INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Introdução de Desenvolvimento de Software Orientado aos Testes

JOSÉ PEDRO GOMES SAMPAIO
Outubro de 2019

POLITÉCNICO
DO PORTO

Introdução de Desenvolvimento de Software Orientado aos Testes
Estudo de caso

José Sampaio

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Computacionais

Orientador: Professor Doutor Alexandre Bragança

Porto, 11 de Outubro de 2019

Dedicatória

À fabulosa equipa de trabalho que abraçou este desafio. Um enorme obrigado.

Resumo

As empresas têm a necessidade de procurar e adotar técnicas e abordagens para o processo de desenvolvimento de software, a fim de melhorarem as métricas de qualidade, reduzir a taxa de incumprimento, aumentarem a produtividade das equipas e consequentemente, produzir software com qualidade.

Atualmente, na empresa onde este estudo de caso se irá realizar, existe uma abordagem tradicional para o processo de desenvolvimento de software, nomeadamente o *Test-Last Development*. Surge, portanto, a necessidade de explorar e aplicar práticas que melhorem todo o fluxo de desenvolvimento, sendo que a relação entre a qualidade do software e a produtividade das equipas tem impacto no negócio. Com o crescimento exponencial da empresa nos últimos anos e com a necessidade de desenvolver novas funcionalidades sobre software já existente, muitas das soluções caminham para o estado de *legacy*, ficando difíceis de manter e escalar. A escassa existência de ferramentas ou processos que ajudem continuamente no desenho das soluções, torna todo o processo vulnerável a decisões menos corretas, o que irá provocar um consequente impacto na qualidade do software e na produtividade das equipas.

O que este caso de estudo se propõe a realizar é a alteração do processo de desenvolvimento de software atual, com a introdução de metodologias *Test-First*, nomeadamente *Test-Driven Development*, em colaboração com *Behavior-Driven Development*, que possam contribuir para a resolução dos problemas evidenciados.

É esperado que o novo processo de desenvolvimento contribua para o continuo *design* do software, diminuição de erros, aumento da qualidade do software e consequentemente, aumento da confiança no software que está em produção.

Palavras-chave: TDD, Desenvolvimento, Software, Processo, Teste

Abstract

Enterprises need to look for and adopt techniques and approaches to the software development process in order to improve software quality metrics, reduce default rates, increase team productivity and therefore produce quality software.

Currently, in the company where this case study will take place, there is a traditional approach to the software development process, namely Test-Last Development. Therefore, there is a need to explore and apply practices that improve the entire development flow, and the relationship between software quality and team productivity impacts the business. With the company's exponential growth in recent years and the need to develop new functionality over existing software, many of the solutions are moving toward legacy, becoming difficult to maintain and scale. The scarcity of tools or processes that continually aid in the design of solutions makes the whole process vulnerable to less correct decisions, which will have a consequent impact on software quality and team productivity.

What this case study proposes to accomplish is the alteration of the current software development process, with the introduction of Test-First methodologies, namely Test-Driven Development, in collaboration with Behavior-Driven Development, which may contribute to the resolution of problems highlighted.

The new development process is expected to contribute to continued software design, reduced errors, increased software quality and, consequently, increased confidence in the software being produced.

Agradecimentos

Um forte agradecimento a todas as pessoas que contribuíam, desde o início, para a análise, desenho e concretização deste estudo de caso, especialmente à equipa que abraçou este desafio, criando todas as condições para que este projeto fosse levado a bom porto, um profundo obrigado.

Muito obrigado ao orientador Professor Doutor Alexandre Bragança, pois sempre de uma forma muito prestável e construtiva, ajudou continuamente para a realização desta dissertação.

Ao Instituto Superior de Engenharia do Porto e a todos os seus docentes que, durante a Licenciatura em Engenharia Informática e Mestrado em Engenharia de Software, contribuíram para o meu conhecimento e formação. Esta dissertação é o reflexo da aprendizagem académica e profissional.

Por último, agradeço a mim mesmo pela persistência e convicção. Foi uma aprendizagem fantástica durante este longo percurso.

Sem mais, um forte obrigado a todos.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

Capítulo 1

Introdução

No âmbito do Mestrado em Engenharia Informática, esta dissertação foca-se no estudo e aplicação da metodologia Test-Driven Development (TDD, Desenvolvimento Orientado aos Testes) em colaboração com Behavior-Driven Development (BDD, Desenvolvimento orientado ao comportamento) em contexto empresarial. Neste capítulo é apresentada a contextualização do problema, os principais objetivos, a análise de valor, a abordagem a seguir e por último, a estrutura do documento.

1.1 Contexto

O presente estudo de caso será aplicado numa empresa de desenvolvimento de *software*, onde existe a necessidade de adotar novas metodologias para melhorar o processo de desenvolvimento e consequentemente, os seus produtos finais.

1.1.1 Empresa

A empresa onde este estudo de caso se irá realizar é uma empresa de desenvolvimento de *software*. O seu negócio opera sobre uma plataforma *online*, que procura estar sempre disponível, cem por cento operacional. Atualmente, a plataforma opera com inúmeros clientes, destacando-se como o meio de interacção principal, tendo assim um enorme impacto no negócio e por consequência no bom funcionamento da plataforma.

1.1.2 Enquadramento

Como uma das ambições da empresa é a construção de soluções robustas e escaláveis, existe a necessidade de adotar e procurar novas metodologias para optimizar e facilitar o processo de desenvolvimento de *software*, e como consequência, aumentar a qualidade das soluções e produtos.

A área de negócio da empresa depende do *software* que é produzido, e tal como todas as empresas que dispõem os seus serviços *online*, falhas no *software*, indisponibilidade, erros ou incumprimento de prazos para entrega de iniciativas podem dar origem a perdas significativas, o que afecta diretamente a área financeira. Assim sendo, o *software* produzido tem de ser fiável, seguro e escalável.

Após uma análise conjunta com a empresa sobre o estado atual do processo de desenvolvimento, concluiu-se que o resultado pode ser mais satisfatório com a introdução de novas metodologias de trabalho, inferindo-se que existe a necessidade de melhorar o processo de desenvolvimento de *software* que existe atualmente.

Este estudo de caso enquadra-se nesta necessidade e tem como objetivo dar resposta com base nas boas práticas de Engenharia.

1.1.3 Test-Driven Development

O paradigma do *Test-Driven Development* substitui o fluxo tradicional que assenta em codificar e testar (Beck 2003). É uma estratégia ágil de desenvolvimento de *software* que aborda o desenho da solução e os respetivos testes. Envolve a escrita de testes automatizados de uma maneira iterativa *Test-First*, que consiste em desenhar o teste primeiro e à *posteriori*, escrever a implementação. O processo passa por inverter o ciclo básico de desenvolvimento de código mais comum, desenvolver e testar, que após reflexões sobre este ciclo, pode-se considerar que não é a abordagem mais eficaz (Beck 2003).

As premissas do TDD apresentam-se como soluções viáveis ao problema a resolver, sendo a escolha do TDD justificada por vários fatores que servem os propósitos deste estudo de caso. A interpretação dos requisitos é fundamental para a execução da implementação, e seguindo o modelo do TDD, que passa por interpretar esses requisitos numa primeira fase, escrever os testes que os satisfazem, implementar e seguidamente proceder ao *refactor* sobre o código escrito, de forma cíclica, leva a que se possa chegar a uma solução mais rapidamente, focando no problema a resolver e na construção de soluções com um bom desenho.

Na secção 2.6.1 será detalhado o TDD de uma forma mais aprofundada.

1.1.4 Behavior-Driven Development

O desenvolvimento orientado ao comportamento (BDD) é uma abordagem de desenvolvimento ágil (*Introducing BDD* 2006). É uma técnica de especificação que certifica automaticamente que todos os requisitos funcionais são tratados adequadamente pelo código-fonte (De Carvalho e De Campos 2006), que utiliza uma linguagem comum, permitindo a que todos os envolvidos de diferentes áreas comuniquem facilmente através da mesma linguagem.

Considerando as premissas teóricas do BDD, os requisitos irão ser compreendidos de uma forma mais clara, permitindo garantir que o comportamento da aplicação é o esperado, e também, permite ligar o produto produzido pela equipa de desenvolvimento aos seus clientes finais.

Na secção 2.6.3 será detalhado o BDD de uma forma mais aprofundada.

1.2 Problema

Atualmente, na organização, são utilizadas metodologias ágeis, nomeadamente o *Scrum* (Schwaber e Beedle 2002). O processo tradicional de desenvolvimento (TLD, Test-Last

Development, Teste após o desenvolvimento) é o processo predominante que as equipas adotam.

Após uma análise em conjunto com a empresa sobre aspetos qualitativos relativos ao *software*, confiança dos testes, efetividade do código e erros em produção, concluiu-se que os resultados podem ser melhorados. Para isso, é necessária a introdução de novas metodologias para o processo de desenvolvimento de *software*, processo esse que é a base para a construção dos produtos finais.

Os problemas que existem atualmente são as falhas (*bugs*) em *software* que está em produção, dificuldade e esforço aplicado sobre o desenvolvimento *software* (*legacy*), defeituosa interpretação dos requisitos, pouca eficácia e eficiência da bateria de testes e pouca consistência no desenho de soluções.

Estes problemas tem uma maior evidência em *software* com um nível de maturidade médio e elevado, que apesar de apresentarem uma cobertura de testes entre 60% a 80%, é um indicador que se revela débil quando nos deparamos com um número de erros em produção acima do esperado. Existem normas arquiteturais que fazem com que, numa fase inicial, novos projetos sejam fáceis de manter e desenvolver, no entanto o que se pretende é aplicar uma alternativa ao processo de desenvolvimento atual, para que exista um acompanhamento contínuo no ato de desenho, desenvolvimento e validação.

O processo tradicional de desenvolvimento de *software* que atualmente é praticado, para além de não conseguir dar resposta às novas necessidades, revela-se pouco eficiente.

1.3 Objetivos

Este estudo de caso tem como principal objetivo responder à hipótese central desta dissertação, *Será que a aplicação do desenvolvimento orientado aos testes, em contexto empresarial, permitirá melhorar a qualidade dos produtos e do processo de desenvolvimento de software?*, que surge após serem encontradas deficiências no processo de desenvolvimento atual.

O caminho delineado para ser possível avaliar esta hipótese passa pela introdução de um novo processo de desenvolvimento de *software*, com base no TDD em colaboração com o BDD, numa equipa de trabalho que não experimentou esta abordagem, com o propósito de analisar o seu impacto na qualidade do *software*, redução de erros e na produtividade da equipa durante o fluxo de desenvolvimento, de forma a responder aos problemas descritos na secção 1.2.

Este estudo de caso pretende analisar o progresso da aplicação da nova metodologia adaptada à realidade da organização, e com base no resultado final, o objetivo será adotar o novo processo de desenvolvimento, caso se revele uma mais valia para a equipa e para a organização.

Assim sendo, os principais sub-objetivos serão:

- Desenhar um novo processo de desenvolvimento, introduzindo o TDD e BDD, no contexto da organização.
- Implementar o processo TDD e BDD desenhado, através da sua aplicação por uma equipa seleccionada, em caso de estudo.

- Avaliar em várias dimensões o impacto da introdução do TDD e BDD desenhado, através da análise ao número de defeitos do código, satisfação da equipa relativamente ao novo processo, qualidade e cobertura do código, qualidade e eficácia da bateria de testes e por último, análise da produtividade da equipa de trabalho.

A hipótese central será detalhada e avaliada no capítulo 6, sobre diversas dimensões funcionais e não funcionais. Para sustentar a avaliação dos objetivos e dimensões referidos, será feita uma comparação com o processo tradicional de desenvolvimento de forma a analisar os resultados de forma coerente, e com base nas métricas utilizadas, retirar conclusões sobre os resultados deste estudo de caso. Esta comparação de resultados terá como base projetos anteriores a este estudo de caso, realizados pela mesma equipa de trabalho onde não foi aplicado o processo de desenvolvimento proposto nesta dissertação, mas sim o processo tradicional de desenvolvimento.

1.4 Análise de valor

Uma nova abordagem para o processo de desenvolvimento com base no TDD e BDD prevê uma melhor compreensão dos requisitos, um *software* mais fiável, com um bom desenho e com menos falhas. O desenvolvimento será mais fluido, facilitando o trabalho de todos os intervenientes que compõe o processo de desenvolvimento de *software*, como *Developers*, *Team-lead*, *Manager*, *Product Owner* e *Quality Assurance*. O aumento da confiança e da efetividade do *software* será um consequente muito importante também a ter em consideração. Parente este cenário, valor será criado para as partes envolvidas no processo, nomeadamente a empresa e a equipa de desenvolvimento, e como consequente, clientes finais que irão usufruir dos produtos com mais qualidade.

A análise de valor do estudo em questão será detalhada na secção 3.2.

1.5 Abordagem

A abordagem adotada foi desenhada de forma a ter em consideração as partes interessadas no projeto (*stakeholders*), sendo elas:

- Equipa de desenvolvimento: pretende melhorar o processo de desenvolvimento de *software*, aumentar a qualidade do *software* e entregar mais valor para os seus *stakeholders*.
- Empresa: com base no resultado deste estudo de caso, fundamentado com um caso real dentro do ecossistema da empresa, pode utilizar o processo proposto nesta dissertação em outras equipas de trabalho.

Como forma de dar respostas às partes interessadas, os seguintes tópicos tem como propósito delinear uma abordagem passo a passo, com o intuito de abordar o problema e responder as suas necessidades.

- Identificação do problema que este estudo de caso de propõe a resolver.
- Estudar e expor a metodologia TDD e BDD, o seu propósito e a sua aplicabilidade.

- Analisar estudos já realizados sobre a introdução da metodologia TDD e BDD, em equipas de trabalho sobre um contexto empresarial/industrial.
- Analisar e desenhar um novo processo de desenvolvimento de *software* com base no TDD e BDD, adaptado à realidade da empresa.
- Implementar o processo desenhado, com o objetivo de solucionar os problemas identificados.
- Avaliar, analisar e expor os resultados obtidos de forma a fundamentar as vantagens, desvantagens e o balanço final do processo desenhado.

1.6 Estrutura do documento

O documento está estruturado em sete capítulos distintos.

O primeiro capítulo é composto pela introdução, onde é apresentado ao leitor o contexto, o respetivo problema que este caso de estudo pretende resolver e os objetivos a seguir. A análise de valor expõe qual o valor que terá a solução proposta e por último, a abordagem a seguir para chegar a esse fim.

O segundo capítulo é referente ao estado da arte, onde é feito um enquadramento teórico com o propósito de expor os conceitos mais importantes para a compreensão deste estudo de caso.

O terceiro capítulo contempla a contextualização atual da equipa que irá abraçar este estudo de caso e a análise de valor.

O quarto capítulo tem como objetivo expor a análise e *design*. Analisa o estado atual dos processos e tecnologias, mostra o desenho de soluções alternativas e por último, documenta a solução para este estudo de caso.

O quinto capítulo é composto pela implementação, que evidência o contexto do projeto, implementação da solução desenhada e detalhes da instanciação da solução sobre a equipa.

O sexto capítulo é referente à avaliação deste estudo de caso. Através da definição de grandezas e hipóteses, com auxilio de métodos de avaliação, este capítulo irá avaliar o estudo de caso e expor os seus resultados de acordo com os objetivos traçados.

Por último, a conclusão apresenta os resultados obtidos fruto do trabalho desta dissertação e o trabalho futuro que se pretende dar continuidade.

Capítulo 2

Estado da Arte

Neste capítulo serão abordados os conceitos teóricos mais relevantes, apresentando métodos e abordagens conhecidas que irão servir de base na resolução do problema em questão.

O estado da arte está dividido em secções, sendo elas testes de *software*, onde irão ser introduzidos os principais tipos de teste, técnicas e ferramentas de teste, contextualizando as ferramentas existentes para o apoio aos testes de *software*, integração contínua, ferramentas utilizadas para automatizar o processo de entrega de *software*, indicadores e métricas de qualidade de testes, metodologias de desenvolvimento de *software*, processos de desenvolvimento de *software* e por fim, revisão das técnicas TDD e BDD a nível industrial.

2.1 Testes de software

O teste de *software* é um processo vital na avaliação da qualidade do *software* (Luo 2001). É uma prática que visa avaliar a capacidade de um programa de forma a determinar que este atende aos resultados exigidos. Esta prática é considerada crucial para a qualidade do *software* (W. C. Hetzel e B. Hetzel 1988).

O teste é mais do que apenas depuração, sendo que o seu objetivo pode ser garantia de qualidade, verificação e validação ou estimativa de confiança (W. C. Hetzel e B. Hetzel 1988). Os testes normalmente consomem entre 40% a 50% do esforço do desenvolvimento (Luo 2001), sendo um *trade-off* entre orçamento, tempo e qualidade (W. C. Hetzel e B. Hetzel 1988).

Segundo (Luo 2001), uma técnica de teste específica a estratégia usada no teste para selecionar casos de teste de entrada e analisar os resultados do teste. Diferentes técnicas revelam diferentes aspetos de qualidade de um sistema de *software*. Existem duas categorias principais de técnicas de teste, funcionais e estruturais.

- Teste funcional: o sistema é visto como uma caixa negra (*black box*). A seleção de casos de teste para testes funcionais é baseada ou derivada do requisito ou especificação de projeto. O teste funcional enfatiza o comportamento externo da entidade de *software* (Pan 1999) (Luo 2001).
- Teste estrutural: o sistema é visto como uma caixa branca (*white box*) ou caixa de vidro, pois a estrutura e o fluxo do *software* é visível para o responsável pela execução do mesmo (*tester*) (Pan 1999). A seleção dos casos de teste é baseada na implementação, ou seja, o objetivo é executar pontos específicos nas entidades de

software, como instruções específicas, ramificações de programa ou caminhos (Luo 2001).

O teste está envolvido em todas as etapas do ciclo de vida do *software*, mas o teste feito em cada nível de desenvolvimento de *software* é diferente por natureza e possui objetivos diferentes (Luo 2001). Nas secções seguintes irão ser expostos os principais tipos de testes de cada camada, que segundo (Luo 2001) são: O teste de unidade (*Unit test*) que é feito no nível mais baixo, o teste de integração (*Integration test*) é realizado quando duas ou mais unidades testadas são combinadas sobre uma estrutura, o teste de sistema (*System test* ou *End-to-End*) onde tende a afirmar a qualidade de ponta a ponta de todo o sistema e por fim o teste de aceitação (*User acceptance test*) onde é feito quando o sistema completo é entregue aos clientes ou utilizadores.

2.1.1 Testes de unidade

A escrita de testes é uma maneira eficaz e prática de melhorar a qualidade do *software*. Contém muitos benefícios quando comparado a métodos mais rigorosos, como raciocínio e provas formais, como simplicidade, funcionalidade, boa relação custo-benefício, *feedback* imediato, compreensibilidade e assim em diante (Cheon e Leavens 2002). A sua prática foi crescendo ao longo do tempo, tendo-se revelado bastante eficaz naquilo a que se compromete.

"O principal objetivo dos testes de unidade é encontrar erros dentro de uma determinada unidade. Se nenhum erro for encontrado, então serve para ganhar confiança na sua correção"(Wappler e Wegener 2006). Este é o principal objetivo dos testes de unidade, validação e garantia. Após o resultado do teste ser verdade, independentemente das vezes que executa, garantimos que a unidade de código em questão é válida. Com isso, conseguimos ter garantia de que se for necessário ocorrer o *refactor*, sabemos que existe algo que valida que o comportamento inicial foi mantido, sem comprometer a conduta da unidade. O código será melhorado, mas o seu comportamento manter-se-á.

Segundo (Wappler e Wegener 2006), no contexto da programação orientada a objetos, as classes que constituem uma aplicação são consideradas as menores unidades que podem ser testadas de forma isolada. Por norma, testar uma classe também envolve outras classes sendo o conjunto transitivo de classes que são relevantes para testar uma classe específica chamado de *cluster* de teste para essa classe.

O código produzido na escrita dos testes deve também ser redigido com qualidade, fácil de ler e simples. Segundo (Cheon e Leavens 2002), o objetivo é tornar a escrita do código do teste mais fácil de ler e compreender. Uma forma de fazer isso é usar uma estrutura que automatiza alguns dos detalhes da execução dos testes, como *frameworks*. As mudanças realizadas no código tem de ser acompanhadas também com mudanças no código dos testes de unidade, caso o comportamento da unidade de código a ser testada altere. Nesse caso, existe a necessidade de alterar o código, e a produção de código fácil de ler ajuda neste processo. "Testes de unidade difíceis de ler e frágeis podem causar estragos no código fonte"(Reese, A. D. George e Wenzel 2018).

Quando o código é fortemente acoplado, pode ser difícil de realizar o teste de unidade. Visto pela ótica da não existência de testes de unidade, o acoplamento pode ser menos aparente, no entanto, no processo de escrita dos testes de unidade, o acoplamento tende a desaparecer, caso contrário ficará muito difícil de testar (Reese, A. D. George e Wenzel

2018). Este é um forte indicador de que o código deverá ser redesenhado. Os testes de unidade relevam bem mais vantagens do que apenas simples validações de comportamento. Na realidade, quando a escrita do teste de unidade envolve muitas instruções e é difícil de realizar, poderemos estar presentes a um caso de necessidade de desacopolamento e *refactor*.

Segundo (Reese, A. D. George e Wenzel 2018), as características de um bom teste de unidade são:

- Rapidez, um teste de unidade deve demorar na ordem dos milissegundos.
- Isolado e independente, sem necessitar de nenhuma dependência ou fator externo.
- Repetível, resultando sempre no mesmo resultado, independentemente do número de vezes que executa.
- Auto-verificável e oportuno, o teste deve ser capaz de detetar se passou ou falhou, de forma automática.

O teste de unidade, tal como a escrita de qualquer código deve seguir normas para que possa ser também um código organizado. "Organizar, Agir e Assertar é um padrão comum quando se testa determinada unidade. Como o nome indica, consiste em três ações principais: Organizar os objetos de forma a criar e configurar conforme necessário para o caso de teste, atuar sobre o objeto e confirmar que o resultado é o esperado"(Reese, A. D. George e Wenzel 2018).

"Arrange, Act, Assert", este padrão permite que diferentes momentos sejam separados, ficando bem definido quais as etapas que o teste contém e a responsabilidade de cada uma. Na figura 2.1 está presente um exemplo que segundo (Reese, A. D. George e Wenzel 2018) segue o padrão dos três A's.

```
1  [Fact]
2  public void Add_TwoNumbers_ReturnsSumOfNumbers()
3  {
4      //Arrange
5      var stringCalculator = CreateDefaultStringCalculator();
6
7      //Act
8      var actual = stringCalculator.Add("0,1");
9
10     //Assert
11     Assert.Equal(1, actual);
12 }
```

Listing 2.1: AAA padrão (Reese, A. D. George e Wenzel 2018)

"A legibilidade é um dos aspetos mais importantes ao escrever um teste. Separar cada uma dessas ações dentro do teste, destaca claramente as dependências necessárias para chamar o código, como o código está a ser chamado e o que se está a tentar afirmar, [...] o objetivo principal é tornar o teste o mais legível possível"(Reese, A. D. George e Wenzel 2018). Estes autores defendem também a utilização de uma normalização para a nomenclatura dos testes de unidade, que passa por agregar o nome com o nome do método que está a ser testado, o cenário a ser testado e por fim, o comportamento esperado do teste.

2.1.2 Testes de integração

O objetivo prático de cada fase de teste é detetar erros que provavelmente não serão detetados pelas fases de testes anteriores. Neste caso, o teste de integração deve ter como objetivo detetar erros que podem não ser encontrados durante o teste de unidade, ou seja, o teste é aplicado quando dois ou mais módulos individuais são combinados para formar um programa ou um componente de *software*. O teste é feito ao nível do módulo, ou invés do nível de instrução como nos testes de unidade (Leung e White 1990). Assim sendo, o objetivo é validar a integração de módulos, a comunicação entre interfaces e avaliar o resultado final do processamento entre os módulos (Luo 2001)

Segundo (Leung e White 1990), após a conclusão do teste de unidade, os módulos devem ser integrados. Durante os testes de integração, os três principais pontos a validar são: *missing function*, *extra function* e *interface errors*. Estes erros acontecem quando o *software* foi mal desenhado ou implementado. Um exemplo dado por (Leung e White 1990) na ajuda à deteção destes problemas passa por integrar o módulo *A* com o módulo *B*:

1. Garantir que *A* não utilizará funcionalidades que não possam ser fornecidas por *B* *missing function*.
2. Assegurar que *A* não utilizará outras funcionalidades de *B* se *B* fornecer mais funções do que aquelas requeridas por *A* *extra function*.
3. Garantir que os padrões de interface entre *A* e *B* sejam preservados *interface errors*.

(Leung e White 1990) defende ainda a integração de apenas dois módulos. Ao integrar mais de dois módulos, podemos integrá-los de forma incremental, adicionando um novo módulo ao conjunto de módulos integrados, de forma sequencial.

2.1.3 Testes de sistema

O teste de sistema (*System test* ou *End-to-End*) tem como objetivo afirmar a qualidade do sistema de "ponta a ponta". É focado na funcionalidade do sistema integrado do ponto de vista do utilizador final. Esta metodologia pressupõe que tanto os testes de unidade como os teste de integração sejam executados e aprovados (Tsai et al. 2001).

O teste do sistema é geralmente baseado na especificação funcional dos requisitos do sistema. Requisitos não funcionais como confiança, segurança e facilidade de manutenção, também são verificados (Luo 2001). O principal objetivo é verificar se o *software* integrado, incluindo subsistemas internos e sistemas de suporte externos, apoia coletivamente as principais funções de negócio da organização (Tsai et al. 2001).

2.1.4 Testes de aceitação

"O teste de aceitação é o fator determinante na satisfação do contrato entre o fornecedor do *software* e o cliente"(Hsia et al. 1994). Esta técnica é aplicada quando o sistema completo é entregue aos clientes, sendo o principal objetivo do teste de aceitação garantir que o sistema funciona tal como foi requisitado pelo cliente, verificando as interações homem-máquina, recursos necessários e restrições de sistema (Luo 2001). Segundo (De Lucia e Qusef 2010), o teste de aceitação é um teste formal que deve ser conduzido pelo cliente para

garantir que o sistema satisfaz os critérios de aceitação contratual. Os testes de aceitação não são diferentes dos testes do sistema automatizados, a diferença é que são realizados pelo cliente.

Entregar *software* ao cliente é um princípio ágil fundamental. Os clientes criam critérios de aceitação para os requisitos e testam os requisitos em relação a esses critérios, podendo os clientes dar *feedback* para melhorar o desenvolvimento de incrementos futuros no sistema (De Lucia e Qusef 2010).

2.2 Técnicas e ferramentas de teste

Esta secção irá apresentar as técnicas e ferramentas de teste, utilizadas ao longo deste estudo de caso, que servem de auxilio ao processo de escrita, manutenção e compreensão dos testes.

