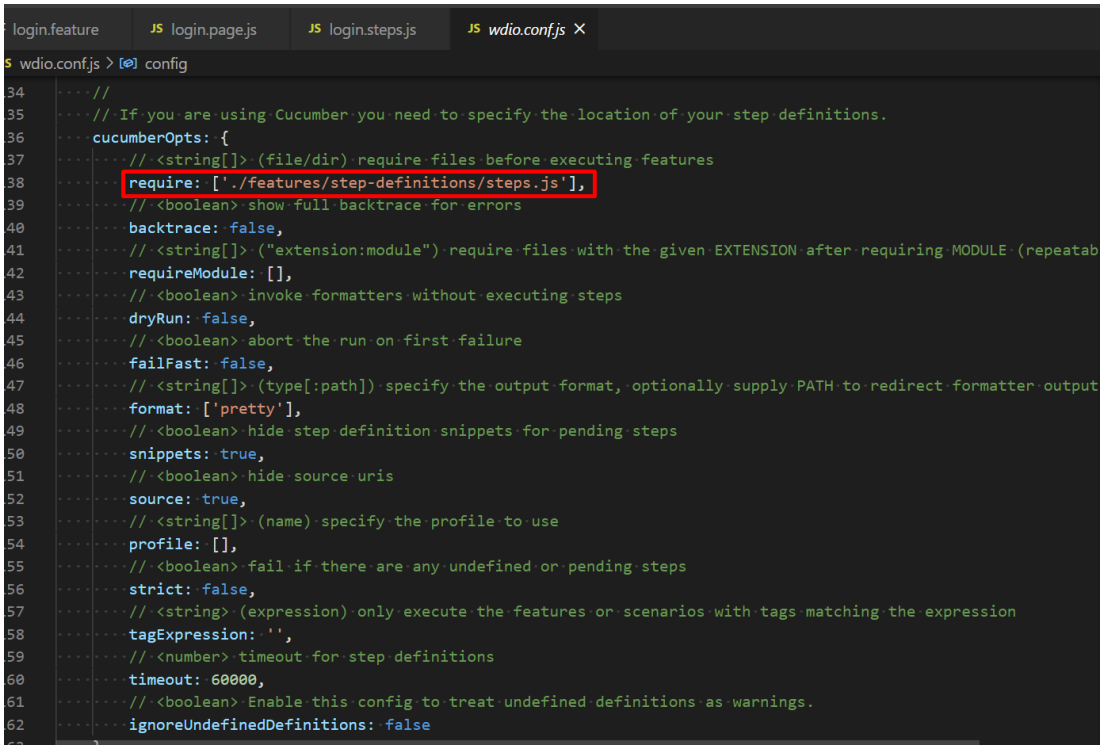



Figura 5-14 : Adição dos *step-definitions* no ficheiro de configuração

### 5.4.3 Revisão de código

Depois de escreverem o código dos testes automatizados, os *testers* fazem *commit* do código da automação de testes e movem no quadro *Kanban* o cartão da *user story* para a coluna em revisão. Posteriormente, é aberto um *pull request* e solicita-se a revisão estática do código a um programador e a outro *tester*. Assim que é aberto um *pull request* é despoletado no *Jenkins* a execução da pipeline de CI (*build*, testes unitários, testes de integração e testes *end-to-end*). Caso sejam identificadas melhorias ou problemas no código pelos revisores no *pull request*, o *tester* pode proceder às respetivas alterações no código (caso concorde com elas), gerando novos *commits* até que todos estejam em conformidade [99]. Esta transição pode ser observada na Figura 5-15.

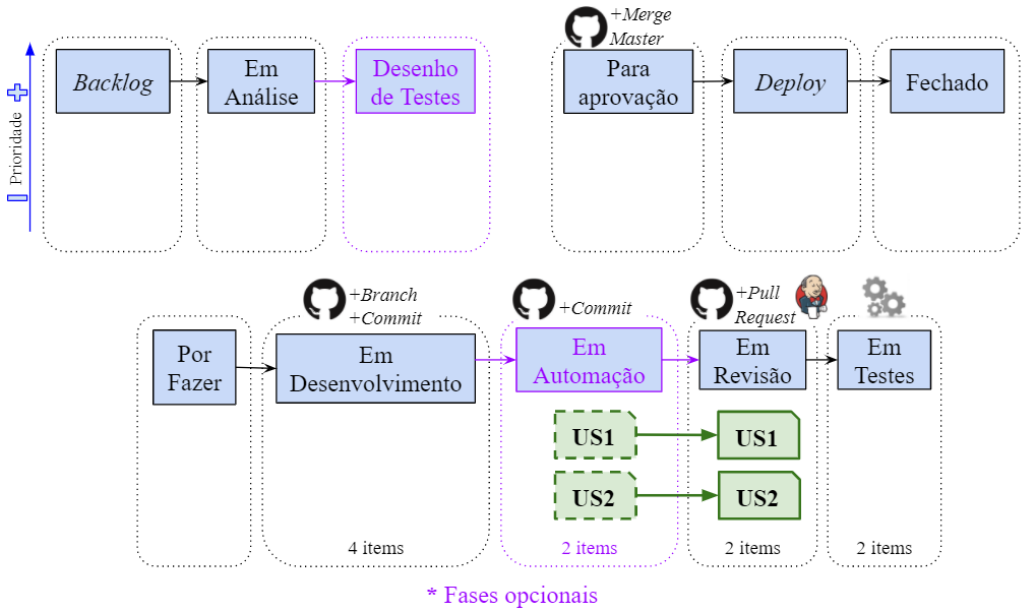



Figura 5-15 : Quadro de *Kanban* na fase de revisão do código dos testes

## 5.5 Execução, gestão de defeitos/alterações e reteste

Depois das revisões do código estarem concluídas, os cartões das *user stories* são transferidos para a coluna em testes do quadro *Kanban*. É nessa etapa que o *tester* executa os casos de teste e faz testes manuais exploratórios. Para isso, é necessário colocar as alterações relativas às *user stories* no ambiente de testes. Esta transição pode ser observada na Figura 5-16.

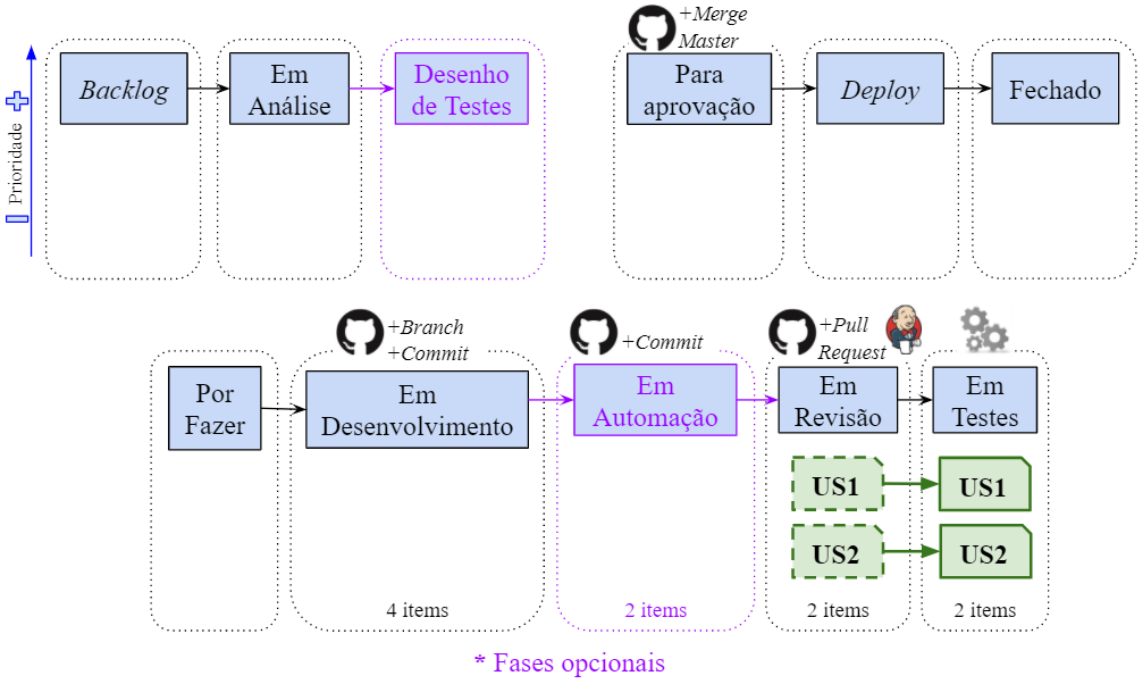



Figura 5-16 : Quadro de *Kanban* na fase de execução de testes

### 5.5.1 Execução dos testes de integração

Para executar os testes de integração, recorre-se ao terminal do sistema operativo e executa-se o comando *npm test*. Este o comando foi definido na instalação das ferramentas (secção 4.5.1), mas também pode ser consultado no ficheiro *package.json* na secção dos *scripts*. Este comando chama o *test runner* – JEST. Executa todos os testes que estiverem dentro dos ficheiros com a nomenclatura "<*nome*>.*test.js*", como se observa na Figura 5-17.


Figura 5-17 : Execução dos testes de integração

Para a execução dos testes de integração na pipeline de CI/CD apenas é necessário colocar os seguintes comandos na parte de execução de testes de integração da pipeline (também ilustrados na Figura 5-18):

- ***cd tests/api*** - para ir para a pasta onde estão os testes de integração;
- ***npm i*** - para instalar o *npm* e todos os packages contidos nas dependências definidas no ficheiro *package.json*;
- ***npm test*** - para executar os testes de integração.