2.2.1 xUnit e NUnit

NUnit é uma ferramenta para realizar testes de unidade (Stolberg 2009) para tecnologias *Microsoft .Net*. Esta é uma ferramenta *open-source* e tem o mesmo propósito que o *JUnit* para a linguagem *Java*. A ferramenta *NUnit* é baseada na família de arquiteturas *xUnit*, especializada em fornecer uma base para testes de unidade ou testes de API (*Application Program Interface*), sendo o *NUnit* também uma ferramenta de teste API (Butt, Bhatti e Sarwar 2017). "xUnit é uma ferramenta de teste *open-source*, focada na comunidade para *.NET Framework*. Escrito pelo autor original do *NUnit*, *xUnit* é a última tecnologia para testes de unidade *C#, F#, VB.NET* e outras linguagens"(Foundation n.d.).

O teste de unidade é escrito na ferramenta de teste *NUnit* sobre um projeto de teste que faz referência à *Dynamic Linked Library* (DLL) em que uma API é baseada. Além disso, a estrutura da ferramenta *NUnit* deve ser configurada no projeto de teste. Esta ferramenta permite anotações sobre os teste, decorando-os de acordo com os atributos disponíveis, facilitando a leitura e compreensão do teste (Butt, Bhatti e Sarwar 2017). No excerto 2.2, podemos observar um exemplo de um teste à classe *PrimeServiceIsPrimeShould* usando o *NUnit*.

```
1  using NUnit.Framework;
2  using Prime.Services;
3
4  namespace Prime.UnitTests.Services
5  {
6      [TestFixture]
7      public class PrimeServiceIsPrimeShould
8      {
9          private readonly PrimeService primeService;
10
11         public PrimeServiceIsPrimeShould()
12         {
13             primeService = new PrimeService();
14         }
15
16         [Test]
17         public void ReturnFalseGivenValueOf1()
```

```
18          {
19              var result = primeService.IsPrime(1);
20
21              Assert.IsFalse(result, "1 should not be prime");
22          }
23      }
24  }
```

Listing 2.2: NUni - exemplo (Prouse et al. 2018)

Segundo (Butt, Bhatti e Sarwar 2017), a execução do conjunto de testes no *NUnit* compila resultados que podem ser exportados para fins de personalização, por exemplo, o relatório XML. O resultado dos testes aparecem a verde em caso de sucesso, a falha é destacada a vermelho e os testes ignorados são sinalizados a amarelo.

2.2.2 Mock Objects

O teste de unidade é uma prática fundamental para garantir a qualidade do código. O problema é que a maioria do código existente num programa não tem instruções triviais, e é difícil de testar de forma isolada. Testar um unidade é uma tarefa árdua sem envolver as suas dependências (Mackinnon, Freeman e Craig 2000). "O que torna o teste de unidade tão difícil é a forma com que a unidade interage com o mundo real. Se tudo o que precisássemos de fazer fosse codificar testes para métodos que ordenassem matrizes ou gerassem séries de Fibonacci, seria tudo mais fácil"(D. Thomas e Hunt 2002). Segundo (D. Thomas e Hunt 2002), os desafios que encontramos no processo de teste são testes ao código que usa base de dados, dispositivos de comunicação, interfaces e aplicações externas. Todas estas dependências impedem que o teste de unidade seja um código preciso e concreto. Se não existir metodologias que auxiliem o processo de teste, de forma inevitável as dependências acabam por ser inicializadas e utilizadas para que o teste funcione, introduzindo acoplamento e perda de tempo no processo de execução. Para além disso, na realidade, se por ventura existir alguma alteração sobre uma interface ou uma tabela na base de dados, consequentemente o teste de unidade poderá falhar, caso a esta alteração provoque um comportamento inesperado, o que fará com que o teste falhe porque a dependência foi modificada, não porque existe um erro na unidade em questão.

Os *Mock Objects* vem dar resposta a esta necessidade, atuando de forma a simular os comportamentos das dependências, ou seja, é possível configurar um comportamento esperado de forma isolada, fazendo com que o teste de unidade não precise de uma instância real da sua dependência, realçando as afirmações sobre o comportamento do código (D. Thomas e Hunt 2002). Testes com *Mock Objects* são testes mais fortes e com uma melhor estrutura, fornecendo um vocabulário comum, tornando o processo de teste de unidade mais fácil, rápido e escalável (Mackinnon, Freeman e Craig 2000).

O excerto de código 2.3 apresenta uma função que converte a temperatura de *Fahrenheit* para *Celsius*. Esta função tem duas dependências externas, *HttpServletRequest* e *HttpServletResponse*. Sem a utilização de *mock objects*, teríamos inevitavelmente de utilizar uma instância real.

```
1  public void doGet(HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException
2  {
3      String strf = req.getParameter("Fahrenheit");
```

```
4
5       res.setContentType("text/html");
6       PrintWriter out = res.getWriter();
7
8       try{
9           int tempf = Integer.parseInt(strf);
10          double tempc = (tempf - 32) * 5.0 / 9.0;
11          out.println("Fahrenheit: " + tempf + ", Celsius: " + tempc);
12      }
13      catch (NumberFormatException e) {
14          out.println("Invalid temperature: " + strf);
15      }
16  }
```

Listing 2.3: Conversor de Fahrenheit para Celsius (D. Thomas e Hunt 2002)

No excerto 2.4, podemos observar o teste de unidade usando *Mock Objects*. Com a utilização desta prática, é possível simular um determinado comportamento sobre as dependências. Neste caso em concreto, o objetivo seria simular uma falha na dependência e validar o retorno da função *doGet*.

```
1  public void testBadParameter() throws Exception {
2      TemperatureServlet s = new TemperatureServlet();
3      MockHttpServletRequest request = new MockHttpServletRequest();
4      MockHttpServletResponse response = new MockHttpServletResponse();
5
6      request.setupAddParameter("Fahrenheit", "boo!");
7      response.setExpectedContentType("text/html");
8      s.doGet(request, response);
9      response.verify();
10     assertEquals("Invalid temperature: boo!\r\n", response.
      getOutputStreamContents());
11 }
```

Listing 2.4: Teste de unidade com Mock Objects (D. Thomas e Hunt 2002)

(Mackinnon, Freeman e Craig 2000) refere ainda um conjunto de padrões para a utilização desta prática, como criação de instâncias de objetos *Mock*, definição do estado nos objetos *Mock*, definição das expectativas nos objetos *Mock*, invocação do código de domínio com objetos *Mock* como parâmetro, e por fim, verificação da consistência dos objetos *Mock*.

Segundo (Marri et al. 2009) e (Mackinnon, Freeman e Craig 2000), esta prática incentiva a testes mais estruturados e reduz o custo, proporcionando um formato comum nos testes de unidade que é fácil de aprender e compreender, simplificando a depuração e fornecendo testes que detetam o ponto exato da falha no momento em que um problema ocorre. *Mock Objects* por vezes é a única forma de testar código que depende de um estado difícil ou impossível de reproduzir, ou seja, testes com *Mock Objects* melhora o código do domínio, preservando o encapsulamento, reduzindo as dependências globais e esclarecendo as interações entre as classes.

2.2.3 Live Unit Testing

Live Unit Testing é um *Plug-in*, desenvolvido para o IDE (*Integrated Development Environment*) *Visual Studio*, que executa automaticamente todos os testes em segundo plano,

durante o desenvolvimento, e apresenta os resultados e a cobertura do código em tempo real no IDE, fornecendo *feedback* sobre as alterações postas em prática, indicando se tiverem impacto positivo ou negativo nos testes, ou se não tiveram qualquer impacto, indicador de que provavelmente será necessário cobrir o novo código. É uma prática que incentiva a criação de novos testes para tarefas de correção de *bugs*, ou simplesmente adição de novas funcionalidades (Rpetrusha 2017).

Exemplo de execução do Live Unit Testing no Visual Studio O *Live Unit Testing* ajuda a ver rapidamente se o código que está a ser escrito está coberto e se os testes que o cobrem estão a executar com sucesso. Os resultados dos testes e a visualização de cobertura são visíveis em cada linha de código, representado por três estados diferentes (Blog 2017):

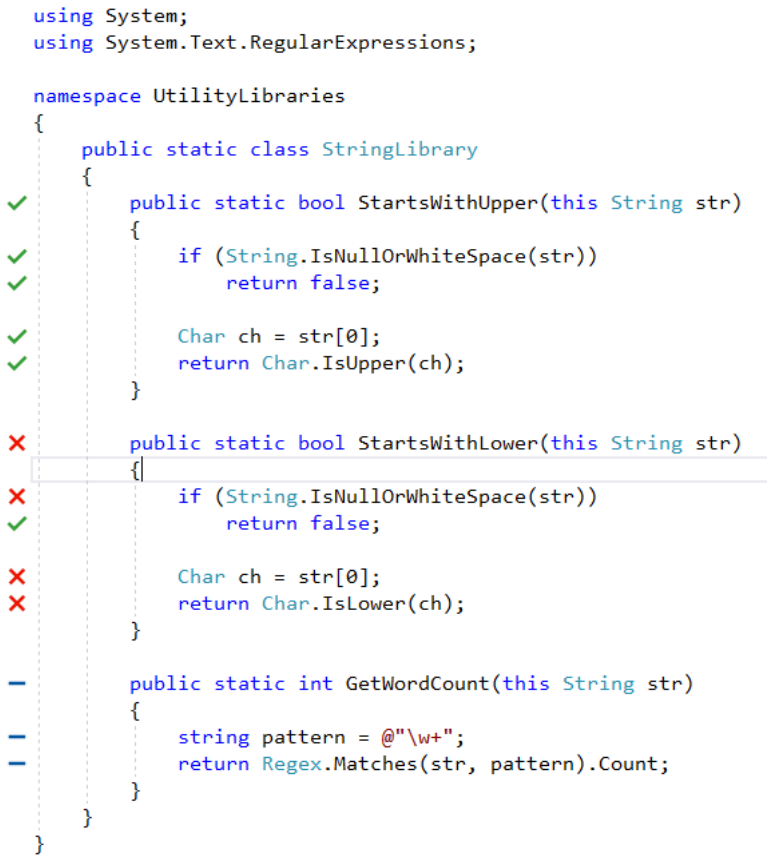

Figura 2.1: Live Unit Tests - Estados (Rpetrusha 2019)

- Uma linha de código executável que é coberta por pelo menos um teste com falha é decorada com um "x"a vermelho.
- Uma linha de código executável que é coberta apenas por testes e com aprovação é decorada com um "v"a verde.
- Uma linha de código executável que não é coberta por nenhum teste é decorada com um traço "-"a azul.

2.2.4 Selenium

O *Selenium* é uma ferramenta de automação de testes de Web que usa *scripts* para executar testes sobre um navegador web. Funciona à base de *JavaScript* e *iframes* para incorporar o mecanismo de automação de testes ao navegador, permitindo a utilização de multi-plataformas (Gojare, Joshi e Gaigaware 2015). O *Selenium* fornece um conjunto de instruções como: abrir um URL, clicar sobre um elemento ou inserir valores sobre uma caixa de entrada de texto, tal como podemos observar no excerto 2.5. Também fornece um conjunto de comandos que permitem a verificação a especificação de valores ou comportamentos esperados (Holmes e Kellogg 2006).

```
@Test
public void testConfirmDismiss()
{
    //Clicking button will show a Confirmation Alert with OK and
    //Cancel Button
    WebElement button = driver.findElement(By.id("confirm"));
    button.click();
    try {
        //Get the Alert
        Alert alert = driver.switchTo().alert();
        //Click Cancel button, by calling dismiss() method of
        //Alert Class
        alert.dismiss();
        //Verify Page displays correct message on Dismiss
        WebElement message = driver.findElement(By.id("demo"));
        assertEquals("You Dismissed Alert!", message.getText());
    } catch (NoAlertPresentException e) {
        e.printStackTrace();
    }
}
```

Listing 2.5: *Selenium* - Exemplo de teste (Gundecha 2012)

Segundo (Umesh e Saraswat 2015), testes de *Selenium* são fáceis de escrever, permitindo a identificação de elementos usando os objetos DOM do navegador. Os testes devem ser independentes e autónomos. A utilização de práticas como *background*, *SetUp* e *TearDown* ajuda na redução da duplicação de código, colocando a aplicação no estado desejado para testar um conjunto de funcionalidades.

No estudo realizado por (Umesh e Saraswat 2015), concluiu-se que o seguinte conjunto de passos ajuda na fluidez dos testes de *Selenium*, facilitando a captação e compreensão dos requisitos.

- Escreva os testes antes do desenvolvimento.
- Após o desenvolvimento, coloque-os a verde.
- Por fim *Refactor*, reduz a duplicação do código e aumentar a capacidade de manutenção do conjunto de testes.

2.2.5 Cucumber

Cucumber é uma ferramenta automatizada de teste de aceitação. Utiliza o BDD como base, permitindo através da estrutura *Given, When, Then* a construção de cenários de

teste, enquanto a componente executável dos casos de teste é gravada em código *Ruby*. Funciona com *Java, Ruby, .NET, Flex* ou aplicações Web (Wynne, Hellesoy e Tooke 2017).

```
Feature: Cash Withdrawal
    Scenario: Successfull withdrawal from an account in credit
    Given: I have deposited $180 in my account
    When I request $20
    Then $20 should be dispensed
```

Listing 2.6: Cucumber - Exemplo de um teste de aceitação (Wynne, Hellesoy e Tooke 2017)

Na figura 2.6 podemos observar um exemplo de um teste em *Cucumber*. A sintaxe do BDD está presente, facilitando a leitura e compreensão. O resultado do testes são apresentados em HTML, simples de interpretar e portáveis (Wynne, Hellesoy e Tooke 2017).

Cucumber com Selenium O *Cucumber* pode ser utilizado com *Selenium*, ou seja, o *Cucumber* permite saber através de texto simples o que um teste deve verificar e o *Selenium* faz chamadas diretas para o navegador web e executa as etapas (Yurtoğlu 2018). Com a utilização destas duas ferramentas em conjunto, as instruções técnicas executadas pelo *Selenium* ficam "escondidas", apenas ficando visíveis os passos escritos com *Cucumber*, que irão ser mapeadas todas as etapas para uma função ou um método (Wynne, Hellesoy e Tooke 2017) que o *Selenium* irá executar. (Yurtoğlu 2018) refere que a lacuna entre a tecnologia e o negócio é preenchida, pois o *Cucumber* através a sua linguagem simples permite que qualquer pessoa com um *background* não técnico compreenda os passos dos testes.

2.2.6 SpecFlow

O *SpecFlow* é uma ferramenta que permite que equipas de desenvolvimento que trabalham sobre *.NET* definam, executem e mantenham testes de aceitação automatizados. O *SpecFlow* é baseado no *Gherkin* e faz parte do sistema ecológico do *Cucumber*. Utiliza uma linguagem que permite a que as entidades de negócio e as entidades mais técnicas se compreendam entre si (Specflow 2019).

Os testes de aceitação no *SpecFlow* seguem o paradigma do BDD. Os testes de aceitação podem então ser testados automaticamente. A sua especificação serve como documentação viva do sistema (Specflow 2019), um aspeto importante que visa também colmatar a falta de documentação existente de *software*.

O *SpecFlow* integra-se no *Visual Studio*, permitindo que o conceito BDD possa ser aplicado dentro do ecossistema *.NET*.

2.2.7 MockServer

O *MockServer* é uma ferramenta que pode ser usada para simular qualquer servidor ou serviço que use o protocolo *HTTP/HTTPS* (Bloom 2018), ou seja, o *MockServer* é considerado um *proxy* que substitui o comportamento de um sistema real com base em configurações. Segundo (Bloom 2018), quando o *MockServer* recebe um pedido, ele corresponde ao pedido em relação às expectativas ativas que foram configuradas. Uma expectativa define a ação

que é executada. Um exemplo pode ser, retornar uma determinada resposta quando um determinado pedido é realizado.

(Bloom 2018) afirma que o *MockServer* suporta as seguintes ações:

- Retornar uma resposta "simulada"quando uma solicitação corresponde a uma expectativa.
- Encaminhar uma solicitação quando a solicitação corresponder a uma expectativa.
- Executar um retorno de uma chamada quando uma solicitação corresponder a uma expectativa, permitindo que a resposta seja criada dinamicamente
- Retornar uma resposta inválida ou fechar a conexão quando uma solicitação corresponder a uma expectativa.
- Verificar se os pedidos foram efetivamente enviados.

Segundo (Bloom 2018), o *MockServer* é bastante útil em cenários onde é necessário recriar todas as dependências externas para que os testes sejam independentes, fáceis e rápidos, isolando a bateria de testes da aplicação alvo, permitindo a que quando os testes falhem, se descubra uma falha interna do sistema e não uma falha ou erro de um sistema externo. Com a facilidade de criação de expectativas e verificação de solicitações a determinados recursos fazem desta ferramenta uma forma de desacoplar a aplicação a ser testada das suas dependências, permitindo validar e similar todas as operações reais efetuadas pela aplicação a ser testada.

2.2.8 WireMock

O *WireMock* é uma ferramenta que tem como propósito simular *APIs* baseadas no protocolo *HTTP/HTTPS* para que no contexto de testes, estes sejam rápidos, robustos e abrangentes. *WireMock* pode ser considerado como uma ferramenta de virtualização de serviços, ou seja, permite que independentemente das *APIs* externas estarem a funcionar ou não, continuar o trabalho sem bloqueios (Minimal e Akehurst 2019).

Segundo (Minimal e Akehurst 2019), esta ferramenta representa um servidor que pode ser configurado para responder a pedidos específicos e capturar pedidos, para que possam ser verificados posteriormente. Pode ser usado como uma biblioteca por qualquer aplicação *JVM* ou ser executado como um processo independente no mesmo *host* que o sistema em teste ou num servidor remoto. Todos os recursos do *WireMock* são acessíveis através de sua interface *REST* e pela sua API *Java*.

WireMock.Net Segundo (WireMock-Net 2019), O *WireMock.Net* é uma biblioteca flexível para fragmentar e simular respostas *HTTP/HTTPS*. É baseado na ferramenta *WireMock* 2.2.8, mas estendido com mais funcionalidades para a plataforma *.NET*.

Pode ser usado em diversas situações, como em testes de unidade ou testes de integração.

2.2.9 Mongo2Go

O Mongo2Go é uma abstração em torno dos binários mais recentes do *MongoDB*. Funciona com Windows, Linux e macOS. Este pacote *Nuget* contém os executáveis do *mongod*, *mongoimport* e *mongoexport* (Mongo2Go 2019a).

Este recurso permite que uma aplicação desenvolvida sobre a *framework .NET* e linguagem C# possa instalar e usufruir de um simulador do *MongoDB* sem que seja necessária a instalação ou virtualização do *MongoDB*.

Segundo (Mongo2Go 2019a), o *Mongo2Go* possui dois casos de uso:

- Fornecer várias bases de dados *MongoDB* temporárias e isolados para testes de unidade ou de integração.
- Fornecendo uma base de dados *MongoDB* de configuração rápida para um ambiente de desenvolvimento local.

2.2.10 AspNetCore Mvc Testing

AspNetCore Mvc Testing tem como objetivo disponibilizar um conjunto de funcionalidades capazes de criar um *host* em memória de uma determinada aplicação *.Net Core*. O uso deste pacote otimiza a criação e execução de testes, permitindo criar diferentes instâncias aplicacionais de acordo com parâmetros (Microsoft 2019c).

2.3 Indicadores e métricas de qualidade de testes

Indicadores e métricas de qualidade podem ajudar a perceber quão fortes são os nossos testes, qual o grau de segurança que temos sobre o código, entre outros. Segundo (Zhang 2012), o principal objetivo de construir testes de *software* é melhorar a qualidade do *software*. A métrica ajuda a avaliar a qualidade dos testes do *software*, a fim de avaliar o quão bem o teste foi feito. É importante medir as partes críticas que realmente são úteis, selecionando medições específicas de acordo com as necessidades.

2.3.1 Code Coverage

Code Coverage é um indicador que mede a cobertura dos testes sobre o código de produção, apontando quais partes do código não estão a ser contempladas pelos testes. Como consequente, expõe as funcionalidades implementadas pelo código que não estão a ser testadas adequadamente ou na sua totalidade, sendo que não vão ser testadas na sua íntegra até que o utilizador final as use. Por norma, um *bug* não pode ser encontrado se não for executado e quanto mais cedo um *bug* for encontrado, mais rápido e mais simples pode ser corrigido (Burr e Young 1998). *Developers* e *testers* usam o *code coverage* para garantir que todas ou substancialmente todas as declarações de um programa tenham sido executadas pelo menos uma vez durante o processo de teste *tikir2002efficient*.

Segundo (Marick et al. 1999), o *code coverage* é um indicador poderoso, exemplificando com "se uma linha nunca foi executada, é uma aposta segura que não foi verificado se existe

um *bug* escondido". Marick alerta também para o facto de que o indicador é facilmente contornável, caso sejam escritos testes fracos com *asserts* fracos para cobrir as instruções do código, fazendo com que se atinja uma maior percentagem de cobertura. "Estar testado"não é a mesma coisa que "exercitar todas as condições de cobertura", ou seja, é necessário fazer com que todas as expressões lógicas avaliem tanto o verdadeiro como o falso. O *code coverage* não diz quais os testes que são necessários escrever, apenas pode dizer qual o código existente com cobertura.

O *code coverage* é portanto um indicador que nos mostra em percentagem as instruções cobertas pelos testes sobre o total de instruções do código. Segundo (Marick et al. 1999), projetar este indicador inicialmente para atingir 100% de cobertura não é uma boa prática, pois será uma meta que levará, por norma, há criação de testes fracos para atingir a métrica estabelecida ao invés de criar testes fortes que validem efetivamente todas as expressões lógicas.

Contudo, o *code coverage* é um indicador muito importante para medir tanto a cobertura dos testes sobre o código fonte como para alertar quais as instruções que não estão a ser validadas pelos testes. Segundo (Marick et al. 1999), este indicador tem de ser bem utilizado e interpretado, pois apenas serve para o propósito a que se compromete, não para expor quais os testes que são necessários de escrever.

Tipos de Code Coverage

Segundo (Santelices et al. 2009), (Elbaum, Gable e Rothermel 2001) e (Fallah, Devadas e Keutzer 2001) existem três grandes tipos de *code coverage*:

- *Statement coverage* (cobertura sobre instrução) mede as instruções que foram executadas de forma isolada.

  A seguinte função representa como é feito o calculo do *statement coverage*.

$$\%Of\ statement\ coverage = \frac{Number\ of\ executed\ statements}{Total\ number\ of\ statements}\ X\ 100$$

- *Branch coverage* (cobertura de ramo ou cobertura de decisão) mede quais as possíveis ramificações nas estruturas do código que são seguidas.
- *Function coverage* (cobertura sobre função), mede se um método ou função foi executado durante a execução dos testes.

2.3.2 Mutation Tests

O conceito de *mutation test* foi introduzido na década de 1970, como uma técnica baseada em mutações que mede a eficácia da deteção de falhas dos testes com base na introdução de falhas de forma consciente, com o propósito de reforçar a qualidade dos testes que cobrem as instruções do código fonte. Durante o processo de *mutation test*, as falhas são introduzidas num programa através da criação de várias versões desse mesmo programa, cada uma das quais contendo uma mutação. O termo mutação nasceu da prática de alteração do programa original, sendo que as versões defeituosas do programa são mutantes do original, e um mutante é dado como "morto"quando um caso de teste falha, satisfazendo a necessidade

de identificar um caso de teste útil (Offutt 1994). Um mutante "vivo", mostra um caso em que o conjunto de testes falha potencialmente no exercício de detetar erros e, portanto, precisa de ser melhorado de forma a conseguir capturar o mutante (DeMillo, Lipton e Sayward 1978). Os *mutation test* dão uma visão sobre a qualidade dos testes, refletindo a necessidade de melhorar o conjunto de testes ou, evidenciando a sua qualidade na cobertura do código fonte.

Segundo (Offutt 1994), existem duas categorias após cada caso de teste ter sido executado contra cada mutante vivo:

- O mutante é funcionalmente equivalente ao programa original. Um substantivo equivalente sempre produz a mesma saída que o programa original, portanto, nenhum caso de teste pode matá-lo.
- O mutante é eliminável mas o conjunto de casos de teste é insuficiente para eliminá-lo. Nesse caso, novos casos de teste precisam ser criados e o processo itera até que o conjunto de testes seja forte o suficiente para satisfazer o testador.

A pontuação de mutação para um conjunto de dados de teste é a percentagem de mutantes não equivalentes mortos por esses dados. Um conjunto de dados de teste de mutação é adequado se a sua pontuação de mutação for de 100% (Offutt 1994).

Apesar de todas as vantagens dos *mutation tests*, segundo (Jia e Harman 2011), esta técnica é considerada como uma técnica de teste dispendiosa em termos computacionais. Na sua publicação, foi feita uma análise às técnicas de *mutation tests* mais relevantes e mais utilizadas, sendo elas:

- *Mutant Sampling* é uma abordagem simples que escolhe aleatoriamente um pequeno subconjunto de mutantes de todo o conjunto.
- *Mutant Clustering* é uma técnica com base na escolha de um subconjunto de mutantes, ou seja, os mutantes são agrupados em *clusters* sendo que cada mutante do mesmo *cluster* tem a garantia de ser morto por um conjunto de casos de teste semelhante. Apenas um pequeno número de mutantes é selecionado de cada grupo para ser usado nos testes. Os mutantes restantes são descartados.
- *Selective Mutation* tem por base a redução do número de operadores de mutação aplicados, procurando encontrar um pequeno conjunto de operadores de mutação que geram um subconjunto de todos os possíveis mutantes sem perda significativa da eficácia do teste. Operadores de mutação geram diferentes números de mutantes e alguns operadores de mutação geram muito mais mutantes do que outros, muitos dos quais podem se tornar redundantes.
- *Higher Order Mutation* é a técnica de encontrar mutantes de ordem superior, raros, mas valiosos, que denotam falhas subtis. No tradicional Teste de Mutação, os mutantes podem ser classificados em mutantes de primeira ordem (FOMs) e mutantes de ordem superior (HOMs). Os FOMs são criados aplicando um operador de mutação apenas uma vez. HOMs são gerados pela aplicação de operadores de mutação mais de uma vez.

Estas técnicas assentam sobre o conjunto de premissas "fazer menos", "fazer mais rápido"e "fazer de forma inteligente", com o objetivo de reduzir o número de mutantes, identificar os pontos críticos e criar mutantes mais robustos e com propósito lógico.

2.3.3 Tempo de execução dos testes

O tempo de execução dos testes é um fator muito importante durante o desenvolvimento de *software*. Depois da correção um *bug* ou implementação de um novo recurso, os *developers* normalmente executam os testes para verificar se os componentes do *software* continuam a funcionar corretamente, de forma a garantir que o comportamento do *software* foi mantido. Espera-se que um conjunto de testes seja executado dentro de alguns segundos ou minutos. No entanto, grandes projetos de *software* tem conjuntos de testes que levam muito tempo a executar, aumentando significativamente o tempo de espera, tornando menos produtivo quem está à espera do resultado dos testes, aumentado a probabilidade de quem está a desenvolver não executar o conjunto de testes, aumentando a probabilidade de criar um *bug* (Kim, Chandra e Zeldovich 2013). Segundo (Beck 2003), o tempo de execução dos testes deve ser o mais curto possível, para que a equipa de desenvolvimento tenha *feedback* imediato sobre o código que está a produzir.

O tempo de execução é uma métrica de qualidade, quando mais curto for, mais benefícios proporciona. Associado ao tempo, estão as boas práticas de escrita de testes, sendo que quando o tempo de execução é demasiado longo, pode ser um indicador de que o teste poderá não estar a seguir a melhor abordagem.

2.4 Integração contínua

O termo "Integração Contínua"apareceu como uma prática da disciplina *Extreme Programming* (Fowler e Foemmel 2006). Um sistema de integração contínua é frequentemente considerado um dos principais elementos envolvidos no que diz respeito ao suporte a um ambiente ágil de desenvolvimento e teste de *software* (Stolberg 2009). A integração contínua é, portanto, uma prática de desenvolvimento de *software* em que os membros de uma equipa integram o seu trabalho com frequência, sendo que cada nova integração é verificada por uma compilação automatizada (incluindo testes), de forma a detetar erros de integração o mais rápido possível (Fowler e Foemmel 2006), dando *feedback* contínuo sobre o novo trabalho desenvolvido.

A CI/CD (Integração e entrega contínua) Pipeline ajuda a automatizar o processo de entrega contínua, permitindo a entrega rápida de *software* (Arachchi e Perera 2018), proporcionando uma implementação automatizada do processo de criação, implantação, teste e lançamento de *software*. Este processo pode ocorrer sempre que exista uma alteração no *software* (Focus 2016), eliminado o processo manual de entrega contínua (implantação). Os benefícios desta prática são a entrega rápida, a segurança, a qualidade e a entrega de forma automatizada (Focus 2016). Diferentes estágios podem compor a pipeline, como o processo de *build* de soluções, execução de testes de unidade, de integração, de contrato ou *smoke tests*, implantação em servidores, geração de relatórios etc. Em suma, CI/CD Pipeline é uma ferramenta configurável que permite automatizar processos de entrega e integração de acordo com as necessidades.

Os pontos seguintes irão apresentar ferramentas que suportam a prática de integração contínua.

2.4.1 Team City

O *TeamCity* é um servidor de integração contínua (CI). Permite executar instruções de forma paralela simultaneamente em diferentes plataformas e ambientes, otimizar o ciclo de integração de código e certificar-se que não existe código danificado (que não compila ou a existência de testes a falharem) no repositório. Através dos seus relatórios é possível obter *feedback* contínuo sobre as aplicações, cobertura de código, taxas de sucesso, métricas personalizadas, etc (Alexandrova 2018).

O *TeamCity* funciona à base de agentes, que são caracterizados por *software* responsável pela construção, compilação e execução da aplicação. Os agentes podem ser configurados para diferentes plataformas, sistemas operacionais e ambientes pré-configurados (Alexandrova 2018).

O fluxo de processamento do *TeamCity* pode ser visualizado através do exemplo dado por (Alexandrova 2018):

- O servidor *TeamCity* deteta uma alteração no código fonte e armazena essa informação.
- O componente de construção observa essa informação e coloca um novo elemento na fila.
- O servidor encontra um agente de compilação compatível e atribui a compilação a esse agente.
- O agente executa as etapas de construção. Enquanto as etapas de compilação estão a ser executadas, o agente de compilação informa o progresso da compilação para o servidor *TeamCity* enviando todas as mensagens de informação (log), relatórios de teste, resultados de cobertura de código, etc. Com isto, é possível em tempo real realizar o processo de monitorização.
- Depois de terminar a compilação, o agente de compilação envia os artefactos de compilação para o servidor.

2.4.2 Jenkins

"O Jenkins é um servidor de automação independente e *open-source* que pode ser usado para automatizar todo o tipo de tarefas relacionadas com a criação, teste e distribuição ou implementação de *software*. [...] é um produto altamente extensível, cuja funcionalidade pode ser estendida através da instalação de *plugins*"(*Jenkins* 2018). Pode ser instalado através do sistema nativo, *Docker* ou até mesmo executado de forma independente por qualquer máquina com o *Java Runtime Environment (JRE)* instalado (*Jenkins* 2018).

O *Jenkins* é portanto uma ferramenta de integração contínua, pode executar vários *scripts* como *Ant* e *Maven* e *scripts* de *shell* para ambientes *Windows* e *Unix/Linux*. A cada compilação (*builds*), podem ser gerados relatórios de teste para que possíveis problemas possam ser facilmente detetados e compreendidos. Os relatórios são gerados de acordo com a estrutura de testes, que embora por omissão exista o *JUnit*, outras estruturas de teste são suportadas através de *plugins*. As ferramentas da gestão de controlo de código suportadas são: *CVS, Subversion, Git, Mercurial, Perforce e Clearcase*. Os *builds* podem ser ativados por *commits*, sondagem ao repositório, manualmente, através de uma API, etc.

Para que toda a orquestração do *Jenkins* seja feita, o *Jenkins* disponibiliza uma página Web para que a manipulação e construção de componentes, *feedback* e configuração seja possível de forma fácil (Hembrink e Stenberg 2013).

2.4.3 Docker

O *Docker* é uma tecnologia de virtualização de *containers* (C. Anderson 2015), aberta para desenvolvimento, envio e execução de aplicações. O *Docker* permite separar as aplicações da infraestrutura para que seja possível entregar *software* rapidamente, sendo possível reduzir significativamente o atraso entre a gravação do código e sua execução em produção (*Docker overview* 2019).

O *Docker* fornece a capacidade de empacotar e executar uma aplicação sobre um ambiente isolado chamado de *container* (*Docker overview* 2019), sendo este caracterizado como uma máquina virtual muito leve. O *Docker* permite que sejam criadas aplicações dentro de *containers* de forma a que sejam partilhadas através do *Docker Hub*, repositório central de *containers*. Desta forma, qualquer *developer* pode executar *containers* que reproduzem ambientes de produção (C. Anderson 2015).

O isolamento e a segurança permitem a execução de muitos *containers* simultaneamente sobre um determinado *host*. Os *containers* são leves porque não precisam da carga extra de um *hipervisor*, mas são executados diretamente no *kernel* da máquina *host* (C. Anderson 2015).

2.5 Métodos ágeis de desenvolvimento de software

Esta secção irá apresentar as metodologias ágeis de desenvolvimento de *software* que servirão de auxilio para a compreensão deste estudo de caso, nomeadamente o *Extreme Programming*, *Scrum* e *Test-Last Development*. O *Extreme Programming* apresenta-se como uma das primeiras metodologias ágeis de desenvolvimento de *software* (Jeffries, A. Anderson e C. Hendrickson 2001), e o *Scrum* como uma metodologia ágil de gestão de projetos, também aplicada ao desenvolvimento de *software*.

2.5.1 Extreme Programming

Segundo (Beck e Fowler 2001), o *Extreme programming* (XP) consiste na integração constante e testes automatizados, com pequenos lançamentos frequentes que incorporam *feedback* contínuo do cliente final. O trabalho em equipa faz do XP uma metodologia excecionalmente flexível e eficaz para o desenvolvimento de *software*. "Antes considerada radical, o *Extreme Programming* (XP) está rapidamente a tornar-se reconhecida como uma abordagem particularmente adequada para equipas pequenas que enfrentam requisitos vagos ou que mudam rapidamente, ou seja, a maioria dos projetos no atual mundo de desenvolvimento de *software* acelarado".

A filosofia chave do XP: O planeamento não é um evento único, mas um processo constante de reavaliação e correção de curso ao longo do ciclo de vida do projeto (Beck e Fowler 2001). Dentro deste contexto de flexibilidade e mudanças rápidas, o planeamento é crítico.

Sem isso, os projetos de *software* podem desmoronar rapidamente. Escrito por autores reconhecidos do XP Kent Beck e Martin Fowler, o XP apresenta métodos e abordagens necessárias para planear e gerir um projeto.

As práticas de XP não incluem a preparação de extensos requisitos ou documentação do projeto antes de iniciar o desenvolvimento. Consequentemente, o XP depende muito da comunicação contínua entre as partes interessadas, dos constantes ciclos de *feedback* para esclarecer e especificar a implementação dos requisitos e desta forma, responder às mudanças de forma orgânica (Layman et al. 2006). O XP apresenta-se como uma metodologia ágil, com o propósito de proporcionar flexibilidade às equipas de desenvolvimento de *software* e a todos os outros intervenientes. Permite a mudança e adaptação dos requisitos sem comprometer o custo que isso possa causar às partes interessadas.

Valores do XP "Os valores do XP são comunicação, simplicidade, *feedback* e coragem"(Lindstrom e Jeffries 2004). O propósito do XP é simples, juntar todas as partes interessadas e comunicar constantemente, utilizando métodos simples de planeamento, acompanhamento e *feedback*, de forma a orientar o projeto às necessidades que surgem. Trabalho e colaboração de forma conjunta dá coragem à equipa (Jeffries, A. Anderson e C. Hendrickson 2001). Este conjunto de valores proporcionam às equipas autonomia e simplicidade nos processos. A comunicação é chave para que a informação flua, sendo um dos principais problemas é a interpretação, que pode levar a conclusões e orientações no sentido oposto daqueles que foram os requisitos iniciais, a comunicação e o constante *feedback* tem o propósito de minimizar a incerteza, impulsionando a coragem e a simplicidade dos processos. "[...] O principal atributo dos membros da equipa deve ser um forte compromisso de entregar o projeto e atingir as suas metas"(Jeffries, A. Anderson e C. Hendrickson 2001).

2.5.2 Scrum

O SCRUM tem como principal objetivo gerir projetos que se encontram numa situação onde é difícil planear com antecedência e os ciclos de *feedback* constituem o elemento central, podendo os requisitos mudar mais regularmente (Schwaber e Beedle 2002). O SCRUM não é apenas um método de desenvolvimento iterativo e incremental, é também um método de desenvolvimento de *software* adaptativo, fornecendo técnicas de auto-organização, organização de processos, criação de conhecimento, etc (Beedle et al. 1999). Esta metodologia apresenta-se assim como uma metodologia ágil, onde tem a capacidade de criar equipas auto-suficientes que conseguem responder rapidamente à mudança. Segundo (Beedle et al. 1999), o SCRUM permite-nos construir *software* mais suave, pelo que não adianta tentar escrever todos os requisitos antecipadamente, sendo que novos problemas surgem no sistema em um novo contexto e como tal, precisamos de outra abordagem mais suave para construir *software*.

Os SCRUM é caracterizado como sendo uma prática onde o *software* é desenvolvido por uma equipa auto-organizada em *sprints*, podendo ter a duração de entre 2 semanas a 4 semanas, sendo que cada *sprint* começa com o planeamento e termina com uma revisão (Schwaber e Beedle 2002). O objetivo é entregar o máximo de *software* de qualidade possível dentro de uma *sprint* (Beedle et al. 1999). As iniciativas que irão compor o *sprint* são provenientes do *backlog*, componente que é responsável por conter as iniciativas que irão ser realizadas pela equipa de desenvolvimento. Depois de reunir e redefinir prioridades as iniciativas restantes e

novas, um novo *backlog* é formado e um novo *sprint* é iniciado. A entidade responsável por decidir quais as iniciativas a entrar no *sprint* é o *Product Owner*, sendo ele quem compõe a próxima *sprint*, que irá ser executada pela equipa de desenvolvimento. O seu papel é gerir e executar todas as iniciativas da *sprint*, e quando existe um bloqueio ou problema, o *scrum master* é responsável por resolver os problemas que impedem a equipa de trabalhar efetivamente, destacando-se também como o elemento desbloqueador (Schwaber e Beedle 2002).

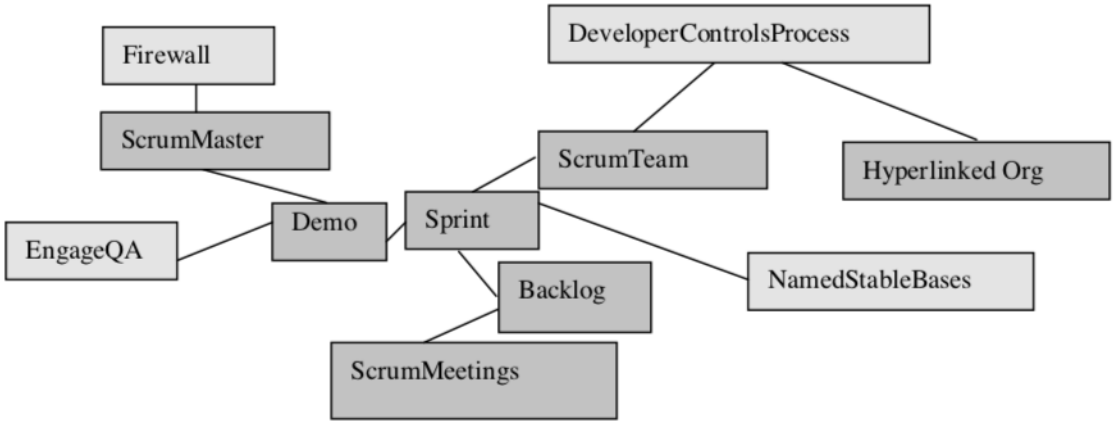

Figura 2.2: SCRUM - Linguagem padrão (Beedle et al. 1999)

A figura 2.2 apresenta a linguagem ou termos padrão que são utilizados no SCRUM, e a relação entre eles. Segundo (Beedle et al. 1999) a "socialização do conhecimento"leva a uma estrutura de equipa auto-organizada, onde o processo é inventado de maneira adaptável no dia a dia. No final de cada *sprint*, há uma demonstração para mostrar ao cliente o valor que foi concretizado e entregue (EngageCustomer), proporcionar à equipa de desenvolvimento a sensação de realização (CompensateSuccess), integrar e testar uma parte razoável do *software* que está a ser desenvolvido (EngageQA) e por último garantir progresso real que consiste na redução do *backlog* (NamedStableBases).

Segundo (Beedle et al. 1999), o SCRUM é composto por reuniões que tem como propósito gerir os vários momentos da equipa e da *sprint*, não existindo um processo pré-definido dentro de um *sprint*, em vez disso, as reuniões conduzem a conclusão das atividades alocadas, permitindo que a equipa de desenvolvimento "socialize o conhecimento dos membros da equipa". O conjunto de reuniões é composto por uma reunião antes de se iniciar o *sprint*, para que sejam discutidas as iniciativas que irão compor o próximo *sprint*. Durante a *sprint*, os membros da equipa tem uma reunião diária, tipicamente de quinze minutos, onde debatem e coordenam o trabalho, expondo quais os itens concluídos, os bloqueios que estão a impedir a progressão e a gestão de atribuição de tarefas aos membros da equipa. Por fim, é realizada uma última reunião para fazer a revisão da *sprint*.

2.6 Processos de desenvolvimento de software

"O processo de desenvolvimento de *software* consiste nas atividades e informações associadas que são necessárias para desenvolver um sistema de *software*. Cada organização tem seu próprio processo de *software* específico, mas essas abordagens individuais geralmente seguem um modelo de processo genérico mais abstrato"(Sommerville 1996).

Desenvolver e manter sistemas de *software* envolve uma vasta variedade de atividades relacionadas entre si, e de modo a gerir todo o processo vários modelos foram desenvolvidos ao longo dos anos. Estes incluem o modelo *Waterfall*, Desenvolvimento iterativo, *Prototyping*, Modelo espiral e RAD. Após esta era, os modelos de processo ágeis como *XP*, *Scrum* e *Kanban* começaram a ganhar a sua relevância, pelo facto de proporcionarem menos *stress* na análise e no design, a implementação começa muito cedo no ciclo de vida do desenvolvimento de *software* e permite que as equipas mudem rapidamente às mudanças. O desaparecimento de modelos mais antigos são fruto das rápidas mudanças e adaptações ao ambiente de desenvolvimento, permitindo a evolução para modelos que melhoram a qualidade das equipas de desenvolvimento de *software* (Kaur e Sengupta 2013).

As secções seguintes irão abordar processos de desenvolvimento de *software* enquadrados em metodologias ágeis, como o *Test-Driven Development*, o *Acceptance Test-Driven Development* e por fim o *Behavior-Driven Development*.

2.6.1 Test-Driven Development

O TDD apareceu inicialmente como parte da disciplina *Extreme Programming* 2.5.1, descrita em *Exetreme Programming Explained*, publicado em 1999. Em 2002, Beck lançou *Test-Driven Development: By example* e Dave Astels *Test-Driven Development: A Practical Guide*, formando a base desta metodologia.

Como referido na secção 1.1.3, o TDD propõe a inversão do fluxo tradicional de *software*, que é desenvolver primeiro e escrever os testes no final. Segundo (Beck 2003), o TDD é uma forma de gerir o medo durante o processo de desenvolvimento de *software*. Uma metáfora que representa aquilo a que o TDD se compromete a resolver: Ao invés de ser um processo experimental, o objetivo é começar a aprender e desenvolver mais concretamente, evitando a ocultação de problemas e bloqueios, ou seja, comunicar de forma mais clara e aberta, indo ao encontro de *feedback* útil e concreto.

O TDD chamou a atenção de uma grande comunidade de *developers* de *software* que considera ser uma forma rápida e eficaz de produzir código fiável. o TDD incorpora elementos de *design* e *refactor*, fazendo com que o processo de desenvolvimento de *software* flua. E como outras artes, a única maneira de realmente entender o TDD é praticá-lo (Jeffries e Melnik 2007). E para que possa ser praticado, o TDD precisa de ser entendido na sua integra. Segundo (Beck 2003), o TDD é definido pelo seguinte conjuntos de premissas:

- Escrever sempre um teste automático que falhe antes de escrever qualquer código.
- Eliminar a duplicação.
- Desenhar soluções organicamente, com o código em execução fornecendo *feedback* entre as decisões.
- O autor do código deve escrever os próprios testes.
- O ambiente de desenvolvimento deve dar uma resposta rápida a pequenas alterações.
- Os projetos devem consistir em muitos componentes altamente coesos e fracamente acoplados, de forma a facilitar o teste.

Tabela 2.1: Características dos testes (Beck 2003)

Característica	**Definição**
Fáceis de escrever	Um teste fácil de escrever dá indícios de que o *software* está simples e preciso. Se existir complexidade ao escrever o teste, podemos estar presente a um cenário em que o código poderá não estar correto.
Fáceis de ler	Os testes serão mais valiosos se forem legíveis, dando uma segunda perspectiva sobre qual o comportamento do código fonte.
Rápidos de executar	Os testes devem dar uma rápida resposta às mudanças, e para isso o tempo de resposta tem de ser curto.
Isolamento	O sucesso ou o fracasso de um teste seria irrelevante para outros.
Determinísticos	Dado um conjunto de entradas, o resultado deve ser sempre igual.
Parciais	É esperado poder escrever os testes uns de cada vez.
Combináveis	É expectável poder executar os testes com qualquer combinação.
Versionáveis	A fonte dos testes deve funcionar corretamente com o resto da fonte no sistema.
À *priori*	Devemos ser capazes de escrever os testes antes de serem executados.
Automáticos	Os testes devem ser executados sem intervenção humana.
Úteis quando se pensa em *design*	Escrever os testes à *priori* deve ser uma experiência de aprendizagem. (Jeffries e Melnik 2007) firmam que o TDD é uma atividade de *design* e programação, não uma atividade de teste em si.

Assim sendo, quem pratica o TDD tem acesso às iterações curtas e rápidas que esta metodologia permite, observando as mudanças e o resultado que cada operação reflete, de forma segura e eficaz (Janzen e Saiedian 2008).

Uma das partes mais importantes do TDD é a realização de testes. A abordagem do TDD amplia a afirmação feita por Boris Beizer em 1983 (Binder 2000): "O ato de designar testes é um dos mais eficazes detetores de bugs conhecidos". São eles que permitem validar o comportamento do *software*, dar *feedback* contínuo e contribuem para o nível de confiança da aplicação. Segundo (Beck 2003), a tabela 2.1 apresenta as principais características que os testes devem ter.

Red, Green, Refactor O processo de alto nível que constitui o TDD é designado como "Red, Green, Refactor". Segundo (Beck 2003) este ciclo repetitivo é o a base do TDD, permitindo que seja aplicado de forma correta. "À medida que seguimos esse ciclo simples, o programa cresce e o design evolui com ele" (Jeffries e Melnik 2007). No início de cada ciclo, todos os testes tem de estar aprovados obrigatoriamente, com exceção do que está a ser criado, o responsável pelo desenvolvimento deste novo ciclo. No final do ciclo, é expectável que o *developer* execute todos os testes, garantindo que todos fiquem aprovados. Na figura 2.3 é apresentado este ciclo de forma ilustrativa.

Como podemos observar na figura 2.3, os três componentes podem ser sintetizados da seguinte forma segundo (Beck 2003):

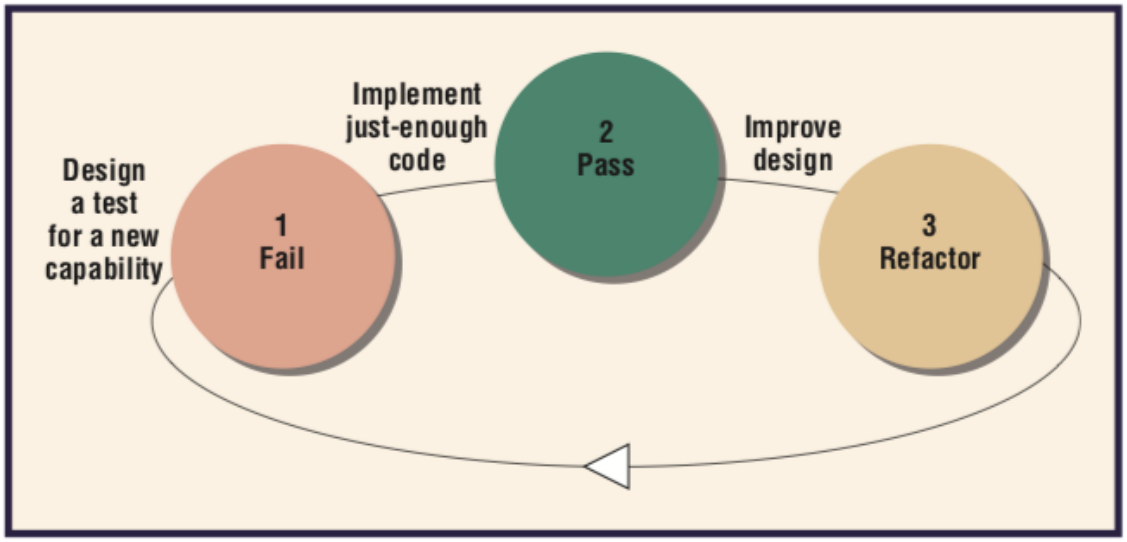

Figura 2.3: Ciclo Red-Green-Refactor (Jeffries e Melnik 2007)

- Vermelho (*Fail*): envolve a escrita de um pequeno teste que não funciona, talvez nem compile no início.
- Verde (*Pass*): envolve o teste funcionar rapidamente, implementando o código fonte.
- *Refactor*: envolve eliminar toda a duplicação criada apenas para fazer o teste funcionar e alterações de código para melhorar o design, passo a passo.

Cada nova unidade de código requer uma repetição deste ciclo. O TDD é uma metodologia de programação em que as atividades de codificação, teste e design são fortemente entrelaçadas (*Guidelines for Test-Driven Development* 2006). Todos os testes criados são executados de forma contínua para garantir que novas alterações não comprometem o comportamento já validado. Com isto, evita-se regressões manuais e trabalho humano.

O seguinte exemplo dado por (Beck 2003) reflete todos os passos necessários para seguir o padrão "Red, Green, Refactor"representado na figura 2.3:

1. Escreva um novo caso de teste.
2. Execute todos os casos de teste e veja o novo falhar.
3. Escreva apenas código suficiente para fazer o teste passar.
4. Volte a executar os casos de teste e veja todos a passarem.
5. *Refactor* para remover a duplicação e melhorar o design.

Durante o ciclo de vida útil de um *software*, a tendência é cair para o estado de *legacy*, a menos que o processo de *refactor* seja feito de forma contínua. Sem um conjunto de testes de regressão completos, o *refactor* é praticamente impossível porque não será possível saber quais partes do *software* são afetadas por uma alteração. O TDD inclui *refactor* contínuo, de modo a que o desenho e o código seja sempre mantido (Kumar e Bansal 2013).

TDD na correção de bugs O ciclo de "Red, Green, Refactor"tanto pode ser aplicado ao desenvolvimento de novas funcionalidades como na correção de problemas (Ficco, Pietrantuono e Russo 2011).

O seguinte processo é um exemplo de como pode um *bug* ser resolvido tendo em conta as premissas do TDD.

1. Escrever um teste que falha devido ao *bug*.
2. Corrigir o *bug* no código.
3. Correr o teste e verificar que ficou aprovado.
4. Correr todos os testes e verificar a aprovação.

Considerado por muitos autores como (Jeffries e Melnik 2007), (Kumar e Bansal 2013), (Ficco, Pietrantuono e Russo 2011) e (Janzen e Saiedian 2008), a correção e prevenção de erros é uma das maiores valias do TDD.

2.6.2 Acceptance Test–Driven Development

A origem do *Acceptance Test–Driven Development* ou ATDD foi evidenciada por (Beck 2003) no seu livro de modo breve, como que o autor a considerasse impraticável. No entanto, o ATDD revelou-se uma prática relevante ao longo do tempo (*Acceptance Test Driven Development (ATDD)* 2018) (Sauvé, Abath Neto e Cirne 2006). Tal como TDD, o ATDD também requer a criação de testes antes do código, e esses testes representam o comportamento que o *software* de ter (Andersson, Bache e Sutton 2003).

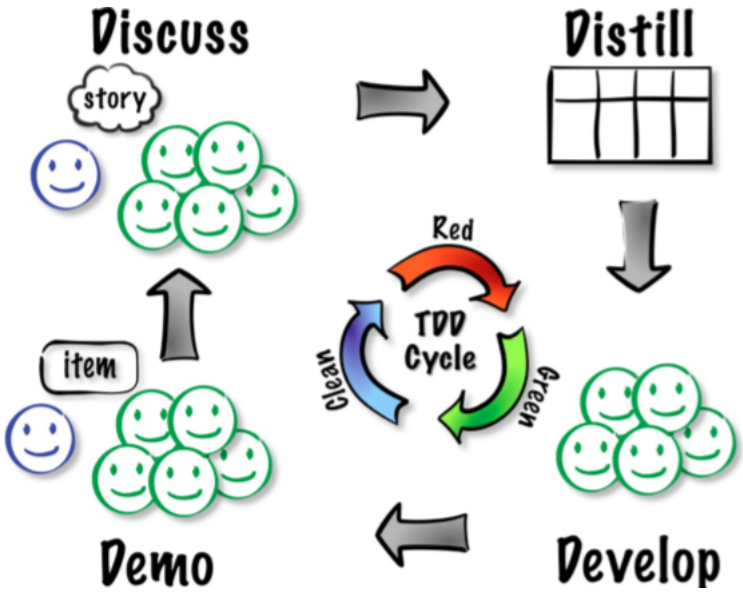

Figura 2.4: Ciclo de etapas do ATDD (E. Hendrickson 2008)

A figura 2.4 representa o ciclo do ATDD. A equipa cria um ou mais testes de aceitação para uma história ou recurso antes de dar início ao trabalho sobre ele, sendo que normalmente esses testes são discutidos e capturados no momento em que a equipa está a trabalhar com as parte interessadas, processo que serve para entender ou capturar um requisito que será uma história no *backlog*. Os testes de aceitação tornam-se requisitos executáveis,

fornecendo *feedback* continuo, dando uma indicação clara do progresso que a equipa está a alcançar. Um requisito é satisfeito se todos os testes ou critérios de aceitação associados forem satisfeitos. No ATDD, os testes de aceitação podem ser automatizados.