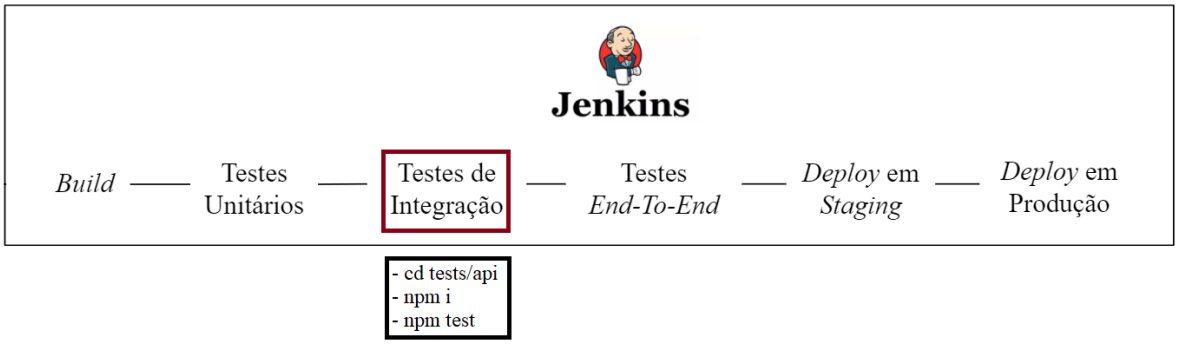



Figura 5-18 : Comandos de execução dos testes de integração na *pipeline* de CI/CD

## 5.5.2 Execução dos testes de *end-to-end*

Para executar os testes de *end-to-end*, é necessário recorrer novamente ao terminal do sistema operativo e executar o comando *npx wdio wdio.conf.js*. Pode também ser adicionado um comando na secção dos *scripts* do ficheiro *package.json* que chame o comando anterior (por exemplo, *e2e-tests : npx wdio wdio.conf.js*). Este comando vai ler o ficheiro *wdio.conf.js* e executar todos os testes que estiverem incluídos no caminho definido na secção *specs*, como se observa na Figura 5-19.

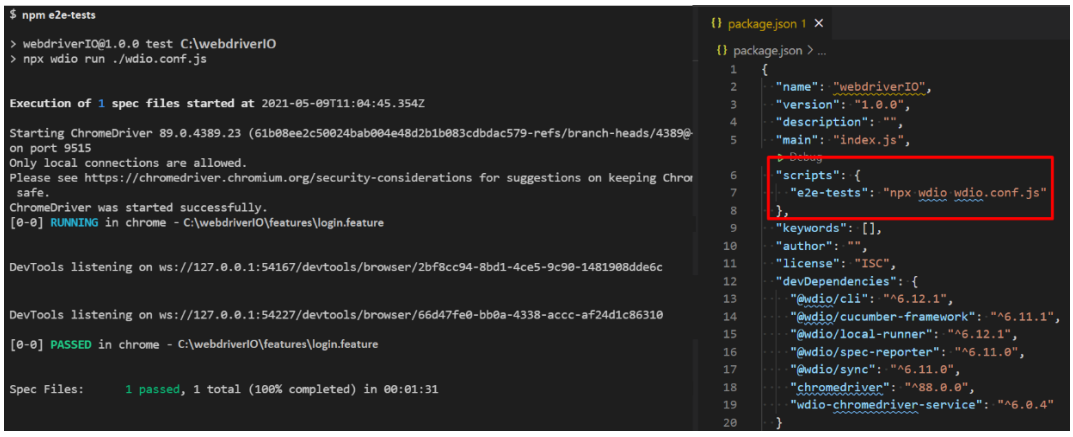



Figura 5-19 : Execução dos testes *end-to-end*

Para a execução destes testes na pipeline de CI/CD, apenas é necessário colocar os seguintes comandos na parte de execução de testes de *end-to-end* (também ilustrados na Figura 5-20):

- ***cd tests/e2e*** - para aceder à pasta onde estão os testes *end-to-end*;
- ***npm i*** - para instalar o *npm* e todos os packages contidos nas dependências definidas no ficheiro *package.json*;
- ***npm run e2e-tests*** - para executar os testes *end-to-end*.

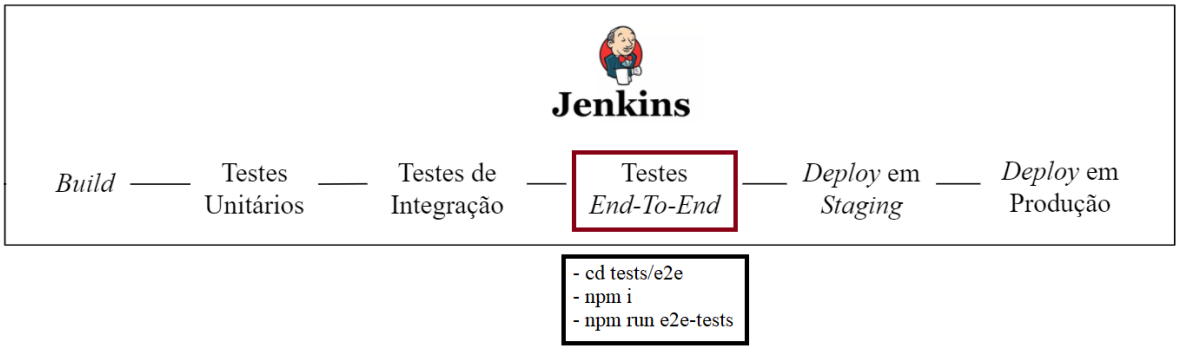



Figura 5-20 : Comandos de execução dos testes de *end-to-end* na pipeline de CI/CD

### 5.5.3 Criação de defeitos

Durante a fase de execução de testes podem ser encontrados defeitos que é necessário reportar. Para isso, é necessário criar um novo cartão do tipo *bug* no *Backlog* do quadro de *Kanban* associado à respetiva *user story*, como se pode observar na Figura 5-21.

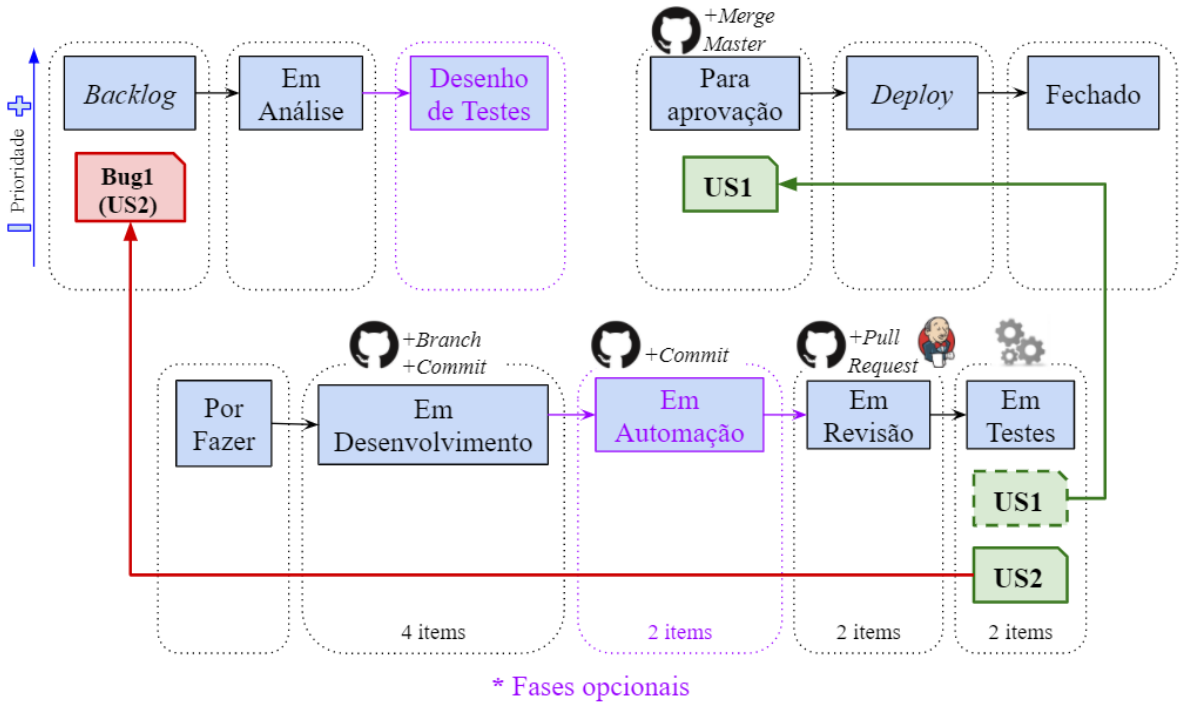



Figura 5-21 : Quadro de *Kanban* durante a criação de um defeito

É necessário fazer a descrição do defeito com todos os detalhes necessários para que outra pessoa o consiga reproduzir e proceder à sua correção. Na Tabela 5-9 podemos observar a descrição do *Bug* 1 associado à *User Story* 2.

Tabela 5-9 : Caso de Uso - Descrição de um defeito

| **Título:** | Não foi possível fazer *login* | | |
|---|---|---|---|
| **Descrição:** | Ao fazer login com o utilizador <*user1234*> está sempre a aparecer a mensagem de erro "Ocorreu algo não esperado, tente mais tarde!" | | |
| **Severidade** | S1 | | |
| **USID** | US2 | | |
| ***Browser*** | Chrome | | |
| **Caso de Teste** | Login válido (TS21) | | |
| **Ação** | **Dados** | **Resultado Esperado** | **Resultado obtido (com evidencias)** |
| Aceder ao sistema | URL= https://produtosdouro.pt | | |
| Aceder à página de login | URL=https://produtosdouro.pt | | |
| Preencher o *username* | *Username* = user1234 | | |
| Preencher a *password* | *Password* = pass1234 | | |
| Clicar no botão Login | | Verificar que somos redirecionados para a lista de encomendas pendentes. | Aparece a mensagem "Ocorreu algo não esperado, tente mais tarde!" |

### 5.5.4 Implementação das alterações para a correção do defeito

Depois de identificado e reportado, o defeito aparece na coluna de *backlog* e dependendo da sua prioridade será endereçado o mais breve possível. Como o defeito tem severidade 1, tem de ser corrigido imediatamente, de acordo com o que está definido na matriz de severidade construída no plano de testes. Deste modo, vai ser movido para a coluna de em análise e o *test manager* vai proceder á sua análise para verificar se é realmente um defeito. Para além disso, analisa se a severidade indicada é a correta e se falta indicar mais alguma informação relevante para que os programadores não tenham dificuldades na sua perceção. Após a análise, move o

defeito para a coluna por fazer e, assim, um programador pode começar a sua correção movendo-o para a coluna em desenvolvimento. De seguida, cria uma *branch* no repositório de código *online* (*GitHub*), faz *commit* de todo o código desenvolvido e abre um *pull request* para o seu código ser revisto por outros, na coluna em revisão. Depois da revisão do código estar concluída, move-se o cartão do defeito para a coluna em testes para que um *tester* possa fazer o seu reteste. Os defeitos não passam pela coluna de desenho de testes porque, normalmente, estão sempre associados a uma *user story* que já passou por esta etapa e onde foram desenhados os seus casos de teste. Este defeito em particular, também não passou na coluna de em automação porque a verificação se o *email* é recebido não é possível automatizar com as ferramentas de automação de testes em uso (sendo esta uma das razões para que estas duas colunas estejam assinaladas como opcionais no quadro *kanban)*. Assim sendo, é um dos casos de testes manuais que tem sempre validação manual. Na Figura 5-22 podem ser observadas todas estas transições.

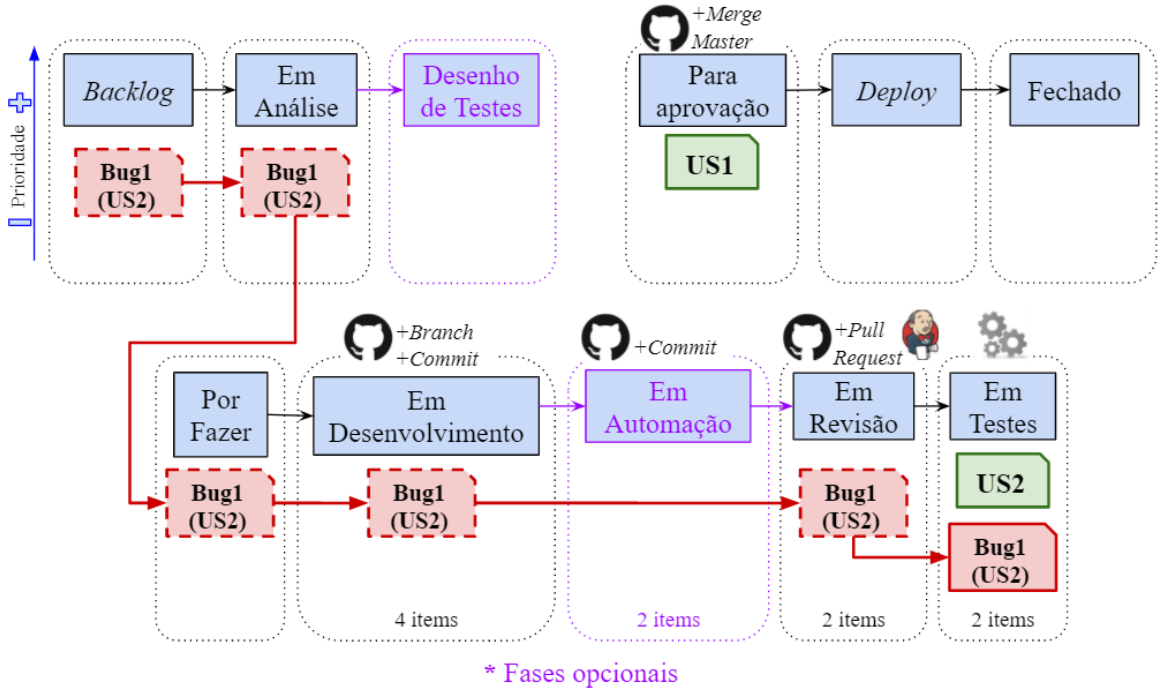



Figura 5-22 : Quadro de *Kanban* durante a correção de um defeito

### 5.5.5 Reteste

No processo de reteste, o *tester* reproduz novamente o caso de teste associado ao defeito e pode fazer manualmente outros testes exploratórios. Nesta fase, é necessário colocar as correções do defeito no ambiente de testes. Se o defeito estiver corrigido, o *tester* passa o defeito para o

estado para aprovação, como demonstrado na Figura 5-23. Se o defeito ainda não estiver corrigido, o *tester* reabre o defeito e este vai passar por todos os estados do quadro *kanban* novamente, até estar realmente corrigido, como se pode verificar na Figura 5-24. Neste último caso, como a *branch* e o *pull request* do defeito continuam abertos, não será necessário criar uma nova *branch* no estado em desenvolvimento, porque se pode reaproveitar a que já está aberta e apenas fazer *commit* das novas alterações.