No caso de estudo de (E. Hendrickson 2008), concluiu-se que as equipas que implementaram o ATDD evidenciaram que o ato de definir testes de aceitação no momento de discussão dos requisitos resulta em uma melhor compreensão. Os testes do ATDD obrigam a chegar a um acordo concreto sobre o comportamento exato que o *software* deve ter. Equipas que seguem todo o processo, automatizando os testes à medida que implementam o recurso, geralmente descobrem que o *software* resultante torna-se fácil de testar e em geral, adicionar novos testes é um processo trivial.

Apesar das vantagens que o ATDD pode proporcionar a quem o adota, segundo (Solis e Wang 2011) muitos *developers* confundem-se ao usar TDD e ATDD em simultâneo, sentindo que a metodologia se tornava confusa, pois não há um processo bem definido para colocar em prática o ATDD. Alguns dos problemas do TDD e ATDD são que eles estão focados em verificar o estado do sistema, ao invés de comportamento desejado do sistema. O facto de a linguagem natural para descrever casos de teste não ser estruturada, faz com que exista liberdade para que cada um "normalize"de acordo com a sua interpretação, tornando os casos de teste difíceis de entender.

2.6.3 Behavior-Driven Development

O desenvolvimento orientado ao comportamento (BDD) (*Introducing BDD* 2006) é uma abordagem de desenvolvimento ágil cada vez mais relevante e adotada nos últimos anos, ganhando assim atenções de pesquisa e prática. O BDD é geralmente considerado como a evolução do TDD e ATDD. O BDD tem como objetivo dar resposta às especificações do comportamento que o sistema deve ter, de forma a automatizar esse processo, validando-o (Solis e Wang 2011) (*Introducing BDD* 2006).

O BDD permite verificar com segurança se todos os requisitos funcionais foram tratados adequadamente pelo código-fonte, conectando a descrição textual desses requisitos aos testes. Os testes são facilmente compreensíveis, pois o BDD usa uma linguagem comum que ajuda as partes interessadas a especificar os seus testes (Carvalho, Manhães et al. 2010).

Segundo (De Carvalho e De Campos 2006), o BDD é uma técnica de especificação que "certifica automaticamente que todos os requisitos funcionais são tratados adequadamente pelo código-fonte, através da conexão da descrição textual desses requisitos a testes automatizados". O BDD começa com descrições textuais dos requisitos usando palavras-chave específicas que marcam o tipo de frase, indicando como a frase será tratada nas fases subsequentes de desenvolvimento. A utilização de palavras-chave comuns fazem do BDD uma linguagem omnipresente e simples.

- As a [X]
- I want [Y]
- so that [Z]

Y representa um recurso, Z é o benefício ou valor do recurso e X é a pessoa que se beneficiará. A vantagem desta abordagem é identificar no momento da definição da história o valor que

Tabela 2.2: BDD Cenário - O cliente levanta dinheiro

Titulo: O cliente levantar dinheiro
As a cliente I want retirar dinheiro de um caixa multibanco so that Eu não tenha que esperar na fila do banco

a sua entrega trará, facilitando a descrição dos requisitos, eliminando alguma redundância ou incerteza.

Utilizando o exemplo dado por (*Introducing BDD* 2006), na tabela 2.2 podemos observar um exemplo de um cenário de teste, e com base nesse exemplo, existem vários cenários a considerar: a conta tem crédito suficiente, a conta não tem crédito, a conta contém crédito mas a caixa multibanco não, etc. Para a definição destes cenários, o BDD defende também uma nomenclatura específica, sendo que o comportamento de uma história será ditada pelo seu critério de aceitação. O conjunto de pontos seguintes representa o modelo para capturar os critérios de aceitação de uma história.

- Given [Q]
- When [W]
- Then [E]

Q representa o contexto inicial do cenário, o W é a fase em que todos os eventos ocorrem e por fim, o E é a fase de validação dos resultados.

Usando o modelo *Given-When-Then*, os dois primeiros cenários podem ser representados pela seguinte forma, segundo (*Introducing BDD* 2006):

```
1 Given the account is in credit
2 And the card is valid
3 And the dispenser contains cash
4 When the customer requests cash
5 Then ensure the account is debited
6 And ensure cash is dispensed
7 And ensure the card is returned
```

Listing 2.7: BDD Cenário: A conta tem crédito

```
1 Given the account is overdrawn
2 And the card is valid
3 When the customer requests cash
4 Then ensure a rejection message is displayed
5 And ensure cash is not dispensed
6 And ensure the card is returned
```

Listing 2.8: BDD Cenário: A conta não tem dinheiro

Seguindo este contexto, o BDD revela capacidade de compreensão sobre todas as partes interessadas, criando uma linguagem comum que facilita a compreensão e a comunicação. Com a descrição dos cenários, a entrega de valor é claramente identificada. O processo

de desenvolvimento acaba por beneficiar deste modelo, pois permite um maior foco nos requisitos e no cenário, evitando desperdício.

2.6.4 Test-Last Development

Test-Last Development ou TLD, segundo (C. M. Thomas 2014) é atualmente um método de teste popular e geralmente é o primeiro método de teste que as pessoas tendem a implementar. Este método nasceu proveniente do modelo de desenvolvimento de *software* em cascata em 1970 (Waterfall), onde o *software* é desenvolvido sobre uma série de fases: requisitos, design, implementação e a fase final é a fase de verificação onde o teste e a depuração ocorrem.

O TLD consiste na prática de escrever testes após o código ter sido escrito para efeito de verificação da funcionalidade do código escrito. Esses testes são usados pelo autor do código para corrigir o código até que nenhum outro erro seja encontrado pelos testes (C. M. Thomas 2014), sendo que os testes de unidade devem ser feitos somente após o código fonte (ou de recurso) ter sido finalizado (Bissi, Neto e Emer 2016).

2.7 Revisão das técnicas TDD e BDD a nível industrial

A seguinte revisão das técnicas TDD e BDD a nível industrial tem como objetivo apresentar, sobre um determinado contexto, experiências, desafios e resultados de diferentes estudos de caso realizados em empresas de desenvolvimento de software.

2.7.1 TDD

Esta secção irá apresentar o resultado de casos de estudo a nível industrial onde foi aplicado o TDD e por fim um conjunto de fatores que limitam a sua adoção.

(Rendell 2008) apresenta um caso de estudo, realizado na *T-Mobile*, sobre um projeto de longa duração incorporado numa equipa de trabalho que utilizava metodologias ágeis. A iniciativa de aplicar TDD foi apresentada à equipa como opcional, não existindo a obrigatoriedade de o aplicar. O projeto em questão tinha uma data bem definida para sua entrega (*Date-Driven*). Como conclusões, (Rendell 2008) refere que o TDD "parece ser uma técnica que muitos *developers* estão cientes, mas falta a experiência necessária que lhes permite utilizá-lo de forma mais eficaz". Quando os *developers* obtêm a experiência necessária no TDD e o aplicam no seu dia-a-dia, melhoram a estrutura do código. (Rendell 2008) refere ainda que o TDD não foi responsável pelo sucesso do projeto, foi no entanto, uma das muitas técnicas importantes aplicadas pela equipa que permitiram que a *T-Mobile* atingisse um grau de adaptabilidade mais elevado do que a exibida por projetos semelhantes, sem sacrificar seu compromisso e a sua qualidade. Como observações, (Rendell 2008) relata que: a aplicação dogmática do TDD poderia levar ao aumento de custos sem o retorno correspondente do investimento, para *developers* mais experientes, o uso do TDD ajuda a tornar o código mais testável, e por fim, "um *developer* pode ser brilhante o suficiente para produzir código de baixo defeito e bem estruturado sem codificar um teste primeiro. Isso não ajuda os *developers* subsequentes, possivelmente menos capazes, a manter esse mesmo código sem o suporte da alta cobertura de teste".

O estudo de caso apresentado por (Nagappan et al. 2008) realizado sobre quatro equipas de desenvolvimento de *software* (três estudos da *Microsoft* e um na *IBM* respetivamente) que adotaram a prática TDD, relata que o TDD foi aplicado e adaptado aos processos da empresa em questão e não sobre o contexto de XP. As equipas trabalhavam sobre a metodologia *Waterfall*. Como resultado deste estudo de caso, o TDD mostrou-se aplicável "em vários domínios e pode reduzir significativamente a densidade de defeitos do *software* desenvolvido, sem redução significativa da produtividade da equipa de desenvolvimento. Versões futuras dos produtos desenvolvidos sobre a prática TDD, à medida que continuam com uso do TDD, também terão baixa densidade de defeitos devido ao uso ativo de testes de unidade e integração que esta prática impõe às soluções de *software*"(Nagappan et al. 2008). Os resultados dos estudos indicam que a densidade de defeitos dos quatro produtos diminuiu entre 40% e 90% em relação a projetos semelhantes que não utilizaram a prática de TDD. Subjetivamente, as equipas evidenciaram um aumento de 15 a 35% no tempo de desenvolvimento inicial após a adoção do TDD.

(Jeffries e Melnik 2007) apresenta uma revisão de estudos industriais e académicos sobre equipas e projetos que adotaram o TDD, projetos estes *legacy* e não *legacy* sobre diferentes linguagens de programação (Java, C++, C# ...). A nível industrial, a maioria dos estudos de caso concluiu: aumento significativo da qualidade do *software*, diminuição da tolerância a falhas e um aumento da produtividade. Apenas dois casos da amostra analisada deram como inconclusivo o impacto do TDD na qualidade do *software*.

(B. George e Williams 2003) realizou um conjunto de experiências estruturadas com 24 *developers* profissionais. Um grupo desenvolveu o código usando TDD, enquanto o outro, uma abordagem semelhante ao modelo *Waterfall* (Cascata). Como resultado, (B. George e Williams 2003) refere que a abordagem TDD parece produzir código com qualidade acentuada, os *developers* do grupo TDD gastaram mais tempo (16%) do que os *developers* do grupo *Waterfall* nos seus desenvolvimentos. Em média, 80% dos *developers* afirmaram que o TDD é uma abordagem eficaz e 78% acreditam que a abordagem melhora a sua produtividade. "Qualitativamente, esta pesquisa também descobriu que a abordagem TDD facilita o design mais simples e que a falta de design inicial não é um obstáculo. No entanto, para alguns, a transição para a mentalidade do TDD é difícil"(B. George e Williams 2003).

(Janzen 2005) apresenta uma pesquisa sobre estudos empíricos de engenharia de *software* aplicados em ambiente profissional para avaliar a influência do TDD na qualidade do *software*, demonstrando que os programadores que usam o TDD produziram código que passou até 50% mais de testes externos do que o código produzido pelos grupos de que não usam o TDD. Houve também uma diminuição significativa no grupo que utilizou TDD no tempo gasto tem depuração (*debug*) em relação ao outro grupo. Por último, a complexidade computacional é muito menor e o volume e a cobertura do teste são maiores no código realizado em TDD.

Desafios na adoção do TDD

(Causevic, Sundmark e Punnekkat 2011) aponta sete razões mais evidentes que limitam a adoção do TDD em ambiente industrial, com base em cerca de 48 casos de estudo:

- Maior tempo de desenvolvimento: Tempo necessário para implementar um determinado conjunto de requisitos. Este pode ser um fator crítico e decisivo quando as organizações ponderam a sua adoção. Segundo (Causevic, Sundmark e Punnekkat 2011),

dependendo da maturidade da organização, uma perda inicial (neste caso, maior tempo de desenvolvimento) pode ofuscar um ganho a longo prazo (por exemplo, diminuição do tempo total do projeto ou aumento da qualidade do produto - ambos relatados em muitos estudos). Portanto, a pressão organizacional interna pode arriscar o uso adequado de TDD.

- Pouca experiência/conhecimento do protocolo TDD: A falta de treino, formação e prática fazem com que o pouco conhecimento deste protocolo se torne um obstáculo a quem a quer adotar. (Kumar e Bansal 2013) refere ainda que "como o TDD é uma nova abordagem, não temos muita experiência nesse domínio em comparação com a abordagem tradicional".
- *Design* insuficiente: Este fator refere-se ao processo de desenhar o *software*. Com base em (Causevic, Sundmark e Punnekkat 2011), alguns casos de estudo apontam para problemas relacionados com a adoção do TDD, referindo que a qualidade do desenho diminui, principalmente em projetos de maior escala e complexos. No entanto, não há apoio empírico em massa que mostre que a falta de desenho deve ser considerado como um fator limitante para a adoção industrial do TDD.
- Pouca experiência/conhecimento na realização de testes por parte dos *developers*: Capacidade do *developer* produzir testes automáticos de forma eficiente e eficaz. O TDD é uma técnica onde os testes guiam o desenvolvimento, e como tal, se o conhecimento for escasso na realização de testes, pode comprometer a adoção e a boa prática do TDD.
- Adesão insuficiente ao protocolo TDD: O TDD é um protocolo que é composto por diretrizes bem definidas e exatas, como a escrita de um teste automatizado antes da escrita do código, e este fator reflete a pouca adesão a estas diretrizes. Segundo (Causevic, Sundmark e Punnekkat 2011), as razões para abandonar o protocolo TDD incluem pressão de tempo, falta de disciplina, desvio das diretrizes do TDD e prazos curtos de desenvolvimento.
- Limitações específicas do domínio e da ferramenta: problemas técnicos na implementação do TDD, ou seja, dificuldades na realização de testes automatizados, (em GUIs por exemplo). O TDD requer suporte de ferramentas que proporcionem as condições necessárias à realização e execução de testes automáticos, o suporte adequado de ferramentas é vital para a adoção bem-sucedida do TDD.
- Código *legacy*: código existente que representa anos ou décadas de desenvolvimento e investimento e serve como *backbone* para produtos existentes e futuros. Kent Beck quando apresenta o TDD não refere como aplicá-lo em código *legacy*, transpondo a impressão de que todo o código é construído do zero. Segundo (Causevic, Sundmark e Punnekkat 2011), como isso raramente acontece nas empresas de desenvolvimento, a adoção do TDD pode ser problemática. A falta de bateria de testes automatizados dificulta a flexibilidade fornecida pelo *feedback* rápido que o TDD impõe.

2.7.2 BDD

Esta secção irá apresentar o resultado de casos de estudo a nível industrial onde foi aplicado o BDD.

(Pereira et al. 2018) apresenta um estudo empírico realizado com 24 profissionais com experiência prática em BDD, de diferentes empresas e com diferentes níveis de especialização em desenvolvimento ágil, de forma a identificar os benefícios e os desafios do BDD. Este estudo recolheu dados sobre como o BDD era entendido, como o BDD era aplicado, quais os seus desafios e quais os benefícios sobre diferentes projetos, concluindo que "o BDD melhora a comunicação e a colaboração em todas as funções de desenvolvimento de *software*, incluindo clientes", o BDD permite também que um projeto tenha *living documentation* (documentação viva), documentação que está sempre atualizada, sendo isto possível porque a abordagem do BDD depende do uso de cenários escritos sobre a linguagem *Gherkin*, e por último, (Pereira et al. 2018) refere que "os profissionais devem estar cientes de que o poder do BDD depende dos seus cenários, sendo eles base do BDD, e qualquer esforço para facilitar sua adoção, como treino, investimento em ferramentas e assim por diante, valerá a pena".

Desafios na adoção do BDD

(Pereira et al. 2018) apenas aponta que "a maioria dos desafios associados ao BDD está relacionada com a sua adoção, portanto, uma vez ultrapassada a curva de aprendizagem inicial, o BDD provavelmente produzirá benefícios".

Capítulo 3

Contexto e Análise de Valor

O presente capítulo irá abordar o contexto atual da organização em detalhe e apresentar a análise de valor.

3.1 Contexto atual

Nesta secção serão abordadas as metodologias e práticas atuais da organização e da equipa de trabalho, de modo a contextualizar o meio ambiente em que este estudo de caso se enquadra.

A organização utiliza um conjunto de ferramentas, técnicas e metodologias que apoiam as equipas de desenvolvimento durante o seu trabalho. Para uma caracterização mais clara, os seguintes pontos representados por: composição da equipa, processo de desenvolvimento, ferramentas e projetos.

- Composição da equipa

  A equipa é composta por quatro *developers*, um *Team-lead* (TL), um Product Owner (PO), e um Quality Assurance (QA). A tabela 3.1 apresenta as métricas relativamente à equipa. Em média, a equipa apresenta um nível de experiência de 8 anos, um conhecimento de negócio médio alto e um grande conhecimento da linguagem de programação C#.

- Processo de desenvolvimento

  O processo de desenvolvimento que a equipa utiliza é composto por práticas da *framework SCRUM* e utiliza *TLD*. As *sprints* tem a duração de duas semanas, sendo acompanhadas por todas as reuniões que o *SCRUM* propõe.

Tabela 3.1: Métricas da equipa de trabalho

Métrica	Dev1	Dev2	Dev3	Dev4	QA	TL	PO
Nível de experiência (anos)	3	10	10	1	4	12	12
Conhecimento de negócio (Baixo, Médio, Alto)	Médio	Alto	Médio	Baixo	Baixo	Alto	Alto
Linguagem de programação	C#	C#	C#	C#	C#	C#	-

Apesar da equipa de desenvolvimento utilizar metodologias ágeis para o processo de desenvolvimento de *software*, gestão de equipa e de eventos, o processo *TLD* é o que atualmente é utilizado, sendo precisamente a metodologia que se pretende substituir com este estudo de caso.

- Ferramentas de trabalho

  As ferramentas que a equipa utiliza são: Como *IDE* (ambiente de desenvolvimento integrado) o *Visual Studio*, visto que o ambiente *.NET* é utilizado na maioria dos projetos. Para testes de unidade é utilizado o *xUnit* com *MockObjects*. O *Selenium* é a ferramenta para testes de aceitação em alguns casos. No que diz respeito a integração contínua, o *TeamCity* e o *Jenkins* são as ferramentas para a finalidade, existindo uma *pipeline* que executa os estágios necessários até à implantação final das aplicações nos servidores.

- Projetos

  Os projetos que a equipa detém estão dentro do âmbito de *Web APIs*, nomeadamente *ASP.NET Web API*, sendo normalmente aplicações focadas em servir as necessidades de outras aplicações quem as consomem. A interface gráfica é quase nula, existindo assim uma vertente de *Backend* muito forte. A equipa detém também aplicações *Web* com interface gráfica, mas em menor número.

3.2 Análise de valor

Esta secção é composta pela análise de valor, que tem como objetivo identificar o valor que este estudo de caso trará ao seu publico alvo, nomeadamente a empresa e a equipa de desenvolvimento. A análise será acompanhada pelo novo modelo de conceitos (NCD), benefícios e sacrifícios e por último, o modelo Canvas, modelo que visa dar a conhecer a visão global do negócio.

Segundo (*Value Analysis* 2005), a análise de valor é uma abordagem para melhorar o valor de um produto ou processo, de forma a compreender os seus componentes e os seus custos associados, e em seguida, procurar encontrar melhorias nos componentes reduzindo o seu custo ou aumentando o valor das funções. Os conceitos chave da análise de valor são: Valor, Função, Análise de valor e Necessidade.

- Valor: a relação entre uma função para satisfação do cliente e o custo dessa função.
- Função: o efeito produzido por um produto ou por um dos seus elementos, com o propósito de satisfazer as necessidades do cliente.
- Análise de valor: metodologia para aumentar o valor de um produto ou processo, sendo que este produto ou processo pode ser novo ou existente ou novo.
- Necessidade: algo que é necessário ou desejado pelo cliente.

A análise de valor identifica-se assim como um processo de análise bem definido com o propósito de avaliar um produto ou serviço, minimizando o desperdício e outros efeitos que não trazem valor para o produto ou serviço em questão. Segundo (*Value Analysis* 2005), este processo é baseado na aplicação de um plano de trabalho sistemático que pode ser dividido em seis etapas: orientação e preparação, informação, análise, inovação e criatividade, avaliação e implementação e monitorização.

"Mesmo com o auxílio de processos, o resultado final pode não ser o mais desejado, sendo que o valor é um conceito subjectivo"(Woodall 2003). A secção 3.2.2 apresenta os benefícios e sacrifícios, que na ótica do cliente poderão ser interpretados como indicadores de valor quando existe um equilíbrio entre ambos.

Com a introdução de um novo processo de desenvolvimento com base no TDD e BDD, pretende-se aumentar a qualidade do *software* e a produtividade das equipas de trabalho. Processos como correção de código e identificação de problemas, normalmente são processos morosos e difíceis de realizar sem comprometer o estado atual do sistema. Com a introdução da nova abordagem, pretendesse detetar problemas prematuramente, diminuir as falhas e aumentar a qualidade do *software* produzido. Com isso, a produtividade irá aumentar de forma natural, criando valor para a organização que despende menos recursos, clientes finais tem acesso a mais valor de forma mais rápida e também para a equipa de trabalho, que trabalha sobre *software* mais fiável e seguro.

3.2.1 Novo Modelo de Desenvolvimento de Conceitos

Segundo (Koen et al. 2001), o Novo Modelo de Desenvolvimento de Conceitos (NCD) fornece uma linguagem comum sobre os principais componentes do Front End Difuso (FFE). A figura 3.1 representa o NCD. Este modelo é composto por três componentes (Koen et al. 2001):

1. A área interna define os cinco elementos-chave que compõem o Front End of Innovation (FEI).
2. O motor (parte central) é alimentado pela liderança e cultura da organização, que impulsiona os cinco elementos do front-end.
3. Os fatores influenciadores (setas) consistem em: Capacidades organizacionais, Estratégia empresarial, o Mundo Externo (ou seja, canais de distribuição, clientes e concorrentes) e a Ciência facilitadora que será utilizada.

Seguindo a ideologia de (Koen et al. 2001) e em analogia ao âmbito deste estudo de caso, os cinco elementos-chave que compõem o *Front End of Innovation* (FEI) serão analisados na seguinte enumeração:

1. Identificação da oportunidade

   Fase onde a organização identifica as oportunidades a seguir orientadas aos objetivos do negócio. A oportunidade pode ser uma resposta a curto prazo a uma ameaça competitiva, uma possibilidade inovadora para adquirir vantagem competitiva ou um meio de simplificar, acelerar ou reduzir o custo das operações (Koen et al. 2001).

   A proposta apresentada neste projeto surge da oportunidade de melhorar e otimizar o processo de desenvolvimento de *software*, tornado-o mais fiável, rentável e consequentemente aumentar a qualidade do *software* produzido. Esta oportunidade enquadra-se em acelerar e reduzir o custo das operações, neste caso, todo o processo de desenvolvimento e correção de *software*.

2. Análise da oportunidade

   Após a identificação da oportunidade, é necessário adquirir informações adicionais para traduzir a identificação da oportunidade em oportunidades específicas de negócio.

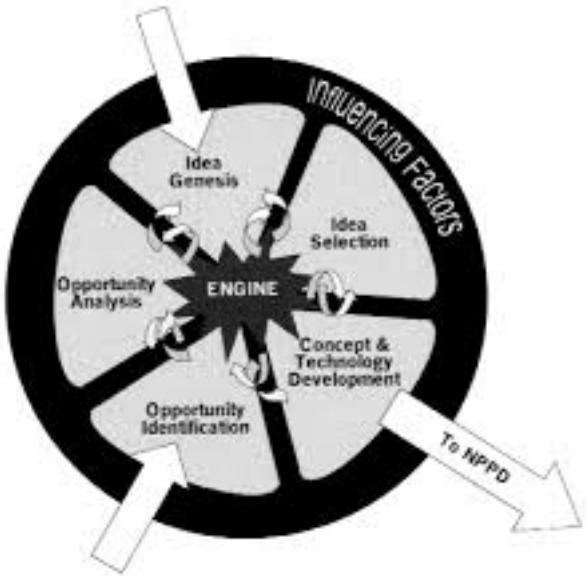

Figura 3.1: Novo modelo de desenvolvimento de conceitos (NCD) (Koen et al. 2001)

Métodos de análise da oportunidade podem ser aplicados, como estudos de mercado, experiências científicas, entre outras (Koen et al. 2001).

Após identificada a oportunidade e a respetiva proposta, foram feitas pesquisas sobre estudos e experiências já realizadas, com a aplicação da mesma proposta a ambientes similares (empresas de desenvolvimento de *software*, algumas delas utilizam inclusive o XP) e após analisar os prós e contras, concluiu-se que existe efetivamente uma oportunidade fortemente relevante para resolver os problemas a que este estudo de caso assume colmatar.

3. Génese da ideia

   É o processo de desenvolvimento e maturação da oportunidade em uma ideia concreta. A génese da ideia representa um processo evolutivo no qual as ideias são construídas, reformuladas, modificadas e atualizadas, passando por muitas iterações e mudanças à medida que são examinadas, estudadas, discutidas e desenvolvidas. O contacto direto com clientes e outras entidades como colaboradores, geralmente aprimoram esta atividade através de processos formais ou informais, *brainstorming* e até mesmo fruto de experiências negativas. O objetivo passa por promover ideias e oportunidades independentemente de onde possam surgir (Koen et al. 2001).

   Nesta fase, através de casos de estudo aplicados em outras empresas, espera-se extrair passos e métodos que foram concretizados com sucesso de modo a introduzir a nova metodologia eficazmente, acompanhado com sessões de *brainstorming*, *Coding Dojo*, partilha de conhecimento, etc, para que exista uma adaptação à nossa realidade, sendo esperado que todos os envolvidos neste processo possam contribuir organicamente para o propósito final.

4. Seleção de ideias

   A seleção da ideia consiste na aplicação de um filtro sobre todas as ideias identificadas até ao momento, de forma a selecionar as ideias teoricamente mais fortes, de forma

a alcançar o maior valor de negócio. É um processo difícil quando existe um grande volume de ideais, sendo que existe a possibilidade de se descartar ideias que poderiam ser muito vantajosas para a organização (Koen et al. 2001).

Nesta fase, foram pesados os gastos financeiros e temporais para o desenvolvimento da proposta em questão, e após a reflexão sobre o valor comercial e de consumo que esta metodologia poderá trazer para a organização, concluiu-se que a sua aplicabilidade tem grande probabilidade de trazer um valor comercial e de consumo superior ao gasto financeiro e temporal que acarreta.

5. Desenvolvimento do conceito

   Por último, o desenvolvimento do conceito envolve o desenvolvimento de um caso de negócio com base em estimativas do potencial de mercado, necessidades do cliente, requisitos de investimento, avaliações de concorrentes, incógnitas tecnológicas e risco geral do projeto (Koen et al. 2001).

   Através de reuniões com a equipa de trabalho e a organização, foram definidos os riscos da solução, as necessidades da equipa de trabalho, o investimento e o esforço necessário.

3.2.2 Benefícios e sacrifícios

Após a introdução ao valor na secção 3.2, segundo (Woodall 2003), o valor para o cliente pode ser caracterizado e observado através de uma perspetiva longitudinal. Na figura 3.2 podemos observar quatro posições temporais, sendo elas Pré-compra, Ponto de negócio, Pós-compra e por fim Após uso ou experiência.

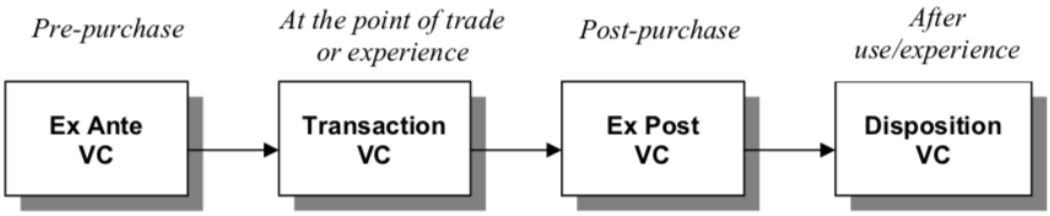

Figura 3.2: Perspetiva longitudinal em VC (Woodall 2003)

- Pré-compra: Etapa de previsão sobre os preconceitos que os clientes criam em relação ao VC no ato da compra.
- Ponto de negócio: Etapa que implica uma experiência e sensação de VC no ponto de negócio em tempo real.
- Pós-compra: Etapa que expõe os resultados de experiências com base nas escolhas dos clientes, sendo possível verificar o valor entregue.
- Após uso ou experiência: Etapa de observação relativamente ao valor entregue e ao ponto de venda.

Segundo o modelo de (Woodall 2003), na tabela 3.2 podemos observar os benefícios e os sacrifícios sobre a proposta apresentada, a introdução da metodologia desenhada TDD.

Tabela 3.2: Análise de Valor - Benefícios vs Sacrifícios

	Benefícios	Sacrifícios
Pré-compra	Qualidade Motivação Robustez Segurança	Mudança de paradigma Esforço Tempo
Ponto de negócio	Adaptação Motivação	Custo monetário Tempo Esforço
Pós-compra	Qualidade Garantia Segurança	
Após uso ou experiência	Qualidade Segurança Eficácia Eficiência Maior produtividade	

Após analisar a tabela 3.2, e de acordo com a definição de (Woodall 2003) sobre a perspetiva longitudinal em CV, o estágio de Pré-compra contempla como benefícios a qualidade, robustez e segurança do *software* e a motivação da equipa de trabalho. Como sacrifícios, o esforço e tempo que a equipa irá despender, bem como a mudança de paradigma provocado pela nova metodologia. O estágio de Ponto de negócio apresenta como benefícios a adaptação ao novo modelo bem como a motivação por parte da equipa. Como sacrifícios, o custo monetário que será maior nesta fase, bem como o tempo e o esforço de igual modo. O estágio de Pós-compra apresenta a qualidade, garantia e segurança do *software* como benefícios. Por último, o estágio de Após uso ou experiência, como benefícios expõe a qualidade e segurança do *software* e a eficiência, eficácia e maior produtividade por parte da equipa de trabalho.

3.2.3 Proposta de valor

Segundo (Martinez-Hernandez 2003), o valor reside na satisfação e realização das expectativas dos clientes, e de forma simultânea, gerar riqueza para as organizações. A proposta de valor deve deixar claro qual produto ou serviço em questão, os seus clientes alvo, o valor fornecido e seu valor exclusivo, sendo essencial para alertar novos clientes sobre o valor dos produtos e serviços de uma organização.

A introdução e aplicabilidade da metodologia documentada nesta dissertação tem como alvo a organização, a equipa de trabalho que abraçou o tema, e consequentemente, os clientes finais que irão utilizar os produtos finais. Esta metodologia tem como propósito melhorar a qualidade do *software* produzido, facilitar o processo de desenvolvimento, permitir de certa modo a capturar erros de forma prematura, aumentar a produtividade da equipa de trabalho e diminuir a quantidade de falhas nos produtos. Com isto, e de forma natural, o resultado

final previsto será *software* com mais qualidade, fiável, com a equipa a ter mais capacidade de resposta, beneficiando todos os envolvidos, bem como os *stakeholders*.

A abordagem a seguir será implementada pela primeira vez na equipa de trabalho, servindo esta dissertação como um estudo de caso, e espera-se que mais equipas adiram a esta metodologia dentro da organização.

3.2.4 Modelo de Canvas

O modelo Canvas (Business Model Canvas) proposto por Alexander Osterwalder (Osterwalder e Pigneur 2010), é uma ferramenta de gestão e planeamento estratégico que tem como propósito permitir a criação de um modelo de negocio ou melhorar um existente. Segundo (Osterwalder e Pigneur 2011), este modelo surgiu da necessidade de criar um modelo de negócio que todos entendam, que facilite a descrição e a discussão, para que partindo do mesmo ponto, diferentes pessoas com diferentes conhecimentos, consigam comunicar.

O modelo Canvas é caracterizado por um diagrama de nove blocos (Osterwalder e Pigneur 2011). Na figura 3.3 podemos observar a caracterização do modelo Canvas relativamente à proposta que é apresentada nesta dissertação.

<table>
<tr><th>Parceiros Chave</th><th>Atividades Chave</th><th>Proposta de Valor</th><th>Relação com os Clientes</th><th>Segmentos de Clientes</th></tr>
<tr><td rowspan="3">Organização
Equipa de desenvolvimento</td><td>Introdução e aplicação da nova metodologia</td><td rowspan="3">Metodologia ágil
Precisão e exatidão
Fiabilidade do software
Qualidade do software</td><td>Equipa de desenvolvimento</td><td rowspan="3">Equipa de desenvolvimento
Clientes finais</td></tr>
<tr><td>Recursos Chave</td><td>Canais de Distribuição</td></tr>
<tr><td>Metodologia de trabalho
Equipa de desenvolvimento</td><td>Plataforma de documentação
Partilha de conhecimento</td></tr>
<tr><th colspan="2">Estrutura de Custos</th><th colspan="3">Fonte de Receita</th></tr>
<tr><td colspan="2">Implementação da metodologia</td><td colspan="3">Redução do tempo de correção do software devido a compreensão deficiente dos requisitos
Redução dos problemas em ambientes de produção
Redução do tempo de manutenção
Redução do tempo em novos desenvolvimentos</td></tr>
</table>

Figura 3.3: Modelo Canvas

A seguinte enumeração contempla a relação e análise a cada bloco do modelo modelo de Canvas, representado na figura 3.3, sobre este estudo de caso:

1. Parceiros Chave

   Os parceiros chave constituem o conjunto de fornecedores e parceiros que apoiam o modelo de negócio e ajudam a realização da proposta de valor, sendo uma peça fundamental para otimizar processos, reduzir riscos ou adquirir recursos (Osterwalder e Pigneur 2010).

   - Organização: entidade responsável por criar o ambiente necessário para aplicar este estudo de caso.

- Equipa de desenvolvimento: entidade que irá abordar o desafio.

2. Atividades Chave

   As atividades chave caracterizam-se por tarefas que são executadas para construir a proposta de valor, chegar aos mercados ou obter rendimentos. São as atividades que produzem valor, fazendo do modelo de negócio rentável (Osterwalder e Pigneur 2010).

   - Introdução e aplicação da nova metodologia: adotar a metodologia de trabalho com o propósito de melhorar a qualidade do *software*, a rentabilidade e produtividade.

3. Recursos Chave

   Os recursos chave podem ser físicos, financeiros, intelectuais ou humanos, sendo os componentes fundamentais que servem para gerar valor para os clientes alvo, podendo pertencer à organização ou pertencer a outrem. São eles que irão fazer o nosso modelo de negócio funcionar (Osterwalder e Pigneur 2010).

   - Metodologia de trabalho: recurso chave para que fornece todos os conceitos e métodos necessários que contribuem para um *software* mais fiável, robusto e rápido de concretizar.
   - Equipa de desenvolvimento: será a entidade fulcral que irá implementar, validar e usufruir da metodologia em questão.

4. Proposta de Valor

   A proposta de valor é composta pela descrição da solução que a empresa propõe oferecer para resolver os problemas dos clientes alvo, quais os benefícios que o produto ou serviço oferece, e outras variantes como o preço. A proposta de valor deve ser inovadora e justificativa, pois é o fator pelo qual o publico alvo comprará ou não o produto ou serviço (Osterwalder e Pigneur 2010).

   - Metodologia ágil: Introdução e aplicação do TDD e BDD.
   - Precisão e exatidão: permite que apenas se escreva código de acordo com os requisitos, focando no problema a resolver.
   - Fiabilidade do *software*: tornará as aplicações mais fiáveis, existindo uma maior cobertura de código por parte dos testes, garantindo que o *refactor* do código seja feito de forma segura sem que comprometa o estado atual do sistema.
   - Qualidade do *software*: Diminuição de erros em produção e aumento da qualidade do desenho do *software*.

5. Relações com os Clientes

   O relacionamento como os clientes pode ser diferenciado de acordo com cada segmento, sendo caracterizado por três tipos: pessoais, baseada em interação humana, *self-service*, sendo a organização a fornecer todos os mecanismos necessários para o cliente satisfazer as suas necessidades e por fim, automatizadas, que funciona como o *self-service* mas de forma automatizada (Osterwalder e Pigneur 2010).

   - Equipa de desenvolvimento: existe uma relação de *feedback* contínuo, de forma a perceber se o valor que se pretende criar está a ser criado efetivamente, e uma relação de confiança no que diz respeito ao apoio que precisa para seguir a

metodologia em questão, de forma a garantir que o trajeto até ao objetivo está a ser seguido.

6. Canais de distribuição

   Os canais são os pontos de contacto entre a organização e os clientes alvo. Os canais podem ser caracterizados como canais de comunicação, distribuição e venda. É portanto, o caminho de entrega do valor aos clientes (Osterwalder e Pigneur 2010).

   - Plataforma de documentação: meio de comunicação utilizado para expor toda a documentação necessária relativamente à nova metodologia.
   - Partilha de conhecimento: meio de comunicação que servirá para partilha de conhecimento entre membros da organização, com o objetivo de expor experiências e facilitar o processo de aprendizagem que a nova metodologia obriga.

7. Segmentos de Clientes

   Este componente tem como propósito definir os diferentes grupos de pessoas ou organizações que uma empresa visa alcançar e servir, através das suas preferências, comportamentos, faixa etária, onde estão localizados, etc. Cada segmento deve ter por base a forma de como o produto ou serviço em questão afeta as necessidades de cada segmento. Dentro de um modelo de negócio pode existir um ou vários segmentos (Osterwalder e Pigneur 2010).

   - Equipa de desenvolvimento: entidade que irá participar no desenvolvimento das aplicações, usufruindo da metodologia e de todos os métodos aplicados.
   - Clientes finais: acesso a aplicações com menos erros e mais fiáveis.

8. Estrutura de Custos

   A estrutura de custos é responsável por descrever os principais custos envolvidos para a realização do modelo de negócio, independentemente de serem custos fixos ou variáveis (Osterwalder e Pigneur 2010).

   - Implementação da metodologia: a curva de aprendizagem tem custos pois envolve mais tempo e dedicação do equipa de desenvolvimento.

9. Fontes de Receita

   A fonte de receita representa os lucros que a organização gera a partir de cada segmento de cliente. Pode haver mais que uma fonte de receita dependendo do modelo de negócio. É através da compra do produto ou serviço por parte dos clientes que esta fonte de receita cresce (Osterwalder e Pigneur 2010).

   - Redução do tempo de correção do *software* devido a compreensão deficiente dos requisitos: o tempo despendido para resolver este tipo de problemas será menor, visto que a nova metodologia visa focar nos requisitos, obrigando a criar um maior foco sobre este aspeto.
   - Redução dos problemas em ambientes de produção: as equipas terão mais disponibilidade para desenvolver novas funcionalidades, sem que sejam interrompidas durante esse processo para a correção de problemas. No caso de surgirem problemas, a metodologia em questão promove a cobertura do problema (teste escrito), sendo a sua resolução mais rápida.

- Redução do tempo de manutenção: com uma maior cobertura de testes sobre o comportamento do sistema, a manutenção e *refactor* do sistema irá ser mais simples e rápida, sem comprometer o estado atual.
- Redução do tempo em novos desenvolvimentos: é expectável que o tempo de desenvolvimento aumente numa primeira fase, mas que diminua ao longo do tempo, sendo necessário menos tempo para novos desenvolvimentos.

Capítulo 4

Análise e Design

Este capítulo tem como objetivo contextualizar o estado atual dos processos, expondo os artefactos formais e analisar soluções alternativas que visam resolver os problemas descritos anteriormente na secção 1.2. Por fim, será apresentada a solução que melhor se adequa a este estudo de caso.

Serão utilizados diagramas *UML* (*Unified Modeling Language*, Linguagem de modelação unificada) de forma a expor os artefactos em questão.

4.1 Análise do estado atual

Nesta secção irá ser apresentado o estado atual dos processos, intervenientes, ferramentas e metodologias utilizadas pela equipa que irá abordar este estudo de caso.

4.1.1 Processo de desenvolvimento

Atualmente, a equipa que irá abordar este estudo de caso utiliza um conjunto de processos e metodologias referidas na secção 3.1. Os papéis envolvidos no processo de desenvolvimento podem ser observados através do diagrama de atividade presente na figura 4.1.

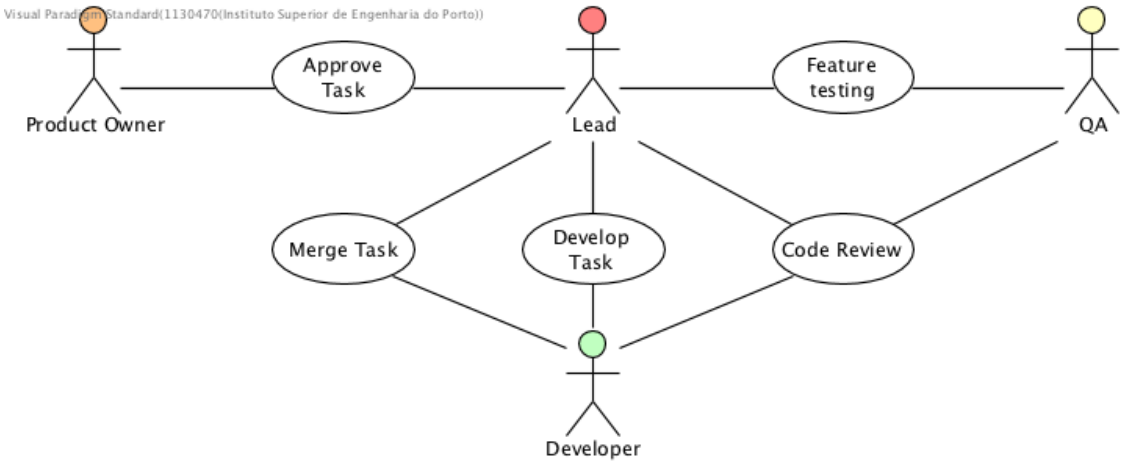

Figura 4.1: Papeis no processo de desenvolvimento de software atual

No diagrama 4.1 podemos então observar os papeis de *Developer*, *Lead*, *Product Owner* e *QA*. Cada papel está associado a etapas do processo de desenvolvimento, sendo elas *Todo*,

In progress, *Code Review*, *Feature Testing*, *PO Approval* e *Ready for Merge*. Na figura 4.2 podemos observar com mais detalhe o fluxo de desenvolvimento sobre estas etapas.

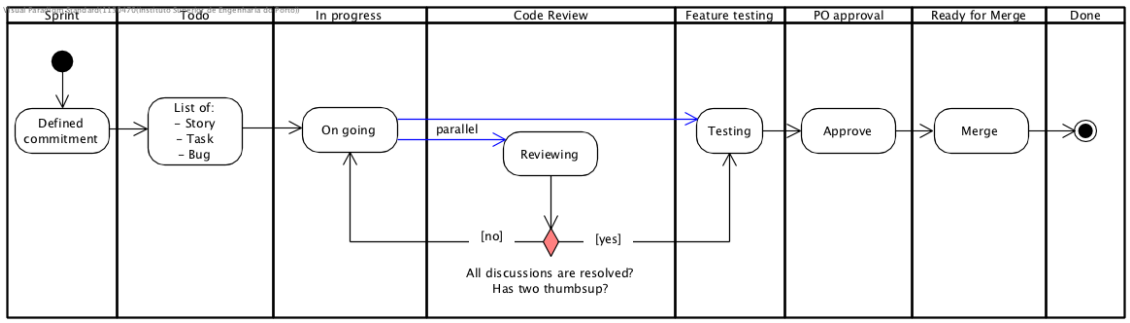

Figura 4.2: Processo de desenvolvimento de software atual

O processo de desenvolvimento de *software* atual refletido na figura 4.2, apresenta as diferentes etapas da *story*, *bug* ou *task*. Inicialmente é definido um *commitment*, ou seja, um pacote de *stories*, *bugs* ou *tasks* que serão o plano de execução da *sprint*. Após definidas as prioridades, os *developers* executam a tarefa e após a conclusão segue-se o *code review*, processo onde as pessoas da equipa analisam as alterações propostas e indicam melhorias e observações, e em paralelo, o *QA* entidade responsável por validar a tarefa, tem de aprovar, bem como o *Product Owner*. Após a tarefa ser analisada pelos *developers* e aprovada, a tarefa fica no estado de pronta para *merge*, operação que junta as novas alterações ao código atualmente existente.

Com um nível de granularidade mais fina, a figura 4.3 apresenta o processo de desenvolvimento atual para uma *story*, por parte dos *developers*, enquadrada na etapa de *In progress*.

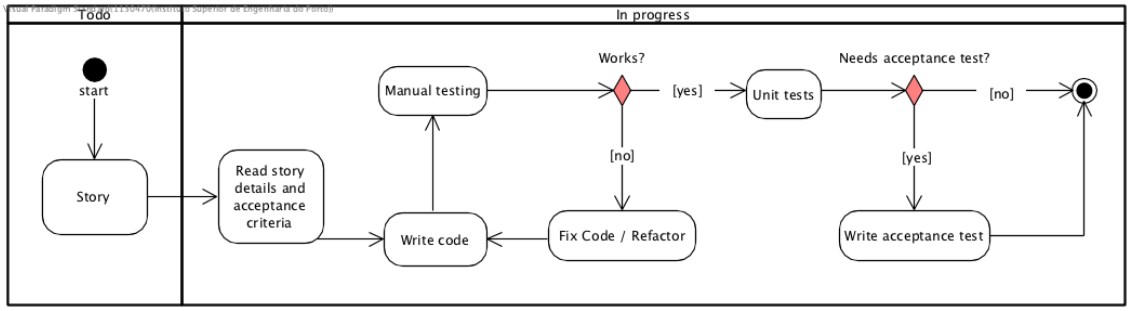

Figura 4.3: *Story* - Processo de desenvolvimento atual

Como podemos observar no diagrama 4.3, a *story* é iniciada e os requisitos são lidos e compreendidos por parte dos *developers*, após isso, o código é escrito e testado funcionalmente pelo *developer*, que após assumir que a funcionalidade corresponde aos requisitos, escreve os testes de unidade de forma a cobrir o código que foi desenvolvido, e por fim, são desenvolvidos os testes de aceitação, caso a *story* o indique.

Seguindo o mesmo fluxo, na figura 4.4 podemos observar o fluxo de correção de um *bug*, um erro que aconteceu de forma inesperada no sistema em produção, por parte dos *developers* enquadrada na etapa de *In progress*.

A correção de um *bug* de produção tipicamente é iniciada tentando replicar o fluxo que originou o erro localmente, ou seja, na máquina do *developer* em ambiente de desenvolvimento. No caso em que não é possível encontrar o erro, é feita uma réplica do estado atual de todo

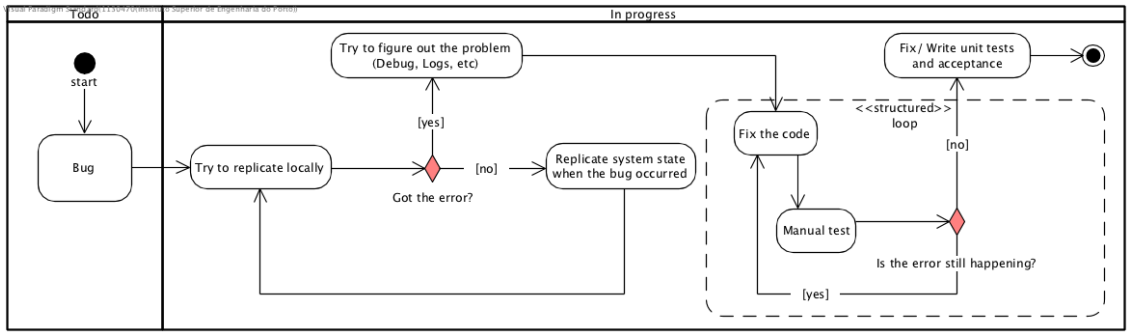

Figura 4.4: *Bug* (produção) - Processo de correção atual

o sistema envolvente que originou o erro. Após replicado o cenário, o código é corrigido com auxílio de ferramentas de *debug* ou *logging*. De seguida, o *developer* testa manualmente até verificar que o bug já não acontece, recorrendo a correções de código case seja necessário. Por fim, serão corrigidos os testes de unidade ou aceitação.

4.1.2 Relação entre entidades no processo de desenvolvimento

As entidades envolvidas no processo de desenvolvimento de *software* formam um conjunto de relações que permitem a concretização de iniciativas, projetos e entrega de valor para os clientes finais. O diagrama 4.5 ilustra a relação entre as entidades envolvidas no processo de desenvolvimento de *software*.

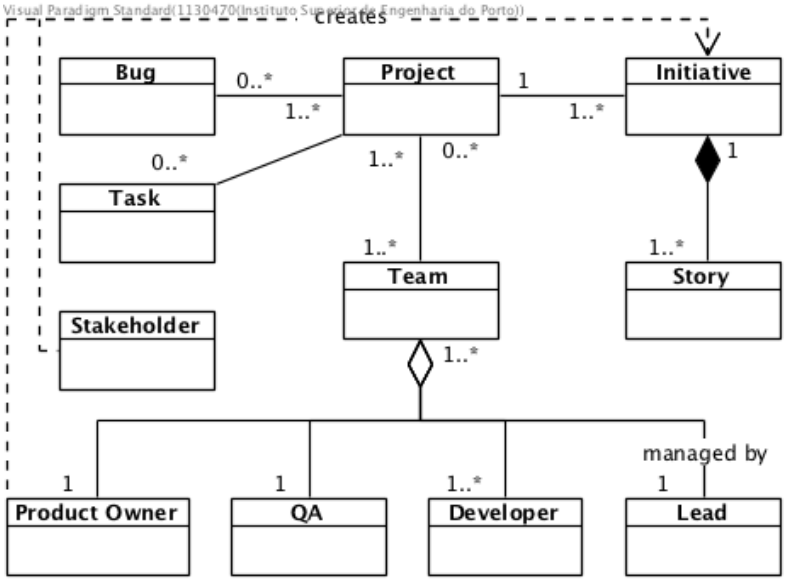

Figura 4.5: Relação entre entidades no processo de desenvolvimento

Todas entidades representadas em 4.5 interferem no processo de desenvolvimento, sendo que cada equipa é composta por um *Lead*, *QA*, *Product Owner* e *Developers* e a si tem associados projetos, projetos esses que contêm iniciativas, criadas pelos *stakeholders* e *product owner*, que originarão *stories*, entidades que após executadas permitem entregar valor ao cliente final.

O processo de definição da *story* pode ser observado no diagrama 4.6. Esse processo é designado por *refinement*. Antes deste evento ocorre o evento de *Story Mapping*, onde as *stories* são criadas com um nível de detalhe ainda muito curto.

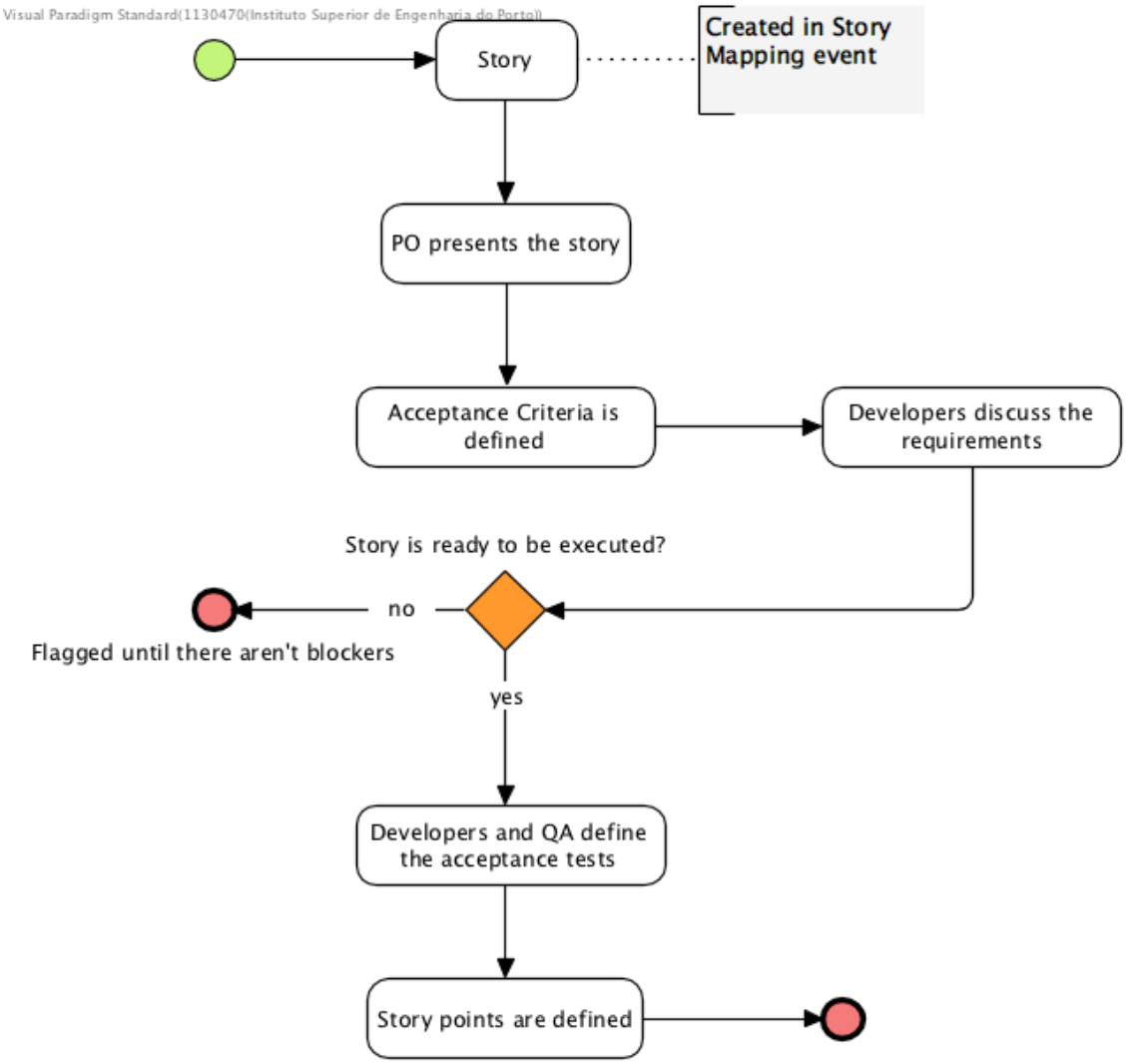

Figura 4.6: Processo de definição da *story*

Como podemos observar no diagrama 4.6, o processo de *refinement* é iniciado pela explicação da *Product Owner* e após ser definido o critério de aceitação, os *developers* discutem os requisitos e decidem se a *story* tem todas as condições para avançar. Nesse caso, existe a definição dos testes de aceitação caso seja necessário, finalizando o processo com a atribuição de *story points*, unidade de medida utilizada. Após este processo concluído, a *story* está pronta a ser executada. A definição dos testes de aceitação nem sempre seguem o rigor esperado nesta fase. Para além disso, este tipo de testes são escassos, ou seja, normalmente são definidos, em média, dois testes de aceitação para cada projeto.