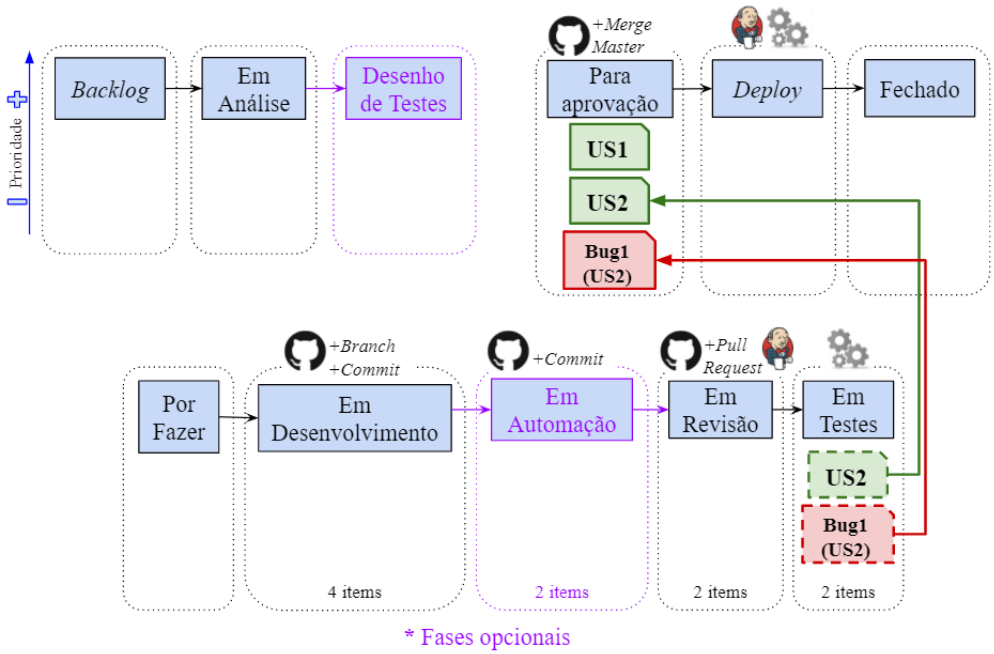



Figura 5-23 : Quadro de *Kanban* durante o reteste bem-sucedido de um defeito

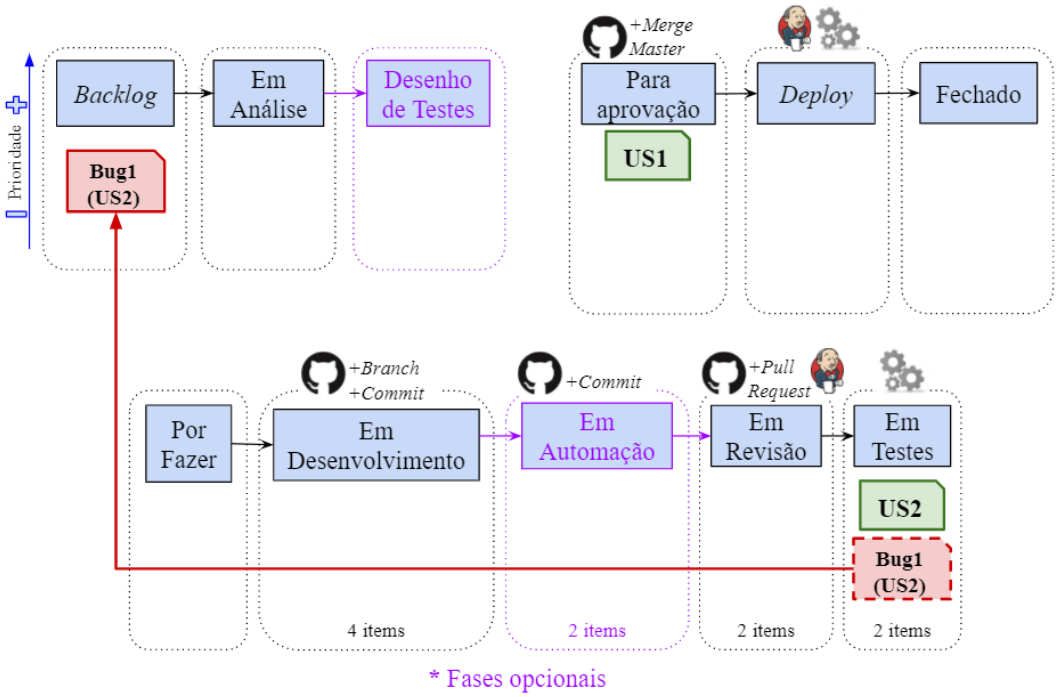



Figura 5-24 : Quadro de *Kanban* durante a reabertura de um defeito

## 5.6 Aprovações e regressões

Depois das *user stories* e respetivos *bugs* passarem para a coluna para aprovação, o *test manager* juntamente com o responsável do produto, vão fazer uma última análise e aprovar o *deploy* das novas funcionalidades ou correções para o ambiente produtivo. Após a análise, e se ambos concordarem na aprovação, um deles necessita de fazer *merge* da respetiva *branch* para *master* e mover os cartões das *user stories/bugs* para a coluna *deploy*.

Depois disto, é despoletado no *Jenkins* a execução da pipeline de CI/CD. Se todas as fases da pipeline passarem com sucesso, vão ser instaladas automaticamente as alterações no repositório *master*, durante a fase de *Deploy em Staging* e de seguida (se a fase anterior terminar com sucesso) vão ser instaladas automaticamente no ambiente de produção. Após o *deploy* em produção estar concluído, deve-se mover os cartões das *user stories/bugs* para o estado fechado. Na Figura 5-25 estão representadas estas transições.

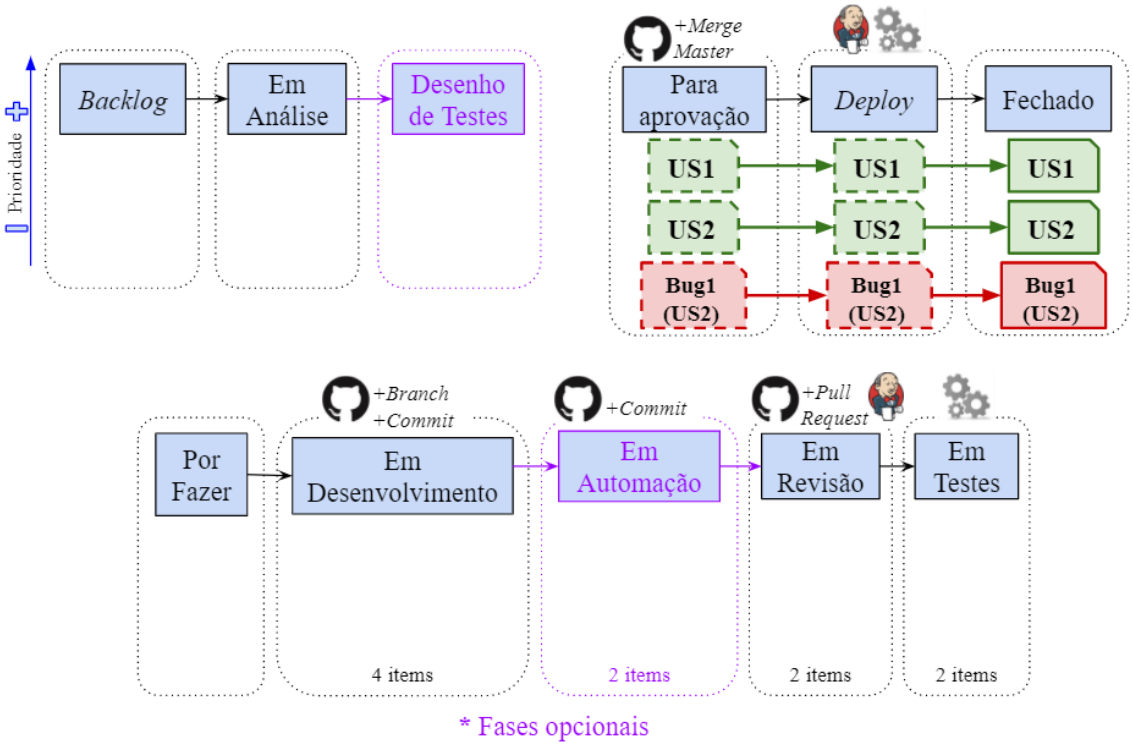



Figura 5-25 : Quadro de *Kanban* durante a fase de aprovação

Caso o *test manager* ou responsável do produto não aprovarem a funcionalidade/correção/alteração, podem movê-la diretamente para a coluna de fechado, mas com a devida justificação (e.g. descontinuada, resolução não aceite, adiada, etc.). Pode-se observar estra transição na Figura 5-26. Isto implica que o cartão não vai passar na coluna de

*deploy*. Por consequente, não se vai fazer *merge* das alterações em *master* e vai-se fechar o *pull request* no *GitHub*. As alterações ficam no histórico de versionamento do *GitHub*, caso mais tarde seja necessário reaproveitar o código.

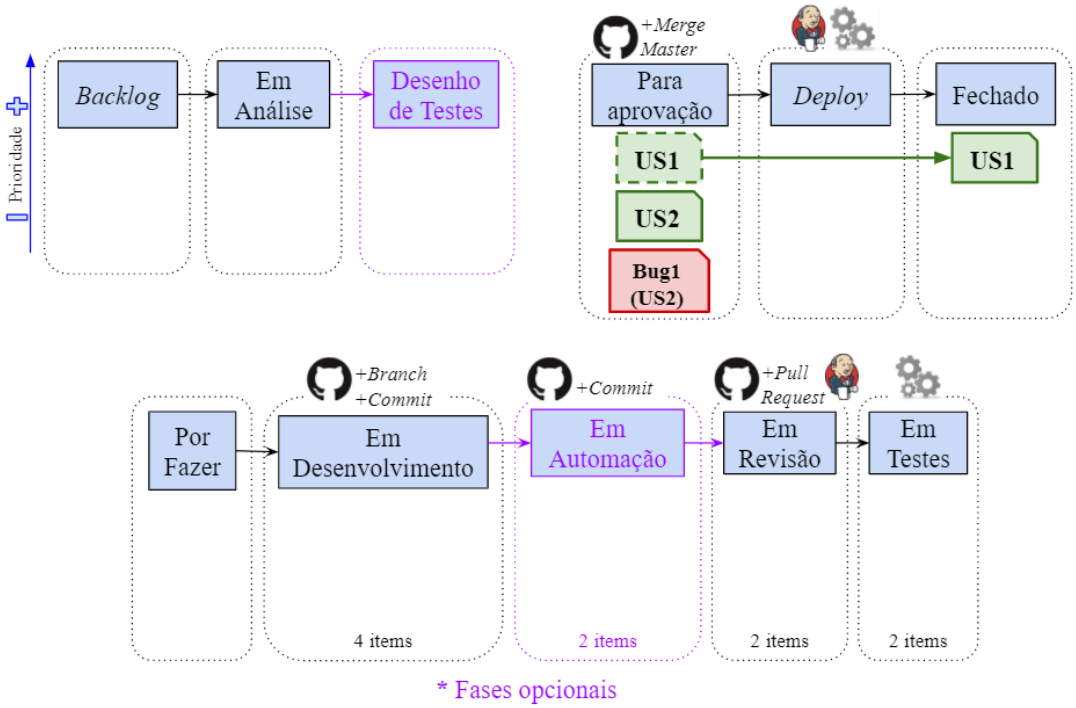



Figura 5-26 : Quadro de *Kanban* durante a fase de não aprovação

## 5.7 Avaliação dos Resultados

Como referido no capítulo anterior, para a avaliação dos resultados constroem-se relatórios que comprovem o estado da qualidade de um projeto, tais como: o relatório de rastreabilidade representado na Tabela 5-10; o relatório de cobertura de testes ilustrado na Figura 5-27; o relatório de defeitos reportados, presente na Tabela 5-11, e o relatório das quantidades de testes ilustrado na Figura 5-28.