4.1.3 Ferramentas e sistemas

As ferramentas e sistemas utilizados podem ser agrupadas por diferentes categorias: Gestão de equipa, Documentação, *Reporting*, Desenvolvimento de *software* e Implantação de

software. O diagrama 4.7 expõe apenas as ferramentas e sistemas que apoiam o desenvolvimento e implantação do *software*:

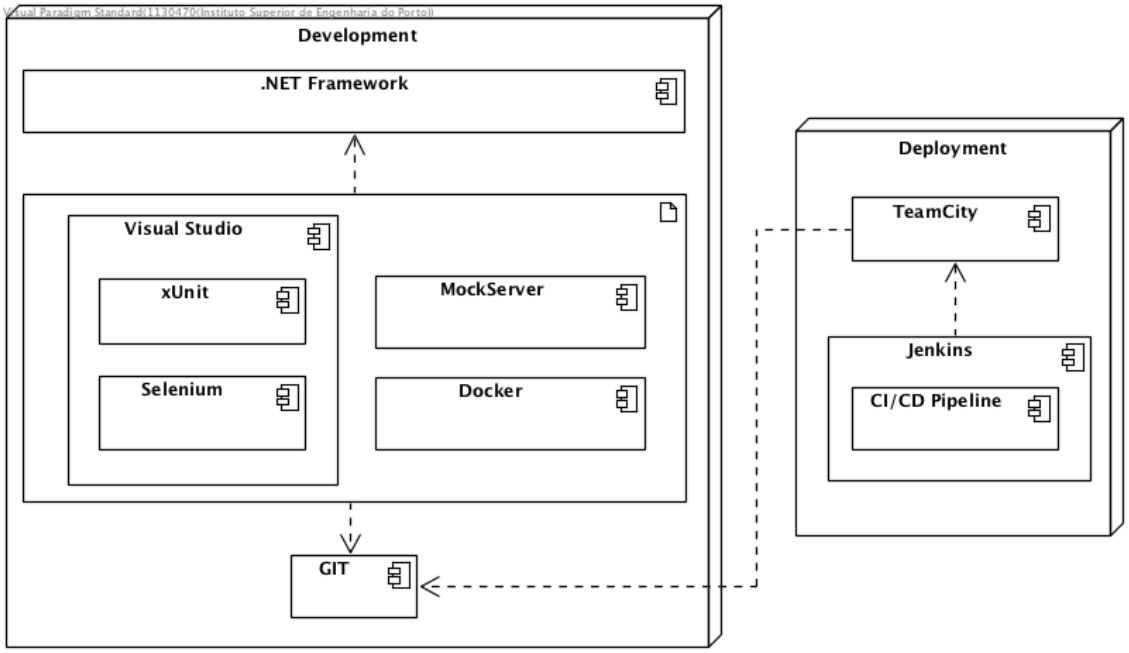

Figura 4.7: Ferramentas e sistemas utilizados atualmente

Podemos observar no diagrama 4.7 dois grandes componentes: Desenvolvimento e Implantação.

- Desenvolvimento: A *framework* transversal utilizada é *.NET*. As ferramentas associadas ao desenvolvimento de código são o *Visual Studio*, IDE utilizado para tarefas de programação. O *xUnit* é a ferramenta para testes de unidade. Quanto a testes de aceitação o *Selenium* e o *MockServer* são as ferramentas utilizadas. Por último, o *Docker* é a ferramenta de virtualização de ambientes ou componentes. O *GIT* será a entidade de controlo de versões que irá interagir com o componente de implantação.
- Implantação: O *Team City* receberá informação de novas alterações e é responsável por gerar uma nova *build* com as novas alterações, comunicando com o *Jenkins*, orquestrador que gere a *Pipeline*, permitindo percorrer todos os passos como execução de testes de unidade, integração, *smoke*, validações de segurança, entre outros, até à implantação final do *software* no destino.

A atual relação entre as ferramentas utilizadas e o processo de desenvolvimento pode ser observada na figura 4.8. O diagrama foi simplificado do diagrama original 4.3 de atividades (o estágio representado com "..."representa os mesmos passos do diagrama base). As ferramentas utilizadas estão ligadas aos estágios do processo de desenvolvimento. O *xUnit* é a ferramenta utilizada para o efeito de testes de unidade. No que diz respeito a testes de aceitação, o conjunto de ferramentas é composto por: *MockServer*, *Selenium* e *Docker*.

4.1.4 Análise do problema

Analisando o processo atual, podemos evidenciar o problema ao qual este estudo de caso pretende combater. Os momentos em que o problema se manifesta enquadram-se no processo

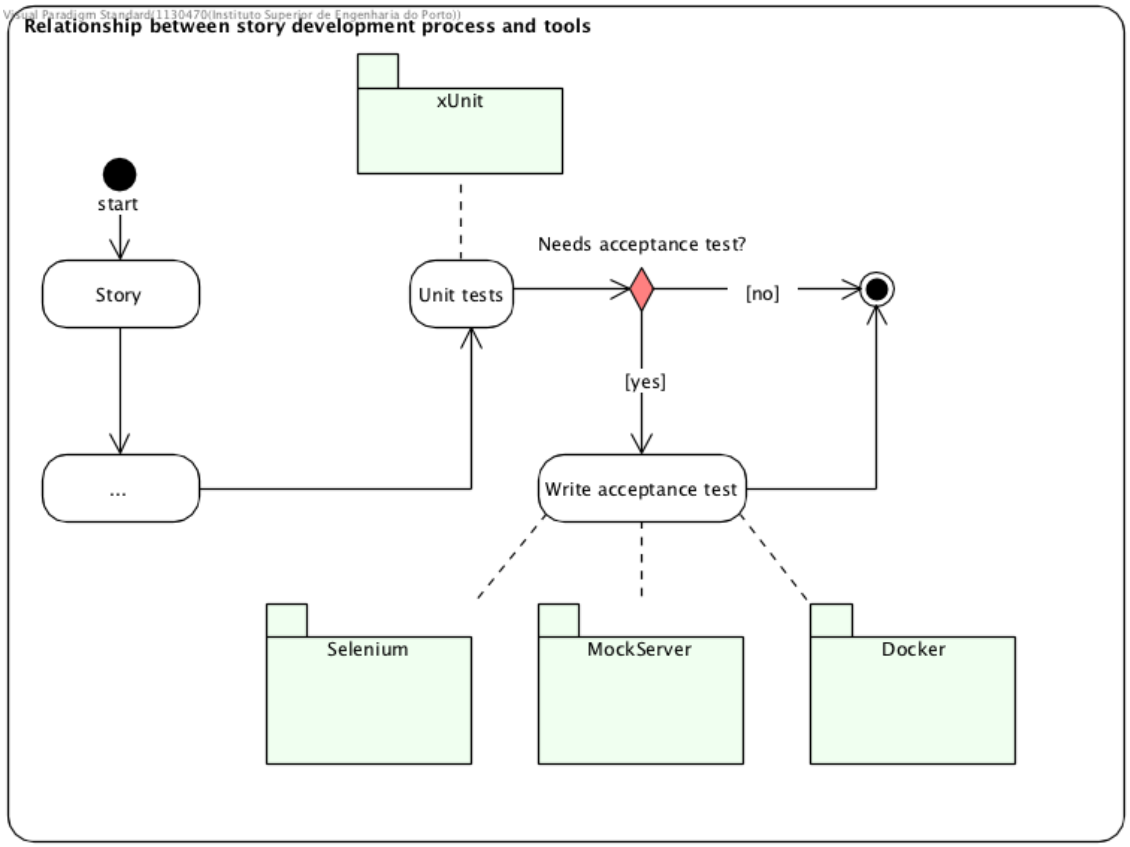

Figura 4.8: Relação entre o processo de desenvolvimento e as ferramentas utilizadas

de desenvolvimento de *software*, nomeadamente no passo de "In Progress"do processo de desenvolvimento representado na figura 4.2. Com mais detalhe podemos observar também as figuras 4.3 e 4.4. Não existe um processo que acompanhe os requisitos iniciais, que ajude no processo de desenho, que valide constantemente as novas alterações no *software* e que permita validar com rigor se o comportamento esperado foi atingido. Este conjunto de reflexões são chave para que possa ser desenhada uma solução para combater o problema. Na secção seguinte irá ser feita uma análise a possíveis soluções.

4.2 Estudo de soluções

Nesta secção serão discutidas as soluções alternativas que permitem resolver os problemas apresentados anteriormente na secção 1.2.

4.2.1 Introdução do TDD

O processo de desenvolvimento é chave para que sejam cumpridas metas e objetivos. Após a análise ao problema e ao estado atual do desenvolvimento representado em 4.3, o diagrama 4.9 apresenta uma alternativa que visa combater os problemas previamente detetados nesta dissertação.

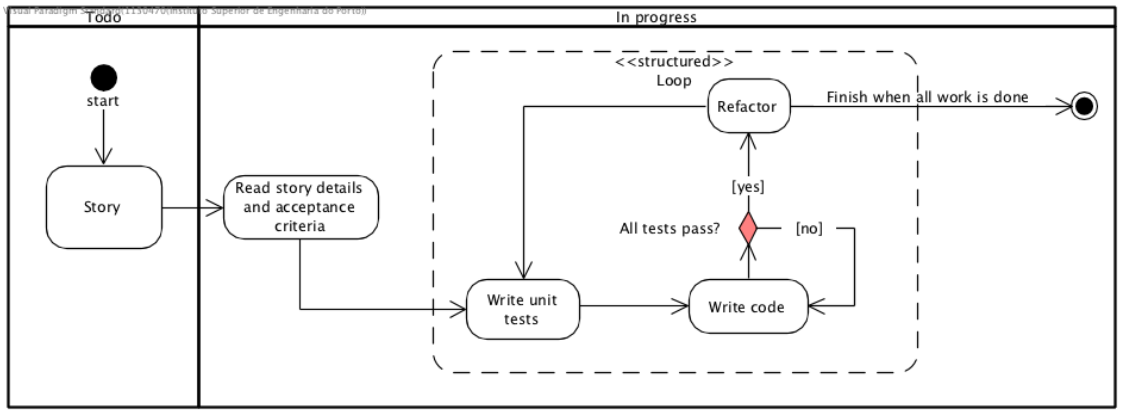

Figura 4.9: *Story* - Processo de desenvolvimento alternativo

O diagrama 4.9 introduz uma nova forma de abordar o desenvolvimento das *stories*, baseada na metodologia TDD, o *Red-Green-Refactor*. A sua premissa já foi descrita na secção 2.6.1. Com esta nova metodologia, o fluxo de desenvolvimento altera-se de forma a realizar os testes antes da implementação, invertendo o fluxo atual de desenvolvimento em que os testes são realizados à *posteriori*. Após já não haver mais *refactor* para realizar, o processo termina.

4.2.2 Introdução dos testes de integração

Atualmente os testes de integração não são praticados pela equipa. A proposta de introdução serve como propósito servir necessidades onde os testes de unidade não sejam suficientes.

Os testes de integração têm como objetivo testar a integração entre módulos. Esta necessidade surge quando existem *stories* mais técnicas. De acordo com a realidade da equipa, que passa maioritariamente pelo desenvolvimento de APIs, este tipo de *stories* é mais frequente.

Perante este cenário, duas propostas são apresentadas:

- Definição do processo de *refinement*: Neste processo, captar a necessidade de escrita de testes de integração, ou seja, adicionar um novo estágio ao diagrama 4.6.
- Definição durante o processo de desenvolvimento: Quando o responsável pela realização do desenvolvimento sente necessidade de complementar a validação do código com um teste de integração.

4.2.3 Introdução do BDD

Após analisar os benefícios do BDD, presente na secção 2.6.3, esta secção tem como objetivo apresentar desenhos para incorporar esta prática.

Na figura 4.10 está representada uma alternativa ao processo de definição da *Story*. Este diagrama foi adaptado do diagrama original 4.6. Aqui surge um novo estágio, definição de testes de aceitação com uso do BDD. Os testes de aceitação são testes mais completos no que diz respeito ao critério de aceitação, rigor da definição e também por conseguir ligar várias áreas, nomeadamente desenvolvimento, negócio e clientes com a mesma linguagem.

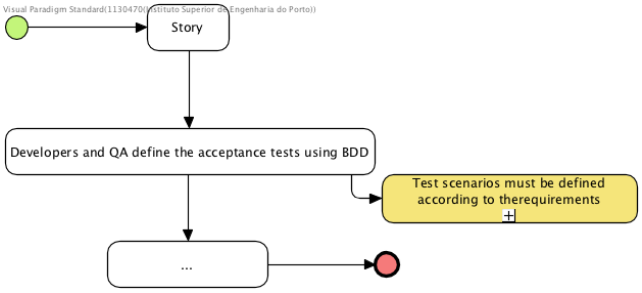

Figura 4.10: *Story* - Processo de definição alternativo

A figura 4.11 vem assim complementar a ideia presente no diagrama 4.10. No que diz respeito à aplicação do BDD durante o processo de desenvolvimento, se a *story* referir a necessidade de escrita de teste de aceitação, então esse teste será escrito antes do processo de desenvolvimento (escrita de código de produção) começar.

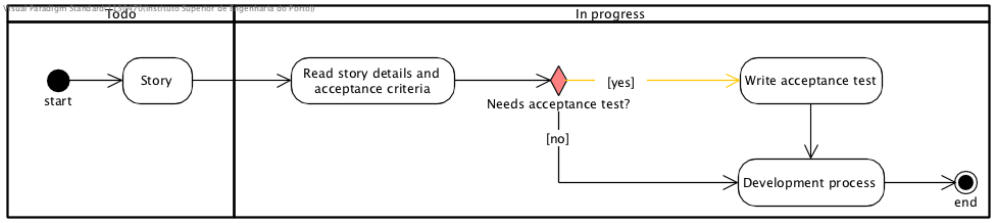

Figura 4.11: Introdução do BDD no processo de desenvolvimento

Cucumber

Esta solução propõe a introdução da ferramenta *Cucumber* 2.2.5. Como já referido, o *Cucumber* é uma ferramenta de testes de aceitação direcionado ao BDD. Na figura 4.12 podemos observar a alternativa ao conjunto de ferramentas de teste. A introdução do *Cucumber* e do *Gherkin*, sintaxe utilizada para descrever os testes, vai ao encontro da possibilidade de introduzir o BDD como uma prática complementar ao desenvolvimento de *stories*.

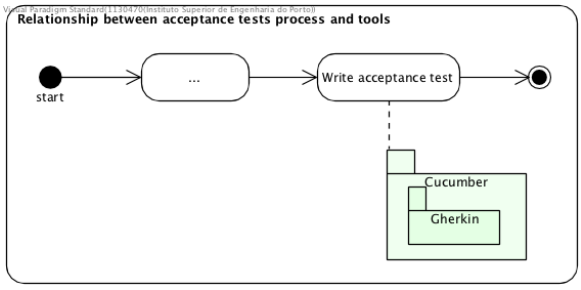

Figura 4.12: Introdução do *Cucumber*

SpecFlow

Seguindo a mesma linha da análise anterior, esta proposta de solução aborda o mesmo conceito presente em 4.2.3, a utilização do BDD mas com o auxílio da ferramenta *SpecFlow* e do *Gherkin*. Estas ferramentas podem ser executadas facilmente dentro do ecossistema do *Visual Studio*.

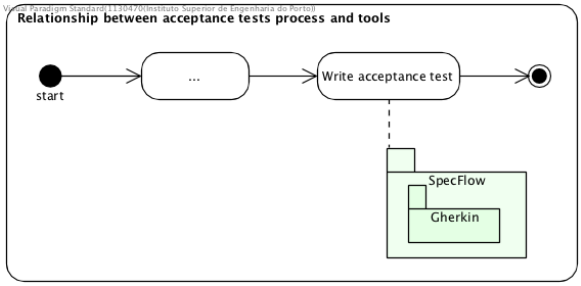

Figura 4.13: Introdução do *SpecFlow*

Selenium

Como ferramenta de automação de testes de *Web*, o *Selenium* apresenta-se como uma possibilidade para os testes de aceitação. Pode ser utilizado em conjunto com o *Cucumber* ou com o *SpecFlow*. O diagrama 4.14 representa a introdução do *Selenium* como uma alternativa ao conjunto de ferramentas de teste. A utilização de *Selenium* tem como dependência um navegador *Web* para executar os testes.

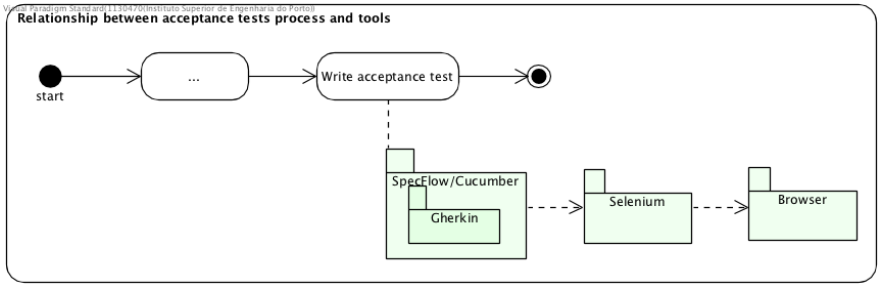

Figura 4.14: Introdução do *Selenium*

4.2.4 Introdução aos Mutation Tests

Como referido em 2.3.2, a aplicação de testes de mutação é uma mais valia a avaliar a qualidade dos testes praticados. A figura 2.3.2 apresenta a introdução deste conceito. Após o processo de desenvolvimento estar concluído, são criados e configurados os testes de mutação, utilizando a técnica referida em 2.3.2. Se existirem mutantes vivos, os testes serão corrigidos, ou até mesmo o código fonte, até não existirem mutantes vivos. Após todos os mutantes estarem "mortos", o fluxo de desenvolvimento segue naturalmente.

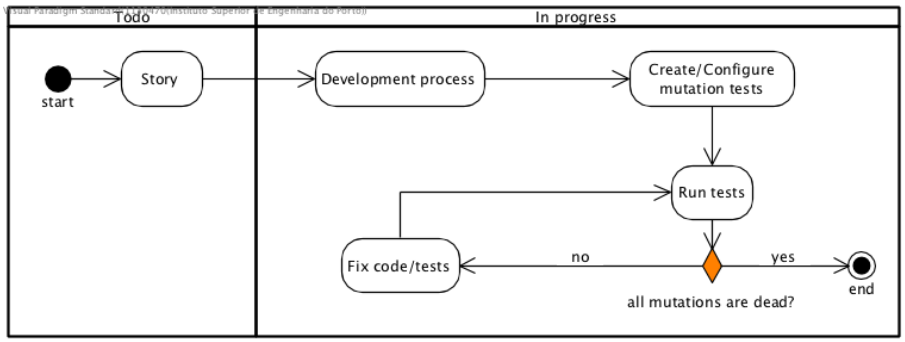

Figura 4.15: Introdução dos testes de mutação

4.2.5 Ferramentas de simulação aplicacional

AspNetCore Mvc Testing

AspNetCore Mvc Testing 2.2.10 tem a capacidade de criar o ambiente para testes de integração e aceitação de forma isolada (Microsoft 2019c), permitindo preparar todo o *setup* inicial de acordo com o teste em questão (Microsoft 2019b). Na figura 4.16 podemos observar a interação entre componentes, onde o componente *Tests* requer uma nova instância ao componente *AspNetCore Mvc Testing*, que por sua vez cria e devolve uma nova instância da aplicação alvo *Application*.

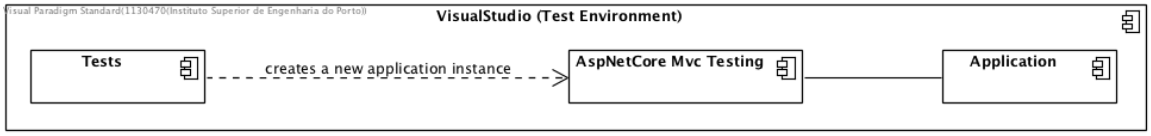

Figura 4.16: Simulador aplicacional - *AspNetCore Mvc Testing*

4.2.6 Ferramentas de simulação de APIs

Testes de integração e de aceitação, sobre a realidade da equipa de desenvolvimento, dependem quase cem por cento das vezes de serviços externos. Uma das forma de combater os problemas que as dependências trazem (indisponibilidade, erros, etc) é a utilização de simuladores de APIs. Esta alternava propõe a utilização de uma das seguintes opções: *MockServer* ou WireMock.

MockServer

O *MockServer* é uma ferramenta que corre na máquina onde os testes são executados. Depende da *JVM* para executar. Está fora do ecossistema do *Visual Studio*, ou seja, funciona como um processo externo na máquina em questão.

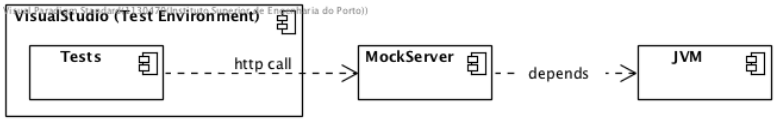

Figura 4.17: Simulador de API - *MockServer*

WireMock

O *WireMock* 2.2.8 apresenta-se como uma ferramenta que está dentro do ecossistema da solução, que por sua vez dentro do *Visual Studio*. Depende do *.NET Framework*.

Na figura 4.18 podemos observar a interação entre componentes.

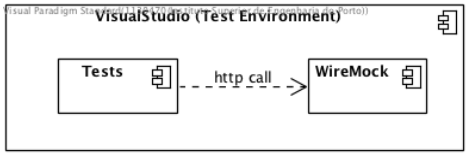

Figura 4.18: Simulador de API - *WireMock*

4.2.7 Ferramentas de simulação de base de dados

Mongo2Go

O *Mongo2Go* 2.2.9 é uma ferramenta que oferece uma simulação da base de dados *MongoDB*. A vantagem da utilização deste componente é o facto de ficar encapsulado à solução, e por sua vez ao *Visual Studio*, ou seja, é uma dependência que executa sobre um processo interno e não externo à execução da solução. Oferece rapidez e facilidade de desenvolvimento, dando a possibilidade de criar várias instâncias de base de dados em simultâneo.

Na figura 4.19 podemos observar a interação entre os principais componentes, sendo eles o componente de *Tests*, que pode representar um teste de unidade, integração ou aceitação, que por sua vez comunica com a *Application Instance*, instância da aplicação alvo a ser testada que naturalmente comunica com o componente de base de dados, neste caso o *Mongo2Go*.

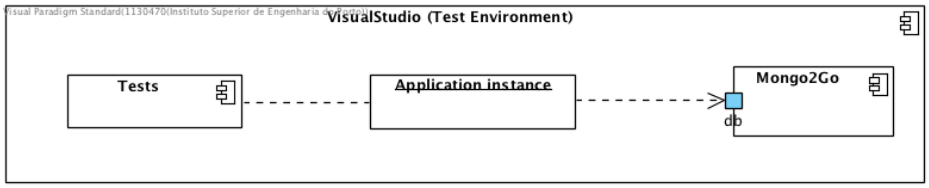

Figura 4.19: Simulador de base de dados - *Mongo2Go*

Docker

O *Docker* 2.4.3 apesar de apresentar muitas capacidades, neste contexto apresenta-se como uma alternativa à virtualização da base de dados, permitindo assim criar uma imagem da

base de dados alvo, manipular e apagar a mesma. Como pressuposto, este processo teria de executar lado a lado com a bateria de testes, e estaria fora do ecossistema da solução.

Na figura 4.20 podemos observar a interação entre componentes, sendo o ponto de destaque a comunicação entre a *Application Instance* e o *Docker*, que está fora do ecossistema do *Visual Studio*. O *Docker* utiliza assim um *DockerImage* que contém uma instância do *MongoDB*.

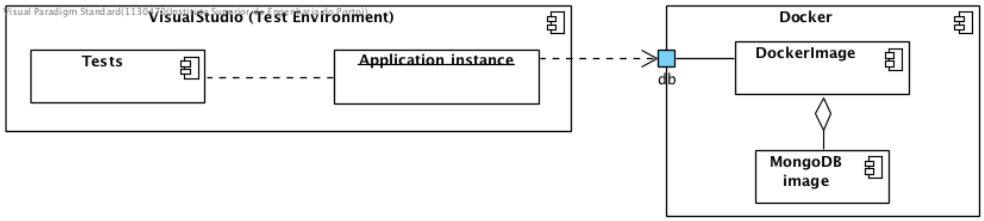

Figura 4.20: Simulador de base de dados - *Docker*

4.3 Escolha da solução

Nesta secção irá ser descrita a escolha da solução que irá ser utilizada neste estudo de caso. Para fundamentar a escolha, esta será comparada com as soluções apresentadas previamente na secção 4.2. Diagramas de UML serão apresentados de modo a fundamentar com base no contexto atual o que irá ser alterado para implementar a solução escolhida.

Após uma análise ao problema e às soluções desenhadas, a solução a adotar passa pela introdução do *Test-First*, TDD e do BDD. Após uma análise ao problema 1.2 e aos objetivos desta dissertação 1.3 vários artigos foram analisados com o intuito de perceber qual o caminho a seguir, e a escolha do *Test-First*, TDD e BDD pode ser fundamentada por algumas conclusões retiradas de casos de estudo previamente realizados.