Tabela 5-10 : Caso de Estudo - Relatório de Rastreabilidade

| USID | TestID | Caso de Teste | Tipo de Teste | Executado por | Estado | Defeitos |
|---|---|---|---|---|---|---|
| US1 | TS1 | Confirmar uma encomenda | Automatizado - API | Auto | Passou | |
| | TS2 | Tentar confirmar uma encomenda que não existe | Automatizado - API | Auto | Passou | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | TS3 | Tentar confirmar uma encomenda com um cliente que não existe | Automatizado - API | Auto | Passou | |
| | TS4 | Tentar confirmar uma encomenda que já foi confirmada anteriormente | Automatizado - API | Auto | Passou | |
| | TS5 | Tentar enviar uma uma *string* em vez de apenas números no ID do cliente | Automatizado - API | Auto | Passou | |
| | TS6 | Tentar enviar menos de 9 caracteres no ID do cliente | Automatizado - API | Auto | Passou | |
| | TS7 | Tentar não enviar o ID do cliente (campo obrigatório) | Automatizado - API | Auto | Passou | |
| | TS8 | Tentar não enviar o ID da encomenda (campo obrigatório) | Automatizado - API | Auto | Passou | |
| | TS9 | Cancelar uma encomenda | Automatizado - API | Auto | Passou | |
| | TS10 | Tentar cancelar uma encomenda que não existe | Automatizado - API | Auto | Passou | |
| | TS11 | Tentar cancelar uma encomenda com um ID de cliente que não existe | Automatizado - API | Auto | Passou | |
| | TS12 | Tentar cancelar uma encomenda que já foi cancelada | Automatizado - API | Auto | Passou | |
| | TS13 | Listar uma encomenda | Automatizado - API | Auto | Passou | |
| | TS14 | Tentar listar uma encomenda que não existe | Automatizado - API | Auto | Passou | |
| | TS15 | Tentar listar uma encomenda com um cliente que não existe | Automatizado - API | Auto | Passou | |
| | TS16 | Listar todas as encomendas para confirmação e verificar se alguma encomenda está na lista | Automatizado - API | Auto | Passou | |
| | TS17 | Autenticação válida | Automatizado - API | Auto | Passou | |
| | TS18 | Autenticação Inválida | Automatizado - API | Auto | Passou | |
| | TS19 | Confirmar uma encomenda | Automatizado – E2E | Auto | Passou | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | TS20 | Cancelar uma encomenda | Automatizado – E2E | Auto | Passou | |
| US2 | TS21 | Login com credenciais válidas | Automatizado – E2E | Auto | Falhou | Bug 1 (US2) |
| | TS22 | Login com credenciais inválidas | Automatizado – E2E | Auto | Passou | |
| US3 | TS23 | Consultar Encomendas pendentes | Automatizado – E2E | Auto | Passou | |
| | TS24 | Consultar Encomendas vazia | Automatizado – E2E | Auto | Passou | |

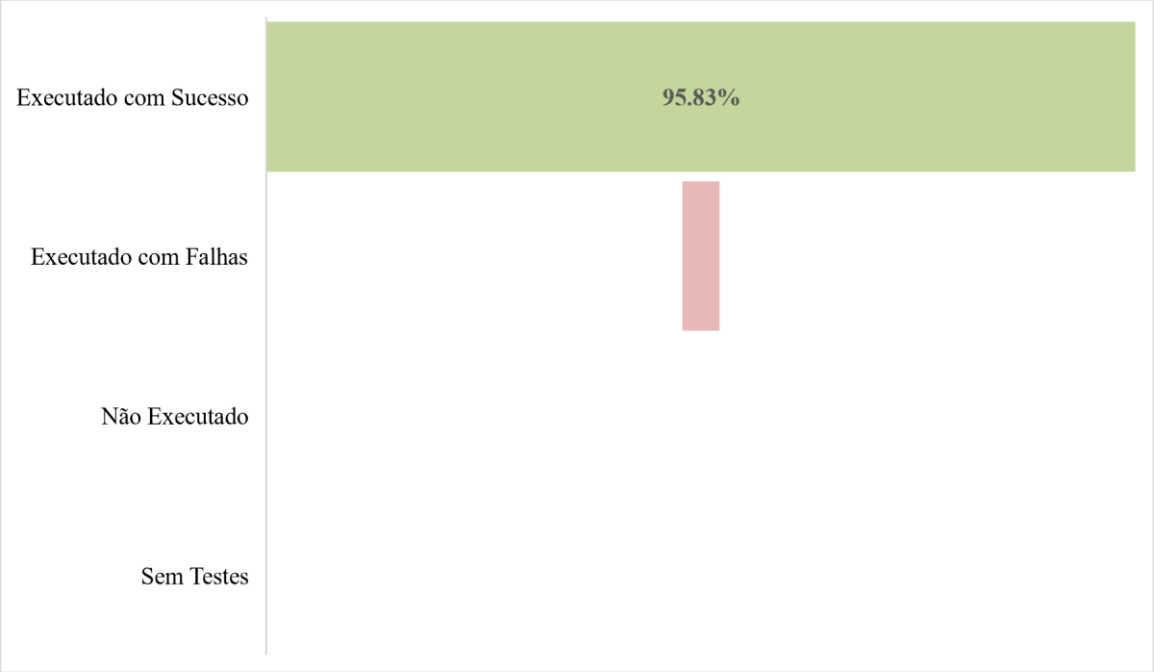



Figura 5-27 : Caso de Estudo - Relatório de Cobertura de testes

Tabela 5-11 : Caso de Estudo - Relatório de defeitos reportados

| | S1 | S2 | S3 | S4 | Total |
|---|---|---|---|---|---|
| Aberto | | | | | **0** |
| Em Análise | | | | | **0** |
| Por fazer | | | | | **0** |
| Em desenvolvimento | | | | | **0** |
| Em Reteste | | | | | **0** |
| Para Aprovação | | | | | **0** |
| *Deploy* | | | | | **0** |
| Fechado | 1 | | | | **1** |
| Total | **1** | **0** | **0** | **0** | **1** |

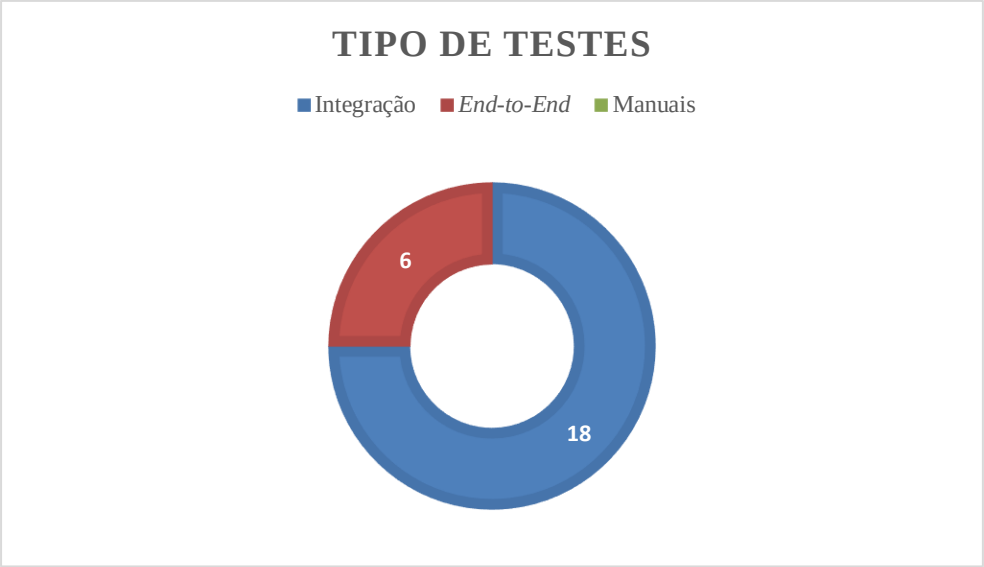



Figura 5-28 : Caso de Estudo - Relatório da quantidade de testes

Depois de avaliarmos os resultados, através da monitorização e métricas, é necessário revisitar os critérios de saída da fase de testes para perceber se todos eles estão concluídos e fechar assim a fase de teses. Na Tabela 5-12 podemos observar que todos os critérios de saída definidos estão concluídos. Visto que não há nenhum teste por executar, a cobertura das *user stories* é 100%, pois cada uma tem pelo menos 1 teste associado e não há nenhum defeito aberto. Pode-se confirmar todos estes dados nos relatórios anteriormente apresentados.

Tabela 5-12 : Caso de Estudo - Conclusão dos critérios de saída da fase de testes

| Projeto | Critério | Descrição | Progresso | Data de conclusão |
|---|---|---|---|---|
| | Saída | 100% dos testes planeados devem ser executados | Concluído | 31/03/2021 |
| | | 100% de cobertura de testes para as *user stories* de prioridade 1. | Concluído | 31/03/2021 |
| | | Número máximo de defeitos abertos com máxima severidade (S1) deve ser igual a 0. | Concluído | 31/03/2021 |
| | | Número máximo de defeitos abertos de média/baixa severidade (S2, S3, S4) deve ser inferior a 2. | Concluído | 31/03/2021 |

# 6. Conclusão

A dependência de suporte tecnológico para todas as atividades é um facto inquestionável. A evolução tecnológica e as constantes mudanças requerem agilidade no desenvolvimento e manutenção do software. A complexidade do software também está a crescer. Por conseguinte, é requerido que o software esteja correto, funcional e pronto a usar. Os testes de software são uma atividade que visa contribuir para a melhoria da qualidade do software. De modo a verificar todos os seus componentes e funcionalidades, é necessário efetuar uma quantidade exaustiva de testes durante o seu desenvolvimento.

A realização de testes de software é uma tarefa complexa, mas fundamental, porque ao detetar falhas antes da entrega do software ao consumidor final, consegue-se poupar o seu tempo de resolução e, por consequência, reduzir custos. Estão também subjacentes requisitos de sistematização desta atividade.

Atualmente, as empresas de software necessitam de validar os seus produtos de forma mais rápida, porque as aplicações passaram a ter ciclos mais curtos devido ao conceito de entrega contínua, cujo objetivo é colocar qualquer tipo de alteração no ambiente de produção de forma segura, rápida e sustentável. Deste modo, as empresas começaram a recorrer a estratégias de automação para tentar reduzir o tempo de execução de testes que eram feitos de forma manual. Porém, existe dificuldade na passagem dos *testers* que realizam apenas testes manuais para a realização de testes automáticos. Tendo em conta a necessidade de entrega contínua e os ambientes ágeis atualmente existentes no ciclo de desenvolvimento de software, quantos mais testes estiverem automatizados, mais rápida vai ser a resposta acerca da qualidade existente nos

sistemas e, por consequência, vai ser possível entregar com mais confiança a nova funcionalidade/alteração ou, até mesmo, a correção de alguma falha.

Com a automatização, também se ganha mais tempo na regressão para fazer testes exploratórios. Ao executar os testes de regressão consegue-se garantir que todas as funcionalidades abrangidas nestes testes continuam a funcionar como expectável depois de alguma modificação no sistema. Se algum teste falhar consegue-se identificar o que falhou e qual seria o resultado esperado, num curto espaço de tempo.

O principal objetivo deste trabalho é propor uma estratégia de automação de testes de software, tendo em conta uma arquitetura básica e passível de aplicação na maioria de sistemas que a usem. Para aplicação da automação é sempre necessário delinear e planear uma estratégia, tendo em conta todas as fases do processo de testes: Planeamento; Controlo e Monitorização; Análise; Desenho; Implementação; Execução e Conclusão.

Neste sentido, efetuou-se uma pesquisa bibliográfica para identificação do estado da arte do tema dos testes de software, onde se explicou alguns padrões ideais para a definição de testes de software, tais como, BBD, *Gherkin* e o padrão *Page Object.*

As estratégias podem variar de acordo com o âmbito projeto, mas, neste caso, pensou-se numa estratégia com base numa arquitetura genérica e passível de aplicação na maioria de sistemas que a usem. A arquitetura adotada pela estratégia de testes proposta é baseada no padrão arquitetural de três camadas (3-*Tiers*), porque é um dos padrões arquiteturais mais usados em aplicações web e de *web services*. Neste padrão o cliente interage com a aplicação e o servidor aplicacional interage com a base de dados.

O âmbito da estratégia passa pela implementação de testes de integração nos serviços da API (camada aplicacional) onde se validam as regras de negócio, o corpo e o código das respostas recebidas. Inclui ainda a implementação de testes *End-to-End* na APP (camada de apresentação) onde se validam interações entre os componentes e as interfaces da aplicação.

Para a implementação dos testes selecionou-se ferramentas/*frameworks open source*, tais como, *Jest* e *Supertest* para os testes de integração na API e *WebdriverIO* e *Cucumber* para os testes *end-to-end* na APP.

Como objetivo do âmbito da estratégia, delineou-se ainda que todos os testes automatizados necessitam de ser executados na pipeline de CI/CD, para que se possa colocar o mais rapidamente possível as alterações no ambiente de produção.

A pipeline de CI/CD é composta pelas seguintes etapas: *Build* (compilação de código), execução de testes unitários, de integração na API e *End-to-End* na APP. Se todas as fases de

testes passarem com sucesso, as alterações vão ser instaladas automaticamente no repositório de *master* e, posteriormente, em produção.

A estratégia proposta tem por base uma metodologia ágil de desenvolvimento para ajudar na gestão dos testes - o *Kanban*. Tem a vantagem de representar um fluxo de trabalho visual, cujo principal objetivo é controlar o trabalho em progresso. Contém os estados que normalmente são usados em metodologias de desenvolvimento ágil (*backlog*, em análise, por fazer, em desenvolvimento, em revisão, em testes, para aprovação, *deploy* e fechado), à exceção dos estados "Desenho de testes" e "Em automação". Foi decidido acrescentar estes estados no quadro de tarefas *Kanban* para reforçar a atividade de automação de testes.

Na abordagem descrevem-se todos os processos e etapas de cada uma das seguintes atividades: plano de testes, desenho, implementação, execução, reteste, gestão de defeitos, gestão de alterações, revisões, aprovações, regressões, monitorização e métricas.

No plano de testes define-se o cronograma das atividades de teste, a matriz de prioridades de requisitos, os critérios de entrada e saída da fase de testes e a matriz de severidade de defeitos.

De seguida, durante a fase de desenho são descritos os casos de teste e cada caso de teste está ligado a uma *user story*. Depois disto, procede-se com a implementação automatizada destes casos de teste, produzindo os *scripts* de automação de testes. Estes *scripts* necessitam de passar pelo conhecido processo de *code review* (revisão de código), por parte de dois outros membros da equipa. Todos estes testes vão ser aproveitados para quando for necessário executar testes de regressão durante a adição/alteração/correção de uma funcionalidade.

Após a implementação dos testes procede-se com a sua execução, desta execução podem surgir defeitos, por isso também foi definido o processo de criação e correção de defeitos, bem como o seu reteste.

Quando todos os casos de teste das *user stories* forem executados com sucesso e todos os defeitos relativos a elas forem corrigidos e retestados, pode-se avançar com a aprovação e com o *deploy* da funcionalidade para o ambiente de produção.

Por fim, faz-se a avaliação dos resultados, analisando todos os relatórios que comprovem o estado da qualidade do projeto. Todas as métricas apresentadas nestes relatórios são monitorizadas ao longo de todas as fases do processo de testes. Durante esta fase analisam-se ainda todos os critérios de saída da fase de testes, para perceber se existem condições para avançar com o *go-live* do projeto.

Após a definição da estratégia e para maior entendimento da mesma, decidiu-se aplicá-la a um caso de uso de um sistema, apresentando as suas *user stories* e critérios de aceitação. Aplicou-se toda a abordagem detalhadamente em cada fase/atividade do processo de testes.

Em jeito de reflexão, pode afirmar-se que os objetivos propostos foram alcançados com sucesso. Conclui-se que, esta é uma estratégia de testes passível de utilização em projetos de várias dimensões, visto que são abordados os aspetos mais pertinentes da definição estratégias de teste, desde a arquitetura do sistema em causa até à execução automática dos testes numa *pipeline* de CI/CD. Explica-se pormenorizadamente todo o fluxo de atividades de testes desde a receção dos requisitos até à sua entrega em ambiente produtivo. Deste modo, qualquer pessoa que tenha interesse na área de automação de testes consegue obter as bases necessárias para proceder á implementação de testes automáticos, criar baterias de testes de regressão automáticas e investir o restante tempo em testes manuais exploratórios, cujo foco são situações que ainda não estão pensadas nem documentadas e de onde podem surgir problemas ainda não identificados, evitando que estes cheguem ao consumidor final.

Testar apenas com testes manuais seria um processo mais demorado. Logo, quanto mais automatização e orientações se tiver num projeto, menor vai ser a possibilidade de uma pessoa não reproduzir algum caso de teste e, também, vai ser menor a fadiga de estar sempre a realizar tarefas repetitivas.

Em termos de trabalho futuro, uma melhoria que poderia ser aplicada a esta estratégia seria a inclusão dos testes unitários neste processo de automatização, visto que a responsabilidade da sua implementação ainda não é algo padronizado, mas tendo em conta a evolução nesta área entende-se que seria uma mais-valia também os *testers* saberem como implementar e manter os testes unitários.

# REFERÊNCIAS


[1] G. E. P. de M. e M. Coutinho, «O papel das ferramentas de automação de testes funcionais em contexto de projetos dinâmicos e complexos», Universidade de Trás-os-Montes e Alto Douro, Vila Real, Portugal, 2016.

[2] J. A. A. Brazeta, «Testes de software no sistema de informação da justiça cabo-verdiana», Universidade de Aveiro, Aveiro, Portugal, 2014.

[3] L. Copeland, A Practitioner's Guide to Software Test Design. Norwood, MA, USA: Artech House, Inc., 2003.

[4] T. Müller et al., «Programa de Certificação de Testador (Tester) de Nível Foundation». International Software Testing Qualifications Board (ISTQB), 2011.

[5] A. P. Mathur, Foundations of Software Testing: Fundamental Algorithms and Techniques. India: Pearson Education, 2007.

[6] E. Dustin, J. Rashka, e J. Paul, Automated software testing: introduction, management, and performance. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1999.

[7] M. Luiz Monteiro Marinho e P. (Orientador) Romero Martins Maciel, «Avaliação de desempenho de processos de testes de software», 2010.

[8] L. Molinari, Inovação e Automação de Testes de Software. Érica, 2010.

[9] O.- www.ondaweb.com.br, «Os 13 principais tipos de Testes de Software!», TargetTrust, 19-Abr-2015. [Em linha]. Disponível em: https://targettrust.com.br/blog/os-13-principais-tipos-de-testes-de-software/. [Acedido: 05-Dez-2017].

[10] «Tipos de teste de software - Guia Completo sobre tipos de testes», Testesdesoftware.com.

[11] «Teste de Software: Teste Funcional e demais tipos de teste», Teste de Software - Veja os tipos de Teste - Testar.me. [Em linha]. Disponível em: https://www.testar.me/teste-de-software. [Acedido: 05-Dez-2017].