Segundo (Nagappan et al. 2008) e (Janzen e Saiedian 2006), *Test-First* não é apenas uma técnica de teste. O TDD é considerado um processo de *design* e, em alguns casos, mais que um processo de teste. A granularidade fina do ciclo de teste e código fornece *feedback* contínuo, sendo que as falhas ou defeitos são identificados muito cedo. As *Breaking change* são detetadas cedo. (Janzen e Saiedian 2008) defende que o TDD produz código mais simples, mais coeso e menos acoplado do que o código desenvolvido sobre o processo tradicional, *Test-Last*. (Jeffries e Melnik 2007) refere ainda que, o TDD é uma disciplina de *design*, não uma atividade de testes. Qualquer pessoa que tenha trabalhado em *software legacy* reconhece a situação em que um sistema continua a funcionar, mas cada vez mais se torna desatualizado até que se transforma num castelo de cartas. Ninguém quer tocar, porque mesmo uma pequena alteração no código provavelmente levará a um efeito colateral indesejado. O TDD não é *silver bullet*, no entanto pode ajudar a formar *developers* mais disciplinados e efetivos. Segundo (Solis e Wang 2011) e (*Introducing BDD* 2006), O BDD é uma evolução do TDD e ATDD, sendo benéfico a utilização do TDD e BDD em conjunto para que os requisitos sejam cumpridos e interpretados da melhor forma.

Após esta análise, e de acordo com as ambições da organização e da equipa de trabalho, o caminho a seguir fica traçado. Os artefactos seguintes exemplificam como será a arquitectura da solução e das ferramentas a utilizar.

Processo de desenvolvimento No diagrama 4.21 está representado o processo de desenvolvimento sobre uma *Story*.

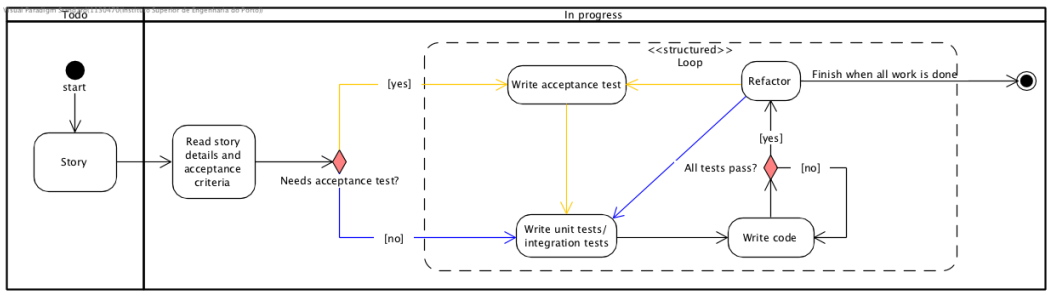

Figura 4.21: *Story* - Processo de desenvolvimento proposto

Após uma análise à alternativa 4.2.1, a solução aí proposta apesar de válida, é incompleta para este estudo de caso. Segundo alguns autores como (*Introducing BDD* 2006) e (Solis e Wang 2011), o TDD deve ser complementado com o BDD. O que o diagrama propõe é complementar a alternativa proposta com a introdução do BDD 4.11, ou seja, a escrita do(s) teste(s) de aceitação antes de qualquer escrita de código, no caso de a *story* refira essa necessidade. Seguindo a mesma abordagem do *Red-Green-Refactor*, o fluxo de desenvolvimento seguirá dessa forma. As vantagens desta abordagem já foram referidas na secção 2.6.3 e 2.6.1.

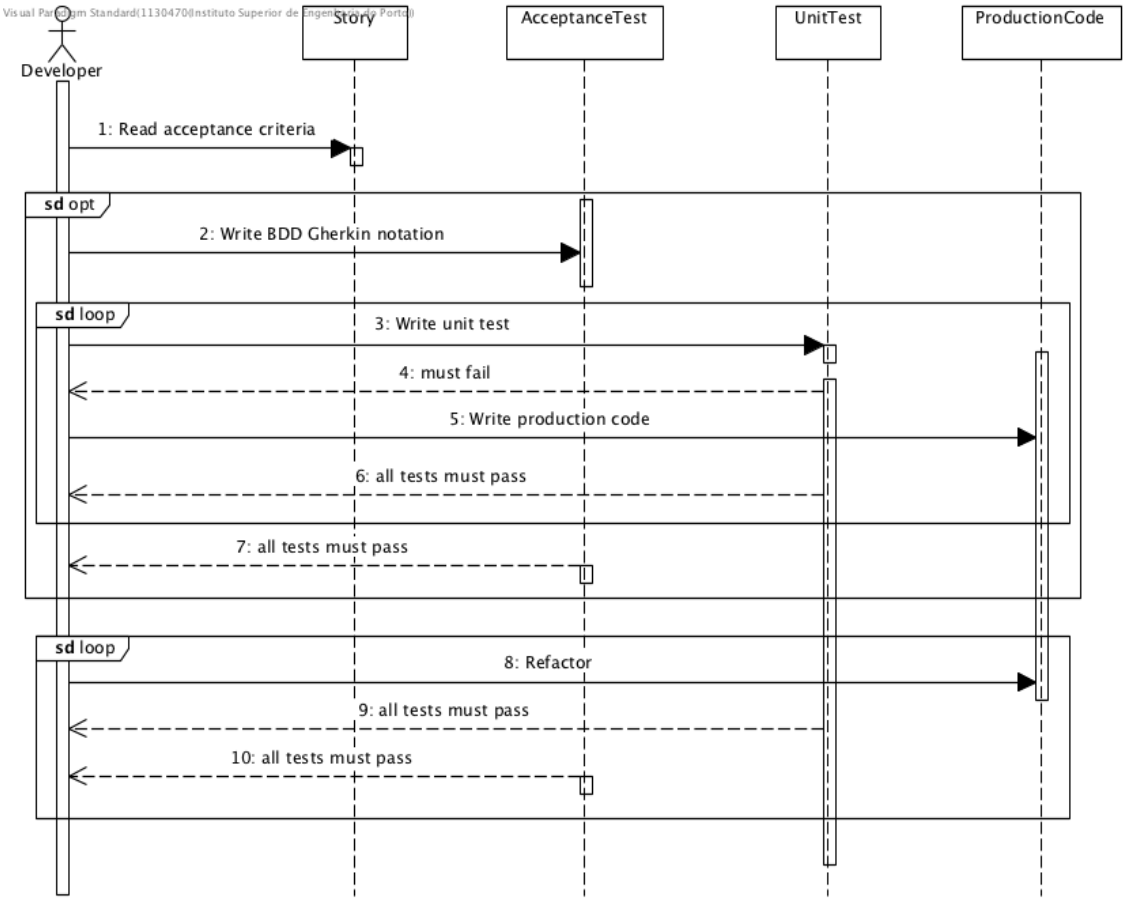

Figura 4.22: *Story* - Diagrama de sequência proposto

O diagrama de sequência 4.22 ajuda a complementar a ideia fundamentada em 4.21. O *developer* inicia o processo de desenvolvimento com um teste de aceitação, caso a *story* o indique, e de seguida, entra dentro do ciclo do TDD até que todos os testes passem e os requisitos sejam satisfeitos.

O grande propósito é então combater os problemas descritos, escrevendo em primeiro lugar os testes, compreender os requisitos e escrever o código seguindo a linha dos requisitos. A cobertura do código será muito importante para o processo de *refactor*. Em suma, para além da aplicação do TDD, que irá abordar os módulos mais curtos de unidade, o BDD será também aplicado de forma a cobrir os requisitos funcionais.

Em paralelo, os testes de integração devem também ser contemplados, para casos onde o *developer* ache necessário complementar o seu desenvolvimento, ou, caso a *story* o indique, tal como referido em 4.2.2.

Por fim, tal como referido no diagrama 4.15, é esperado implementar esta abordagem para os testes de mutação. Dentro deste contexto, os testes de mutação serão aplicados após o processo de desenvolvimento 4.21 estiver concluído. Uma das abordagens no auxílio de raciocínio será o envolvimento, para além dos *developers*, do *quality assurance*.

Processo de correção (bug) O diagrama 4.23 apresenta a solução para o processo de correção de um *bug* sobre código de produção.

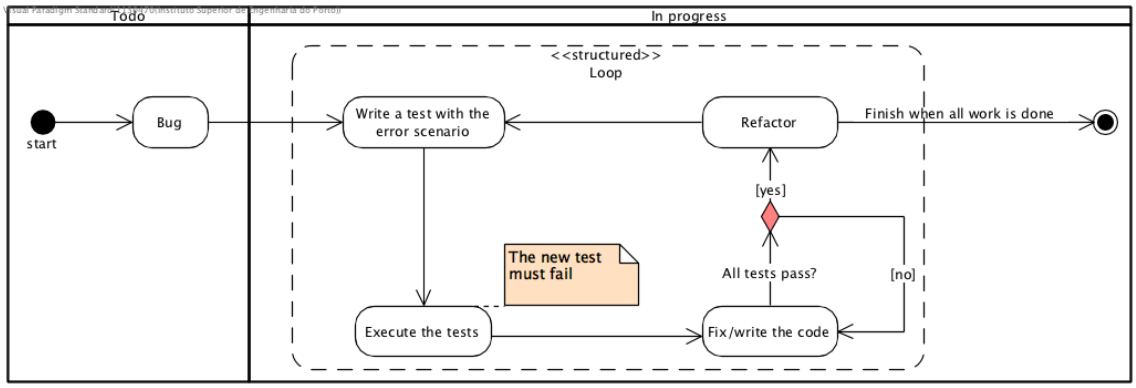

Figura 4.23: *Bug* (produção) - Processo de correção proposto

Segundo os princípios do TDD e BDD, o processo de correção passa por escrever o teste que reporta esse *bug*, e de seguida é feita a correção do código até o(s) teste(s) passar(em). Após a barra verde aparecer, o processo de *refactor* entra em cena, seguindo a mesma lógica referida anteriormente *Red-Green-Refactor*.

O diagrama de sequência 4.24 fundamenta e exemplifica como um *developer* procede na correção de um *bug*, de acordo com a proposta presente em 4.23. O componente *Test* pode representar um teste de unidade, integração ou aceitação. Com este método, garantimos que o mesmo erro nunca mais acontece, aumentando a confiança no código e a sua cobertura.

Componentes O diagrama 4.25 apresenta os componentes que farão parte de todo o ecossistema em torno do projeto que irá ser desenvolvido.

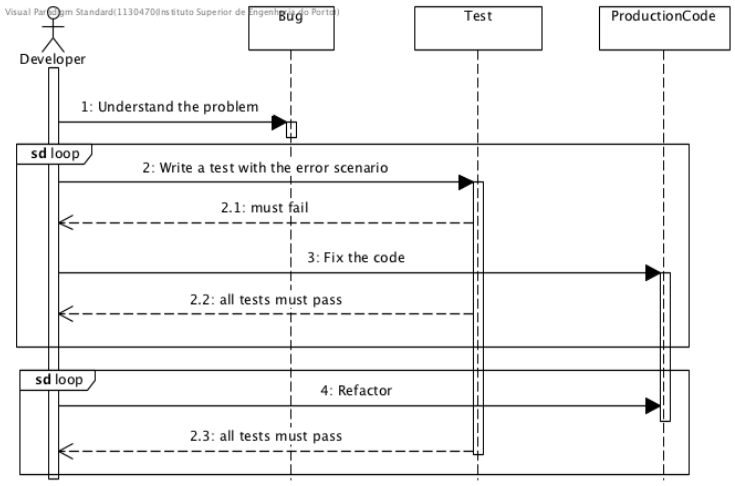

Figura 4.24: *Bug* (produção) - Diagrama de sequência proposto

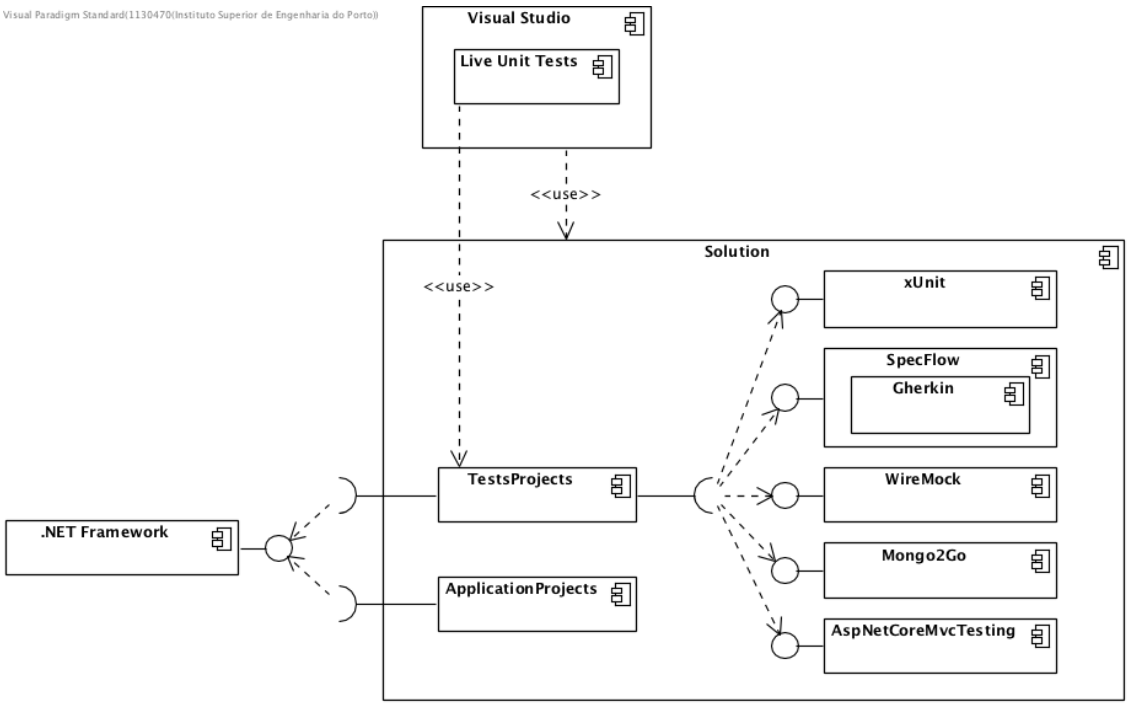

Figura 4.25: Diagrama de componentes proposto

Para além dos componentes presentes em 4.25, o diagrama apresenta também a interacção entre eles. Podemos assim evidenciar dois dois grandes componentes, *Visual Studio* e *Solution*. O *Visual Studio* integra o *Plug-In LiveUnitTestes* 2.2.3, ferramenta que dá resposta em tempo real do resultados dos testes e da cobertura dos mesmos sobre os código fonte. A *Solution* apresenta os *ApplicationProjects* que naturalmente dependem de *.NET Framework*, ferramenta utilizada pela empresa e *TestsProjects*, que estão ligados a uma série de componentes. Após uma análise às propostas apresentadas na secção 4.2, este diagrama reflete a escolha de cada componente.

A escolha *xUnit* é óbvia, pois é a ferramenta já utilizada pela equipa e aquela que oferece todas as condições para a realização de todo o tipo de testes de forma fiável.

AspNetCoreMvcTesting referida em 4.2.5 apresenta-se como a única alternativa fiável e aquela que é recomendada pela *Microsoft* (Microsoft 2019c).

A escolha do *SpecFlow* sobre o *Cucumber* referido na proposta 4.2.3 é justificada pelas seguintes razões:

- O *SpecFlow* permite a que equipas que trabalhem com .NET utilizem o BDD de forma fácil.
- A introdução do *Cucumber* iria provocar uma curva de aprendizagem maior, requerendo mais tempo fazendo com que o estudo de caso não se concretize no tempo esperado.
- O *SpecFlow* permite a que a equipa de desenvolvimento trabalhe dentro do ecossistema do *Visual Studio*, sem necessitar de dependências externas.
- Como o ambiente de desenvolvimento irá continuar a ser .NET, não existe vantagem no uso do *Cucumber*, sendo o *SpecFlow* mais orientado para este ambiente.

A ausência do *Selenium* pode ser justificada por: Apesar de a equipa de desenvolvimento já estar bastante familiarizada com *Selenium*, os projetos que irão ser aplicados neste estudo de caso são à base de Web APIs, aplicações muito direccionadas para camadas de serviços. Existe interface gráfica, onde está presente a documentação da API e onde é possível interagir com as suas rotas (*endpoits*). Apesar de ser possível a utilização de *Selenium*, não se justificaria o custo/benefício da sua adoção.

O *WireMock* 4.2.6 será a ferramenta de simulação de APIs a utilizar. Atualmente, o *MockServer* é a ferramenta que a equipa de trabalho utiliza para cenários onde é necessário simular APIs. A opção pelo *WireMock* é justificada:

- O *WireMock* permite a que a equipa de trabalho consiga utilizar um simulador de APIs dentro do ecossistema da solução e do *Visual Studio*.
- O *MockServer* requer uma instalação/ativação na máquina em que os testes estão a executar.
- O *MockServer* é um processo externo que corre na máquina em que os testes estão a executar.
- O *WireMock* permite uma maior performance no que diz respeito ao processo contínuo de execução dos testes.
- O *WireMock* permite a criação de inúmeras instâncias em memória, cada uma com a sua porta, permitindo que inúmeros testes corram em simultâneo e utilizem uma porta diferente para os pedidos HTTP/HTTPS.

Em suma, o uso do *WireMock* vai ao encontro do objetivo da equipa de trabalho e deste estudo de caso, obter um ambiente em que os testes executem o mais rápido possível, de preferência, dentro do ecossistema do *Visual Studio*.

O *Mongo2Go* irá ser a ferramenta de simulação de base de dados 4.2.7, ao invés do *Docker* 4.2.7 e esta escolha pode ser justificada por:

- *Mongo2Go* oferece rapidez na criação de uma imagem de base de dados comparando com o *Docker*.
- *Mongo2Go* é um componente que pode ser usado de dentro do ecossistema da solução.

- O *Docker* teria de executar sobre um processo paralelo, na máquina onde os testes executam.
- *Mongo2Go* permite a criação de inúmeras instâncias em memória de base de dados, permitindo que inúmeros testes corram em simultâneo e utilizem uma instância diferente de base de dados.

Capítulo 5

Implementação

Neste capítulo serão abordados os conceitos utilizados para implementar, de acordo com a solução desenhada no capítulo 4, a solução final.

5.1 Enquadramento

A seguinte lista representa os principais tópicos que serão abordados para expor todo o trabalho realizado durante a execução desta dissertação.

- Contexto do projeto: Ambiente e enquadramento em que este estudo de caso se realizou.
- Instanciação do projeto: Estágios que fizeram parte deste estudo de caso e exemplificação de como foram executados.
- Treino sobre a equipa de desenvolvimento: Treino realizado com o intuito de formar e introduzir os conceitos chave para a realização da proposta desenhada na secção 4.3.
- *Test-Driven Development*: Apresentação das metodologias introduzidas, métodos e técnicas de auxilio a esta prática.
- *Behavior-Driven Development*: Introdução, enquadramento e aplicação desta metodologia.
- Solução de testes: Desenho e implementação da bateria de testes montada sobre a solução.
- Integração com CI/CD *Pipeline*: Integração da bateria de testes desenvolvida com a *Pipeline* de CI/CD.

5.2 Contexto do projeto

O projeto em que se realizou este estudo de caso foi um projeto iniciado do zero. O projeto passou pelo processo de *setup* (configuração) inicial após a sua criação, adotando as normas arquitecturais recomendadas, que passam por preparar a infraestrutura da aplicação para seguir a estrutura comum existente e separar em diferentes camadas os componentes que compõe a aplicação. Após o *setup* estar concluído, novos desenvolvimentos começaram a ser desenvolvidos.

O projeto em questão é um serviço, mais precisamente uma *Rest API*, desenvolvida sobre a tecnologia *ASP.NET Core 2*, na linguagem C#, não contendo nenhuma interface gráfica. "ASP.NET Core suporta a criação de serviços *RESTful*, também conhecidos como Web APIs, usando C#. Para lidar com solicitações, uma web API usa controladores"(Microsoft 2019a).

A figura 5.1 apresenta o diagrama de alto nível da arquitectura dos componentes. O componente *clients* representam os consumidores da *Rest API*, que é composta pelos seguintes componentes:

- *HTTP Pipeline*: Responsável por gerir os pedidos HTTP.
- *WEB API Controllers*: Componente que expõe os contratos e *endpoints* (ponto de entrada) que a aplicação aceita.
- *Service Layer*: Camada de serviços da aplicação.
- *Data Layer*: Camada de gestão interna.
- *Repository Layer*: Camada de persistência.

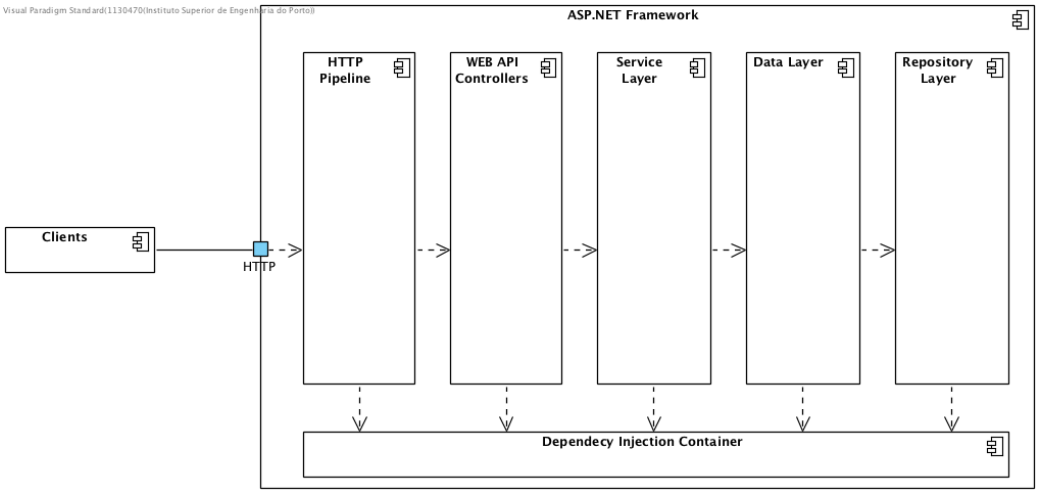

Figura 5.1: Arquitetura de componentes

O IDE utilizado foi o *Visual Studio*. O projeto estava ligado ao sistema de controlo de versões e a uma *pipeline* de CI/CD. A *pipeline* era responsável por criar uma imagem da aplicação, executar toda a bateria de testes (de unidade, integração e aceitação), executar passos internos definidos pela organização e por último, implantar nos servidores de produção.

A solução desenhada foi apresentada à equipa de trabalho em várias sessões, onde foi possível discutir e analisar a solução proposta e receber *feedback* adicional dos diferentes membros com sugestões, de forma a criar um ambiente contributivo e construtivo em prol de melhorar os processos atuais. Como a nova solução desenhada obriga a uma mudança de paradigma de programação e trabalho, foi dada a abertura a todos os membros da equipa de utilizarem ou não a nova abordagem, criando assim um ambiente com menos atrito e rigidez. A estratégia passou por incentivar os membros a utilizarem a solução desenhada, recorrendo a estudos científicos realizados a nível industrial com demonstração de resultados e também, em forma de alerta, expor os problemas identificados que persistem no processo de desenvolvimento anterior à nova proposta.

5.3 Instanciação do processo

A instanciação do processo passou por, numa primeira fase, pesquisar informação relevante sobre as técnicas que iriam ser introduzidas à equipa. Após esse processo, essa informação foi adaptada e enquadrada à realidade da empresa e da equipa para que de uma forma paralela, dois processos ocorressem: o treino sobre a equipa e o desenvolvimento do projeto. O diagrama de atividades 5.2 exemplifica a instanciação do processo.

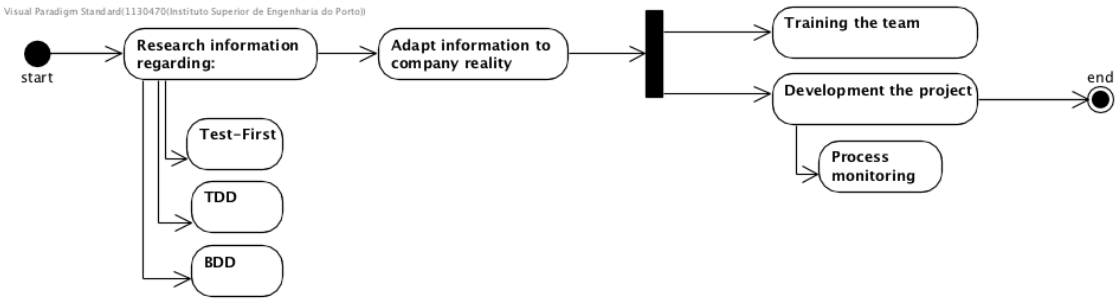

Figura 5.2: Instanciação do processo

5.4 Treino sobre a equipa de desenvolvimento

Tal como referido na secção 5.2, a proposta desta dissertação passa por alterar o processo de desenvolvimento atual, e para isso foram agendadas reuniões recorrentes de forma a introduzir os principais temas e treinar a equipa para que adquira todo o conhecimento necessário para executar a proposta detalhada na secção 4.3.

O treino exercido sobre a equipa abordou os seguintes tópicos:

- Apresentação do problema a que esta dissertação se compromete a resolver.
- Apresentação da possível solução para o problema existente.
- Introdução ao *Test-First*, TDD e BDD, práticas chave para executar a solução desenhada.
- Resultados esperados.

A técnica utilizada para realizar as formações e treino foi o *Coding dojo*, um encontro em que vários *developers* se reúnem para trabalhar sobre um desafio de programação. O objetivo é aprender, ensinar e melhorar com outros *developers* sobre um ambiente não competitivo, num curto intervalo de tempo (1 a 1,5 horas) (Wright 2009).

Todas as práticas referidas foram acompanhadas com sessões de treino específico, com utilização de exemplos reais que a equipa lida habitualmente no seu dia a dia de trabalho. Para complementar estas sessões, existiram e existem ocasionalmente sessões de esclarecimento e acompanhamento quando surgem dúvidas ou questões sobre qualquer tema introduzido. Segundo (Shull et al. 2010), é recomendável abordar o TDD associando um *developer* inexperiente com alguém mais experiente. Esta prática irá ajudar no processo de interiorização do TDD, enquadrado no processo de desenvolvimento de *software*.

O diagrama temporal 5.3 mostra com mais detalhe o momento no tempo onde os temas foram introduzidos à equipa.