[12] E. Machado, «Projeto em contexto de estágio - ALTRAN-Software Quality Assurance (Lisboa)», Instituto Politécnico da Guarda, Lisboa, Portugal, 2017.

[13] D. Graham e M. Fewster, Software Test Automation, 06/1999. Pearson Education (US).

[14] C. Kaner, J. Bach, e B. Pettichord, Lessons Learned in Software Testing: A Context-Driven Approach. Wiley, 2001.

[15] International Software Testing Qualifications Board, “Certified Tester Foundation Level Syllabus.” 11-Nov-2019. [Em linha]. Disponível em: https://www.istqb.org/downloads/send/2-foundation-level-documents/281-istqb-ctfl-syllabus-2018-v3-1.html. [Acedido: 28-Dez-2019].
[16] International Software Testing Qualifications Board, “Standard Glossary of Terms used in Software Testing - Foundation (New) Terms.” Disponível em: https://www.istqb.org/downloads/send/20-istqb-glossary/209-extract-of-terms-used-in-the-foundation-level-syllabus-2018.html. [Acedido: 28-Dez-2019].
[17] «bliki: TestPyramid», *martinfowler.com*. [Em linha]. Disponível em: https://martinfowler.com/bliki/TestPyramid.html. [Acedido: 29-Dez-2019].
[18] Prime Control - Fábrica de Testes, «Automação de Testes na metodologia Continuous Test Automation», www.primecontrol.com.br, 2018. [Em linha]. Disponível em: https://www.primecontrol.com.br/automacao-de-testes/. [Acedido: 29-Dez-2019].
[19] «Fechadura Para Porta de Vidro x Alvenaria JBM FA2», Casa Blindada. [Em linha]. Disponível em: https://www.lojacasablindada.com.br/fechaduraportadevidroalvenaria. [Acedido: 29-Dez-2019].
[20] «Kit Aparente para Porta de Correr 2 m Preto 01 Porta (50 Kg) - Gasometro». [Em linha]. Disponível em: https://www.madeirasgasometro.com.br/kit-aparente-para-porta-de-correr-2m-preto-01-porta-50kg/p. [Acedido: 29-Dez-2019].
[21] «testing pyramid», WatirMelon. [Em linha]. Disponível em: https://watirmelon.blog/tag/testing-pyramid/. [Acedido: 29-Dez-2019].
[22] PJ Stevens, «Enhance Your Testing Pyramids with BDD | HipTest», https://cucumber.io/. [Em linha]. Disponível em: https://cucumber.io/blog/bdd/enhance-your-testing-pyramids-with-bdd/. [Acedido: 29-Dez-2019].
[23] jillre, «C# unit test tutorial - Visual Studio». [Em linha]. Disponível em: https://docs.microsoft.com/en-us/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code. [Acedido: 30-Dez-2019].
[24] jillre, «Unit testing fundamentals - Visual Studio». [Em linha]. Disponível em: https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics. [Acedido: 30-Dez-2019].
[25] M. Cohn, «User Stories and User Story Examples by Mike Cohn», *Mountain Goat Software*. [Em linha]. Disponível em: https://www.mountaingoatsoftware.com/agile/user-stories. [Acedido: 30-Dez-2019].

[26] T. Harbridge, «How to Write Good User Stories in Agile Software Development», Medium, 18-Nov-2019. [Em linha]. Disponível em: https://blog.easyagile.com/how-to-write-good-user-stories-in-agile-software-development-d4b25356b604. [Acedido: 30-Dez-2019].

[27] K. Ravlani, «7 Tips for Writing Acceptance Criteria with Examples - Agile For Growth», Scrum Certification Training and Agile Coaching, 01-Mai-2017. [Em linha]. Disponível em: https://agileforgrowth.com/blog/acceptance-criteria-checklist/. [Acedido: 30-Dez-2019].

[28] G. G. Fernandes, «Pirâmide de testes — uma boa estratégia para automação de testes na prática», Medium, 21-Fev-2018. [Em linha]. Disponível em: https://medium.com/@gianegf/pir%C3%A2mide-de-testes-uma-boa-estrat%C3%A9gia-para-automa%C3%A7%C3%A3o-de-testes-na-pr%C3%A1tica-1d87e64c3a44. [Acedido: 30-Dez-2019].

[29] G. Santos, «Node.js — O que é, por que usar e primeiros passos», Medium, 10-Mar-2017. [Em linha]. Disponível em: https://medium.com/thdesenvolvedores/node-js-o-que-%C3%A9-por-que-usar-e-primeiros-passos-1118f771b889. [Acedido: 04-Jan-2020].

[30] N. js Foundation, «What is npm?», Node.js. [Em linha]. Disponível em: https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/. [Acedido: 04-Jan-2020].

[31] «Jest · Delightful JavaScript Testing». [Em linha]. Disponível em: https://jestjs.io/. [Acedido: 04-Jan-2020].

[32] «supertest», npm. [Em linha]. Disponível em: https://www.npmjs.com/package/supertest. [Acedido: 04-Jan-2020].

[33] «bliki: PageObject», martinfowler.com. [Em linha]. Disponível em: https://martinfowler.com/bliki/PageObject.html. [Acedido: 12-Jan-2020].

[34] «What is Continuous Delivery? - Continuous Delivery». [Em linha]. Disponível em: https://continuousdelivery.com/. [Acedido: 12-Jan-2020].

[35] «User Story Template for Agile | Agile Alliance», 17-Dez-2015. [Em linha]. Disponível em: https://www.agilealliance.org/glossary/user-story-template/. [Acedido: 22-Mar-2020].

[36] «What is User Story?» [Em linha]. Disponível em: https://www.visual-paradigm.com/guide/agile-software-development/what-is-user-story/. [Acedido: 22-Mar-2020].

[37] «What is Use Case Diagram?» [Em linha]. Disponível em: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/. [Acedido: 22-Mar-2020].

[38] «UML Use Case Diagram Tutorial», Lucidchart. [Em linha]. Disponível em: https://www.lucidchart.com/pages/uml-use-case-diagram. [Acedido: 22-Mar-2020].

[39] «Acceptance Criteria: Purposes, Formats, and Best Practices», AltexSoft. [Em linha]. Disponível em: https://www.altexsoft.com/blog/business/acceptance-criteria-purposes-formats-and-best-practices/. [Acedido: 22-Mar-2020].

[40] «Making your UI tests resilient to change». https://kentcdodds.com/blog/making-your-ui-tests-resilient-to-change [Acedido Abr. 07, 2020].

[41] M. Rahman e J. Gao, «Rahman-Gao- A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development-camera ready version». Mai. 15, 2015.

[42] «Introducing BDD», *Dan North & Associates*, Set. 20, 2006. https://dannorth.net/introducing-bdd/ [Acedido Out. 06, 2020].

[43] «bliki: Business Readable DSL», martinfowler.com. https://martinfowler.com/bliki/BusinessReadableDSL.html [Acedido Out. 06, 2020].

[44] «Gherkin Syntax - Cucumber Documentation». https://cucumber.io/docs/gherkin/ [Acedido Out. 06, 2020].

[45] GREENIA, Mark W. History of Computing. Editora Lexikon Services, 2001.

[46] RIOS, Emerson. Documentação de teste de software: Dissecando o padrão IEE 829. Editora Art Studio, 2008.

[47] F. Barbosa e I. Torres, «O TESTE DE SOFTWARE NO MERCADO DE TRABALHO», *TECNOLOGIAS EM PROJEÇÃO*, vol. 2, n. 1, Art. n. 1, Jul. 2011, [Acedido: Out. 24, 2020]. [Em linha]. Disponível em: http://revista.faculdadeprojecao.edu.br/index.php/Projecao4/article/view/82.

[48] Marie-Fabienne Fortin, O Processo de Investigação, Lusodidacta, 04/2000.

[49] Briony J Oates, *Researching information systems and computing*, SAGE. London, 2006.

[50] «Introducing the Software Testing Cupcake (Anti-Pattern)», ThoughtWorks, Jun. 11, 2014. https://www.thoughtworks.com/insights/blog/introducing-software-testing-cupcake-anti-pattern [acedido Out. 26, 2020].

[51] A. Z. Saccol, «Back to basics: understanding the research paradigms and their application in management research», p. 21.

[52] «NET - Definindo a arquitetura de uma aplicação». http://www.macoratti.net/12/11/net_arq1.htm [Acedido Out. 31, 2020].

[53] «Engenharia de Software Moderna (Livro Digital)». https://engsoftmoderna.info [acedido Out. 31, 2020].

[54] «Uma visão geral do HTTP», MDN Web Docs. https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview [Acedido Nov. 01, 2020].

[55] «Integrando aplicações com Web API». https://www.devmedia.com.br/integrando-aplicacoes-com-web-api/32821 [Acedido Nov. 01, 2020].

[56] «http://www.programatic.com.br/canal/blog/programatic/2015/06/20/a-natureza-das-aplicacoes/». http://www.programatic.com.br/canal/blog/programatic/2015/06/20/a-natureza-das-aplicacoes/ [Acedido Nov. 01, 2020].

[57] «O que é CI/CD?» https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd [Acedido Nov. 01, 2020].

[58] ALESSANDRO FERREIRA LEITE, «Conheça os Padrões de Projeto», devmedia, 2005. https://www.devmedia.com.br/conheca-os-padroes-de-projeto/957 [Acedido Ago. 11, 2020].

[59] «Sobre Node.js®». https://nodejs.org/pt-br/about/ [Acedido Out. 11, 2020].

[60] «Getting Started». https://reactjs.org/docs/getting-started.html [Acedido Out. 11, 2020].

[61] «What is .NET Framework?» https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework [Acedido Out. 11, 2020].

[62] «O que é SQL Server?» https://www.portalgsti.com.br/sql-server/sobre/ [Acedido Nov. 11, 2020].

[63] «How to create Test Strategy Document (Sample Template)», *Guru99*. https://www.guru99.com/how-to-create-test-strategy-document.html [Acedido Nov. 11, 2020].

[64] «Web service: o que é, como funciona, para que serve?», Opensoft, Jun. 07, 2016. https://www.opensoft.pt/web-service/ [Acedido Set. 05, 2021].

[65] Kent Beck, Test Driven Development: By Example 1st Edition. Addison-Wesley Professional; 1st edition (November 8, 2002).

[66] Marylene Guedes, «Afinal, o que é TDD?», Abr. 11, 2019. https://www.treinaweb.com.br/blog/afinal-o-que-e-tdd/ [Acedido Nov. 16, 2020].

[67] Rodnney, «Unit Testing – AAA Pattern», Mar. 31, 2016. https://www.thephilocoder.com/unit-testing-aaa-pattern/ [Acedido Nov. 16, 2020].

[68] Nelson Souza e Nelson Souza, «Page Object — Design Pattern», Jan. 08, 2017. https://medium.com/@nelson.souza/page-object-design-pattern-ed5f6374d32d [Acedido Nov. 16, 2020].

[69] The State of Developer Ecosystem in 2020 Infographic», JetBrains: Developer Tools for Professionals and Teams. https://www.jetbrains.com/lp/devecosystem-2020/ [Acedido Nov. 29, 2020].

[70] «Slant - 26 Best programming language to learn for backend developers as of 2020», Slant. https://www.slant.co/topics/7812/~programming-language-to-learn-for-backend-developers [Acedido Nov. 29, 2020].

[71] «10 Best Front End Development Languages in 2020», Buzz Interactive, Jul. 18, 2020. https://www.buzzinteractive.co/best-front-end-development-languages/ [Acedido Nov. 29, 2020].

[72] «Top 7 Web Development Languages To Use In 2020», Designveloper, Ago. 18, 2020. https://www.designveloper.com/blog/web-development-languages-2020 [Acedido Nov. 29, 2020].

[73] «Best Programming Languages to Learn in 2020 (for Job & Future)», Hackr.io. https://hackr.io/blog/best-programming-languages-to-learn-2020-jobs-future [Acedido Nov. 29, 2020].

[74] Archiveddocs, «Chapter 19: Physical Tiers and Deployment». https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658120(v=pandp.10) [Acedido Nov. 30, 2020].

[75] «How To Write Test Strategy Document (With Sample Test Strategy Template) https://www.softwaretestinghelp.com/writing-test-strategy-document-template/ [Acedido Nov. 30, 2020].

[76] « What Is Defect/Bug Life Cycle In Software Testing? Defect Life Cycle Tutorial ». https://www.softwaretestinghelp.com/bug-life-cycle/ [Acedido Nov. 30, 2020].

[77] « Defect/Bug Life Cycle in Software Testing ». https://www.guru99.com/defect-life-cycle.html [Acedido Nov. 30, 2020].

[78] «O Fluxo de Testes © Alexandre Vasconcelos - ppt carregar». https://slideplayer.com.br/slide/372104/ [Acedido Dez. 01, 2020].

[79] M. S. R. de Azevedo, «O papel do gestor de projetos - Portal Gestão». https://www.portal-gestao.com/artigos/7773-o-papel-do-gestor-de-projetos.html [Acedido Dez. 06, 2020].

[80] «Função: Gerenciador de Teste». https://www.cin.ufpe.br/~gta/rup-vc/core.base_rup/roles/rup_test_manager_53B4DA8F.html [Acedido Dez. 06, 2020].

[81] J. Mesh, «Método Kanban: Guia detalhado e 5 modelos prontos para usar». https://blog.trello.com/br/metodo-kanban [Acedido Dez. 06, 2020].

[82] Manoel Pimentel Medeiros, «Exemplos de User Stories», Acedido: Dez. 07, 2020. [Em linha]. Disponível em: https://www.slideshare.net/manoelp/exemplos-de-user-stories.

[83] «Mapeando os Papéis do Desenvolvimento de Software Tradicional para o Scrum», InfoQ. https://www.infoq.com/br/news/2009/03/traditional-roles-to-scrum/ [Acedido Dez. 08, 2020].

[84] «Funções em equipes ágeis: de pequenas a grandes equipes». http://www.ambysoft.com/essays/agileRoles.html [Acedido Dez. 13, 2020].

[85] «The 3 Main Roles in an Agile Team | The Redbooth Blog», Redbooth, Abr. 09, 2018. https://redbooth.com/blog/main-roles-agile-team [Acedido Dez. 13, 2020].

[86] Atlassian, «Kanban - A brief introduction», Atlassian. https://www.atlassian.com/agile/kanban [Acedido Dez. 13, 2020].

[87] «Exact Sales | Método agile: severidade versus prioridade», Severidade x Prioridade na gestão de defeitos em Equipes Ágeis - Exact Sales. https://www.exactsales.com.br/academia-exact-blog/severidade-x-prioridade-com-metodo-agile// [Acedido Dez. 30, 2020].

[88] «Gestão de defeitos: Ferramentas Open Source e melhores práticas na gestão de defeitos», DevMedia. https://www.devmedia.com.br/gestao-de-defeitos-ferramentas-open-source-e-melhores-praticas-na-gestao-de-defeitos/8036 [Acedido Dez. 30, 2020].

[89] «What is a Defect Life Cycle or a Bug lifecycle in software testing?» http://tryqa.com/what-is-a-defect-life-cycle/ [Acedido Dez. 30, 2020].

[90] sharath, «What Is Bug Life Cycle or Defect Life Cycle In Software Testing», Software Testing Material, Dez. 06, 2015. https://www.softwaretestingmaterial.com/bug-life-cycle/ [Acedido Dez. 30, 2020].

[91] «Backlog», Wikipédia, a enciclopédia livre. Out. 20, 2020, Acedido: Jan. 02, 2021. [Em linha]. Disponível em: https://pt.wikipedia.org/w/index.php?title=Backlog&oldid=59629946.

[92] «Klaros-Testmanagement - Professional Test Management Software», Klaros. https://www.klaros-testmanagement.com [Acedido Jan. 16, 2021].

[93] «TestRail QA Metrics - Quality Assurance Metrics», TestRail. https://www.gurock.com/testrail/qa-metrics [Acedido Jan. 16, 2021].

[94] «Xray - Cutting Edge Test Management for Jira», Xray. https://www.getxray.app/ [Acedido Jan. 16, 2021].

[95] «Xray for Jira: The Leading Test Management Tool [2021 Guide]», iDalko, Mai. 14, 2020. https://www.idalko.com/xray-for-jira/ [Acedido Jan. 16, 2021].

[96] «Erro de login windows 10». https://answers.microsoft.com/pt-br/windows/forum/windows_10-windows_store-winpc/erro-de-login-windows-10/31e7f3f7-fe8f-49ce-9521-41883978b0c8 [Acedido Jan. 17, 2021].
[97] «WebdriverIO · Next-gen browser and mobile automation test framework for Node.js». https://webdriver.io/index.html [Acedido Fev. 07, 2021].
[98] «ChromeDriver - WebDriver for Chrome». https://chromedriver.chromium.org/ [Acedido Fev. 07, 2021].
[99] «The Standard of Code Review», eng-practices. https://google.github.io/eng-practices/review/reviewer/standard.html [Acedido Abr. 25, 2021].
[100] «O que é front-end e back-end? | Alura Cursos Online», Alura. https://www.alura.com.br/artigos/o-que-e-front-end-e-back-end [Acedido Jun. 20, 2021].
[101] «API Endpoints - What Are They? Why Do They Matter?», smartbear.com. https://smartbear.com/learn/performance-monitoring/api-endpoints/ [Acedido Jun. 20, 2021].
[102] «What is Firmware? — Definition by Techslang», Techslang — Tech Explained in Simple Terms, Ago. 30, 2019. https://www.techslang.com/definition/what-is-firmware-in-computer/ [Acedido Set. 05, 2021].
[103] «How to Write A Good Bug Report? Tips and Tricks». https://www.softwaretestinghelp.com/how-to-write-good-bug-report/ [Acedido Dez. 08, 2021].
[104] «O custo real da mudança em desenvolvimento de software», iMasters - We are Developers, Nov. 28, 2013. https://imasters.com.br/devsecops/o-custo-real-da-mudanca-em-desenvolvimento-de-software [Acedido Dez. 08, 2021].
[105] «Matriz GUT (Matriz de Priorização)», Ferramentas da Qualidade, Abr. 17, 2019. https://ferramentasdaqualidade.org/matriz-gut-matriz-de-priorizacao/ [Acedido Dez. 08, 2021].
[106] A. Tavares, «Qualidade, Processos e Teste de Software: Como priorizar falhas / defeitos?», Qualidade, Processos e Teste de Software, Jan. 05, 2011. https://qualidadeeteste.blogspot.com/2011/01/como-priorizar-problemas-defeitos.html [Acedido Dez. 08, 2021].