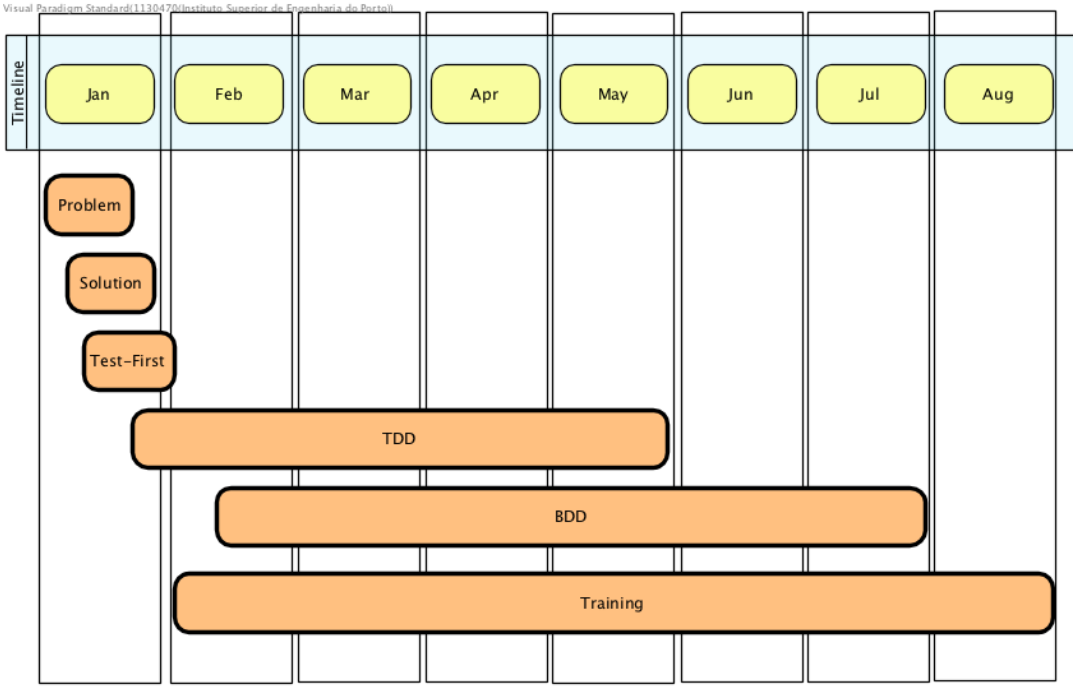

Figura 5.3: Diagrama temporal - Introdução de conceitos à equipa

O diagrama 5.3 mostra que no primeiro mês (janeiro) foi realizada a introdução do problema e a proposta de resolução, sendo que seguidamente foram introduzidos de forma contínua as técnicas TDD e BDD em simultâneo com sessões de treino.

5.5 Test-Driven Development

A introdução e implementação do TDD sobre a equipa de desenvolvimento foi realizada de forma gradual ao longo do tempo, com o objetivo de iterativamente consolidar os principais conceitos desta prática. O TDD era um técnica conhecida pela maioria dos elementos da equipa, mas apenas dos conceitos mais simples, como o ciclo de *Red, Green, Refactor* e *Test-First*. Não existia um conhecimento aprofundado sobre estratégias de implementação, técnicas de *refactor*, padrões e conhecimento das regras chave para obter uma bateria de testes que consiga dar resposta ao TDD. A estes fatores, juntou-se o facto de que apesar do conhecimento superficial já existente, o TDD nunca foi utilizado de forma regular pelos membros da equipa e perante este cenário, várias apresentações foram realizadas de forma a introduzir os seguintes temas suavemente:

- Introdução ao TDD: Como surgiu e em que âmbito, quais as motivações, os conceitos base como o *Test-First*, o ciclo iterativo *Red, Green, Refactor* 2.6.1 e o resultado de estudos de caso realizados industrialmente sobre a adoção do TDD. Tudo isto serviu para expor esta técnica de forma a sensibilizar e motivar os membros da equipa a adotarem para combater os problemas já conhecidos.

- Caraterísticas dos testes: Para que o TDD seja aplicado e utilizado na sua íntegra de forma regular, a bateria dos testes da solução deve obedecer a um conjunto de características, de forma a que seja possível executar todos os testes o mais rápido possível para que o *feedback* seja imediato mediante novos desenvolvimentos de código. Para isto ser possível, várias abordagens foram estudadas e aplicadas de forma a garantir rapidez e eficiência no que diz respeito aos testes, tal como referido na tabela 2.1. Na secção 5.7.2, irão ser apresentadas as caraterísticas dos testes com exemplos.
- Padrões (testes): Os padrões introduzidos sobre TDD foram baseados no livro publicado por (Beck 2003). O objetivo destes padrões é facilitar e viabilizar a implementação dos testes utilizando técnicas que para Beck são essenciais ao aplicar o TDD. O conjunto de padrões *Test n*, *Isolated Test*, *Test List*, *Test-First*, *Assert First*, *Test Data* e *Evident Data* compõem a lista que foi introduzida na equipa.
  - *Test n*: Devemos escrever um teste automatizado sempre que existam mudanças no sistema, de forma a garantir que o comportamento será mantido, e que as novas alterações não comprometeram o estado do sistema.
  - *Isolated Test*: A execução de testes deve ser completamente isolada, ou seja, cada teste não pode afetar nenhum outro teste. Beck defende que os testes devem ser rápidos de executar, permitindo uma análise contínua ao resultado de todos os testes e com isso garantir que nada está errado, e que o resultado de um teste é independente do resultado dos restantes, garantindo que o problema é detetado de forma clara e precisa.
  - *Test List*: Deve ser feita uma lista de testes que à *priori* terão de ser escritos antes de começar a escrever código. O objetivo é perceber exatamente aquilo que queremos realizar, pensando antes de executar, colocando na lista os cenários que temos em mente quando pensamos no problema. À medida que os testes são executados, podem surgir novas necessidades de acrescentar testes, e nesse caso, adicionamos à lista. "A primeira parte de nossa estratégia para lidar com o *stress* da programação é nunca dar um passo adiante, a menos que saibamos onde o nosso pé vai pousar."(Beck 2003)
  - *Test-First*: Devemos testar primeiro, antes de escrever o nosso código. Existe a necessidade de pensar no *design* e ter um método para controlar o comportamento da nossa funcionalidade, ao desenhar o teste primeiro torna-nos mais propícios a testar e assim não ignoramos este passo.
  - *Assert First*: Devemos escrever as afirmações do teste primeiro. Beck defende esta prática como sendo um processo simplificador e esclarecedor.
  - *Test Data*: Os dados de entrada para os testes devem ser fáceis de ler e seguir. Exceto os testes que precisam de entradas precisas com determinados dados, as entradas devem ser combináveis. Quando um teste contém um cenário com várias entradas talvez estamos presentes a um caso onde devemos repensar o nosso *design* e implementação.
  - *Evident Data*: Devemos incluir resultados esperados e reais no teste, criando uma relação entre os dados de entrada e o resultado esperado. O objetivo passa por escrever o teste de forma clara para que o próximo leitor consiga compreender de forma mais clara o cenário, os dados de entrada e o resultado esperado.

- Estratégias de implementação: Segundo (Beck 2003), o TDD não é seguir cegamente um conjunto de regras sobre como programar, trata-se de escolher inteligentemente o tamanho dos passos a dar e a quantidade de *feedback* dependendo das condições. As estratégias de implementação introduzidas tem como fonte o seu livro, sendo compostas pelo seguinte conjunto: *Fake It ('Til You Make It)*, *Triangulate*, *Obvious Implementation* e *One to Many*. Este conjunto de conceitos para além de introduzidos foram exemplificado com casos reais que a equipa de desenvolvimento contacta frequentemente, de modo a criar uma empatia e sensibilização maior por estas estratégias, mostrando a sua utilidade e aplicabilidade.

  - *Fake It ('Til You Make It)*: É uma técnica que consiste em retornar uma constante após ter o respetivo teste a falhar. Sendo que, depois de executar o teste e este passar, a constante será transformada gradualmente numa expressão com uso de variáveis. (Beck 2003) defende que este método é muito poderoso por dois motivos, fator psicológico e controlo de *scope*. Relativamente ao fator psicológico, ter uma barra verde é completamente diferente do que não ter uma barra verde. Quando a barra está verde, o *refactor* é feito com muito mais confiança. Quanto ao controlo de *scope*, os *developers* são bons a imaginar todos os tipos de problemas futuros, e começar com um exemplo concreto e generalizar a partir daí impede que se confundam prematuramente com preocupações estranhas.

  - *Triangulate*: Técnica de triangulação que consiste em escrever dois ou mais testes para validar determinada unidade mais complexa. "Só uso *triangulate* quando não tenho muita certeza sobre a abstração correta para o cálculo. Caso contrário, confio em *Obvious Implementation* ou *Fake It*" (Beck 2003).

  - *Obvious Implementation*: Quando estamos perante um cenário de uma implementação simples, basta implementar. As técnicas *Fake It* e *Triangulate* são passos curtos que podem ajudar a desbloquear a implementação, mas quando temos a certeza da implementação, devemos simplesmente implementar.

  - *One to Many*: Técnica que pode ser aplicada em casos de implementação que utilizem listas de objetos. Neste caso, (Beck 2003) defende que a implementação deve ser feita tendo em conta um objeto singular, e após funcionar, implementar para receber uma lista de objetos.

- Técnicas de teste: Como o processo de teste é muito importante nesta metodologia, (Beck 2003) agrupou um conjunto de técnicas de teste que pretendem facilitar e resolver problemas comuns na prática de testes. Este conjunto composto por *Clean Check-in*, *Broken Test*, *Crash Test Dummy*, *Log String*, *Mock Object* e *Child Test*, também foi introduzido à equipa com o auxilio de casos reais.

  - *Child Test*: Perante um caso de teste que se torna muito grande, esta técnica consiste em escrever pequenos casos de teste, fazendo-os funcionar iterativamente, e por fim, juntar todos os casos em apenas um. "Fazer três coisas funcionarem ao mesmo tempo era demais. Se eu tivesse A, B e C a trabalhar, fazer com que os três funcionassem ao mesmo tempo seria uma coisa fácil" (Beck 2003).

  - *Mock Object*: Técnica que consiste em criar uma versão falsa de um determinado componente, permitindo ser configurável o seu comportamento. Esta necessidade surge quando é preciso testar um objeto que depende de serviços

externos. O exemplo clássico é um teste que execute uma unidade que dependa de uma base de dados, que levam muito tempo para serem iniciados, são difíceis de manter limpos. Com o uso de *Mock Object*, esta dependência é facilmente criada sem comprometer a velocidade do teste.

- *Log String*: Técnica utilizada para testar uma sequência de chamadas. Para isso, pode ser utilizado um *log* que é escrito a cada chamada, contendo a informação da sequência.
- *Crash Test Dummy*: Utilização de um objeto especial que lança uma exceção, utilizado para replicar um cenário de erro e verificar o comportamento do sistema. Esta técnica ajuda no sentido em que não é necessário replicar um ambiente real para simular este comportamento.
- *Broken Test*: Técnica utilizada quando se está a programar sozinho, onde o último teste deve falhar, para que quando o executante retomar a sessão de programação se lembre exatamente no ponto onde estava.
- *Clean Check-in*: Consiste em sair de um sessão de programação em equipa e deixar todos os testes a executar com sucesso, de forma a não comprometer o trabalho futuro do resto da equipa.

- *Refactor*: No TDD, *refactoring* é um passo muito presente e utilizado, pois é o ultimo passo do ciclo *Red, Green, Refactor*. As técnicas de *refactor* apresentadas à equipa foram: *Reconcile Differences*, *Isolate Change*, *Migrate Data*, *Extract Method*, *Inline Method*, *Extract Interface*, *Move Method*, *Method Object*, *Add Parameter* e *Method Parameter to Constructor Parameter*. Todos estes conceitos foram baseado no livro de (Beck 2003), onde refere que "o *refactor* não pode alterar o comportamento do programa sob nenhuma circunstância". Estes princípios são tão ou mais importantes que todos os conceitos já mencionados, pois é importante para a equipa perceber e saber como proceder ao *refactor* de forma segura sem alterar o comportamento aplicacional.
  - *Reconcile Differences*: Unificar duas partes do código semelhantes, eliminando a duplicação.
  - *Isolate Change*: Isolar determinada parte do código que precisa de ser alterada antes de proceder à alteração.
  - *Migrate Data*: Técnica que consiste em duplicar temporariamente o componente que se pretende migrar, contendo assim uma parte interna e externa. Na parte interna será alterada a representação e, em seguida, será alterada a interface visível.
  - *Extract Method*: Processo de criação de um novo método simples quando estamos perante um método longo e difícil de ler. Após a criação do novo método, o método original terá de o invocar, mantendo o comportamento da unidade.
  - *Extract Interface*: Criar uma interface com base nas operações de uma classe.
  - *Move Method*: Mover um método para a classe corresponde e em seguida, relacionar todas as invocações à nova localização.

  - *Add Parameter*: Processo de adição de um parâmetro, que passa por adicionar o parâmetro à interface primeiro, e em seguida usar os erros do compilador para informar qual código será necessário alterar.
  - *Method Parameter to Constructor Parameter*: No caso onde vários métodos recebem como argumento o mesmo parâmetro, a API pode ser simplificada com a passagem desse argumento para o construtor da classe, eliminando a duplicação e simplificando o código.

Relativamente aos principais desafios/limitações que o TDD apresenta 2.7.1, em comparação com a realidade apresentada neste estudo de caso, podemos considerar que alguns aspetos contribuíram para reduzir o atrito da sua adoção como:

- O facto de todos os membros da equipa estarem bastante familiarizados com a realização de testes e boas práticas dos mesmos ajudou, pois o TDD implica a realização de testes de forma constante.
- O facto de o projeto não ser um projeto *legacy* ajudou a introduzir e seguir o TDD de uma forma mais clara. Todos os desenvolvimentos foram realizados de raiz, sendo que a equipa teve a liberdade para construir todas os componentes do *software* sem dependência de código *legacy*.
- O pouco conhecimento desta técnica, apesar de ter sido uma realidade, foi minimizado com a introdução de técnicas de TDD ao longo do tempo, bem como sessões de acompanhamento e sessões de aplicação desta técnica em conjunto.

5.6 Behavior-Driven Development

O BDD, tal como o TDD 5.5, foi introduzido e implementado sobre a equipa de desenvolvimento gradualmente ao longo do tempo. A equipa relativamente ao BDD já tinha um certo conhecimento desta prática, pois já tinha experienciado antes de uma forma superficial, o que facilitou toda a introdução e aplicação. Perante este cenário, várias apresentações foram realizadas de forma a introduzir os seguintes temas:

- Introdução ao BDD: A introdução teórica deste tema teve como base o seu autor (*Introducing BDD* 2006). As suas motivações e qual é o principal problema a que o BDD se compromete a resolver, com base em exemplos que o próprio autor refere em (*Introducing BDD* 2006), serviram para consolidar as bases que a equipa já tinha, e fundamentar e discutir alguns conceitos que esta prática incorpora, como o *Gherkin*.
- Boas práticas: As boas práticas introduzidas serviram de apoio ao desenvolvimento saudável e escalável da solução de testes de aceitação. Baseadas em (Adams 2007) e (*Introducing BDD* 2006), o seguinte conjunto de boas práticas foi introduzida à equipa: *Behaviour Focus*, *Ubiquitous Language*, *Declarative Features*, *Avoid Conjunctive Steps*, *Reuse Step Definitions* e *Make Scenarios Atomic*.
  - *Behaviour Focus*: Deve existir foco no comportamento, de modo a definir cenários reais aos que o utilizador final irá utilizar.
  - *Ubiquitous Language*: A linguagem deve ser omnipresente, simples e transversal.
  - *Declarative Features*: Os cenários devem ser escritos como que se um utilizador os descrevesse.

  - *Avoid Conjunctive Steps*: Evitar criar ações conjuntas, ou seja, ações compostas com um "e"(*and*), por exemplo "*Given I am on the homepage and scrolled page down*". Perante este cenário, a ação deve ser dividida, ou seja, separar em dois *steps* diferentes.
  - *Reuse Step Definitions*: Devem ser reutilizadas as etapas (*steps*), de modo a melhorar a capacidade de manutenção, como por exemplo, quando é necessário alterar um determinado comportamento, basta alterar sobre uma etapa única.
  - *Make Scenarios Atomic*: O cenário deve ser executado independentemente, sem dependências de outros cenários. Isso ajuda o processo de depuração na percepção de erros.
- Introdução ao *SpecFlow*: Foram realizadas sessões de experimento com BDD e o uso da *framework SpecFlow*, de forma a familiarizar e ajudar os membros a utilizarem esta ferramenta, diminuindo a curva de aprendizagem inicial.

Relativamente aos desafios/limitações do uso do BDD, a seguinte enumeração irá apresentar os diferentes motivos:

- Utilização do *SpecFlow3*: A implementação do *SpecFlow3* na solução foi difícil, pois a *framework* apesar de ser compatível com o tipo de projeto *ASP.NET Core 2*, requer algumas configurações não triviais para evitar erros inesperados, erros de compilação, etc. Após a configuração ficar concluída, a equipa não experenciou nenhum problema adicional, sendo que podemos concluir que apenas na fase inicial é que os problemas surgiram.
- Definição de cenários: A definição de cenários não foi tão trivial como esperado, pois a maioria dos exemplos documentados em artigos remetem para sistemas onde normalmente apresentam interface gráfica, e o processo de visualização surge como um forte fator de ajuda na definição dos cenários. Neste caso, como o projeto é uma API, a adaptação foi um pouco mais lenta, mas após o paradigma de pensamento estar alinhado, os cenários fluíram de forma muito natural.
- *Feedback* do cliente: Infelizmente a equipa não conseguiu experenciar o contacto com o cliente, de forma a o envolver no processo, quer na definição de cenários como no seu *feedback*.

5.7 Solução de testes

A solução de testes (ou bateria de testes) foi desenvolvida e pensada de acordo com muitos dos princípios defendidos por autores como Kent Beck (Beck 2003), Martin Fowler (Fowler 2014), Dan North (*Introducing BDD* 2006) e (Cheon e Leavens 2002). A seguinte lista detalha as principais caraterísticas da solução de testes implementada:

- Segregação: A solução de testes está dividia em camadas, respeitando a pirâmide de testes (Luo 2001), testes de unidade, integração e aceitação.
- Isolamento: Todas as camadas estão isoladas de forma independente, e dentro de cada camada, cada teste é completamente isolado.
- Rapidez: A solução de testes executa num curto intervalo de tempo, dando *feedback* recorrente às alterações introduzidas.

- Escalabilidade: Permite que sejam desenvolvidos novos testes de forma rápida e segura, graças à sua infraestrutura.
- Encapsulamento: A solução de testes encapsula todas as suas dependências e tem autonomia para executar em qualquer ambiente de forma isolada.
- Paralelização: Todos os testes, independentemente da camada a que pertencem, podem ser executados de forma paralela em simultâneo.

A figura 5.4 apresenta a atual estrutura da solução de testes. Dentro da secção *Tests* podemos observar três diferentes secções: *Unit* (Testes de unidade), *Integration* (Testes de integração) e *End-2-End* (Testes de aceitação). Esta estrutura tem como base a pirâmide de testes 2.1, segundo (Luo 2001).

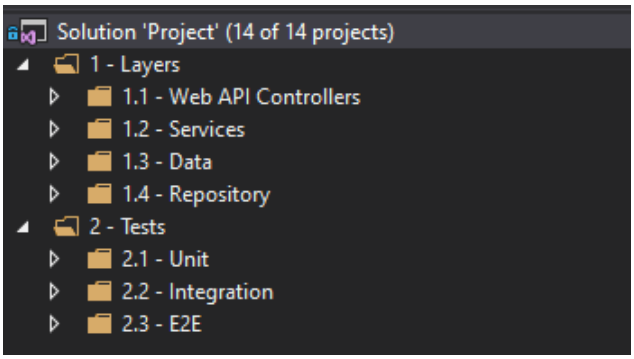

Figura 5.4: Solução de testes - Estrutura

Relativamente ao encapsulamento de todos os testes dentro da solução e do IDE, a figura 5.5 apresenta o resultado de uma execução de todos os testes dentro do próprio IDE.

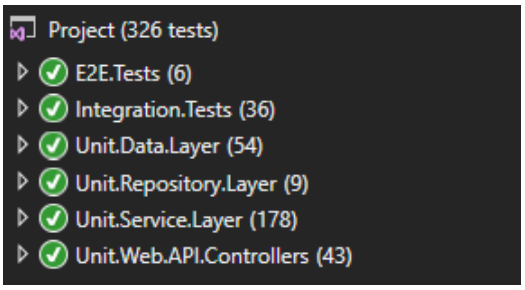

Figura 5.5: Solução de testes - Execução

5.7.1 Componentes

Os componentes utilizados para estruturar a solução de testes foram escolhidos com o intuito de satisfazer os requisitos afirmados em 5.7, sempre com o objetivo de facilitar o desenvolvimento e tornar a suite de testes amigável. A seguinte lista apresenta os componentes utilizados na solução de testes:

- *Moq*: Esta *framework* serve essencialmente para utilizar *mocks* de forma simples e rápida. Apresenta uma performance excelente, não impactando a velocidade dos testes.

Permite simular comportamentos e verificar ações realizadas. "A sua API é extremamente simples e direta, e não requer nenhum conhecimento prévio ou experiência com conceitos de *Mock*"(Moq 2019).

- *FluentAssertions*: Permite que de forma natural e simples se definam resultados esperados da execução dos testes, facilitando a sua escrita e o seu *assert*. Segundo (FluentAssertions 2019), esta *framework* é "um conjunto muito extenso de métodos que permitem especificar de forma natural o resultado esperado de um teste sobre o estilo TDD ou BDD".
- *xUnit*: *Framework* apresentada em 2.2.1, foi utilizada de forma recorrente em todas as camadas de testes.
- *Live-Unit-Tests*: *Plug-in* que se apresenta como uma peça chave no que diz respeito ao *feedback* contínuo 2.2.3.
- *WireMock*: Referido em 2.2.8, *WireMock* com a sua versatilidade permitiu simular chamadas externas da aplicação, isolar os testes de forma independente e como consequente, permitir a sua paralelização.
- *Mongo2Go*: Componente capaz de criar e fornecer bases de dados *MongoDB* temporários e isolados para testes de unidade ou integração (Mongo2Go 2019b), o que permite mais uma vez criar ambientes isolados para cada teste.
- *Microsoft.AspNetCore.Mvc.Testing*: Ferramenta utilizada para criar o ambiente para testes de integração e aceitação de forma isolada (Microsoft 2019c), permitindo preparar todo o *setup* inicial de acordo com o teste em questão (Microsoft 2019b) .
- *SpecFlow 3*: Ferramenta 2.2.6 utilizada para testes de aceitação.

Ao longo das secções 5.7.2, 5.7.3 e 5.7.4 os componentes referidos serão evidenciados e exemplificados de forma a sinalizar a sua utilização em cada camada de testes.

5.7.2 Testes de unidade

Os testes de unidade ocupam a maior percentagem de testes que compõe a solução. Para uma melhor compreensão de como o processo é realizado na realização de um teste de unidade, o diagrama 5.6 apresenta a sequência de passos que o *developer* executa.

Tal como exemplifica a figura 5.6, o *developer* inicia o processo de escrita de um teste de unidade seguindo a anotação dos três As 2.1, de seguida e de forma opcional, pode escrever o *Assert* (Asserto) do teste, prática do TDD *Assert First*. Após isso, o teste deve falhar e a escrita do corpo do mesmo deve prosseguir, seguindo os padrões TDD referidos em 5.5. Após o teste concluído, a escrita do código de produção deve ser realizada até que o todos os testes passem. O último passo é o processo de *refactor*, que serve para eliminar a duplicação e limpar o código, estando este processo terminado quando todos os testes passarem.

O diagrama 5.7 apresenta os componentes utilizados na soluções de testes de unidade, sendo eles o *FluentAssertions*, *Moq*, *Live-Unit-Testing* e *xUnit*.

O excerto 5.1 mostra um exemplo de um teste de unidade. O teste utiliza a norma dos três As 2.1 e o próprio nome enfatiza aquilo que (Reese, A. D. George e Wenzel 2018)

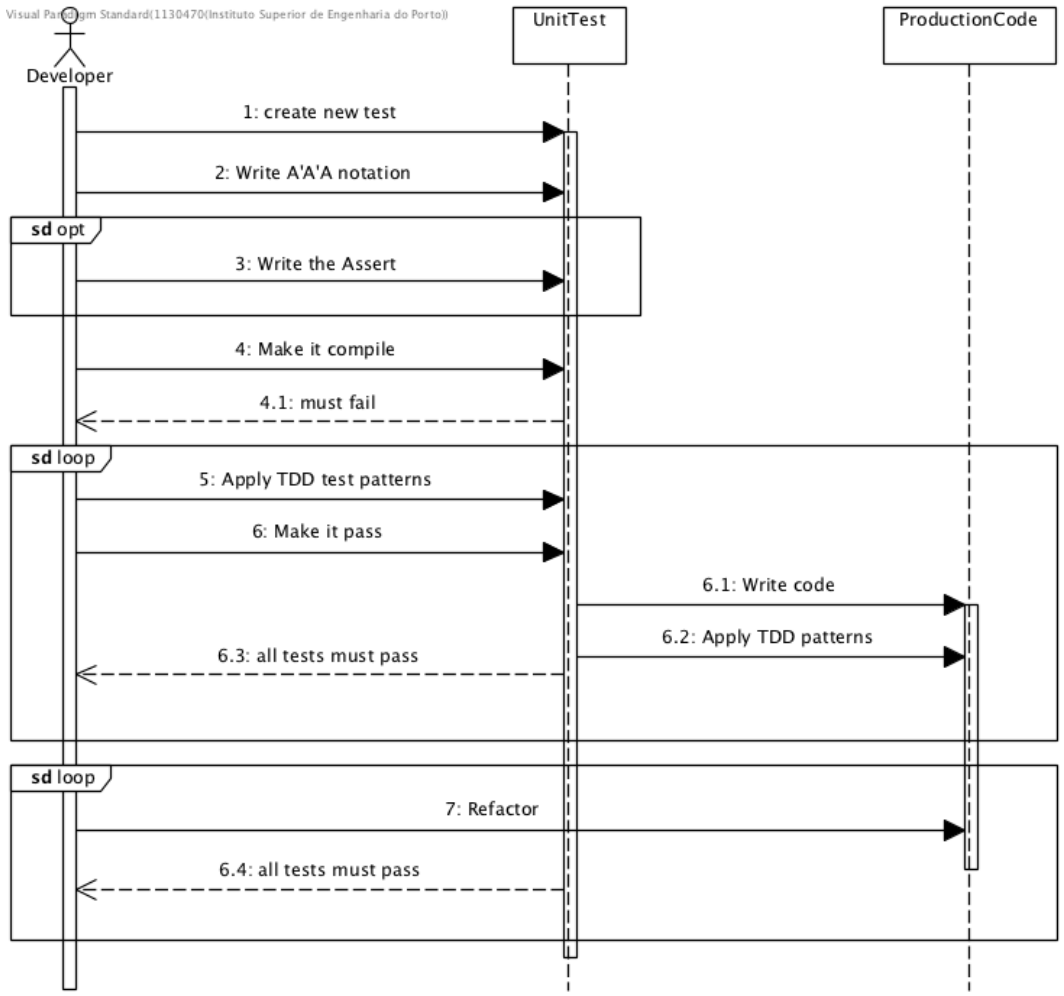

Figura 5.6: Diagrama de sequência - Testes de unidade

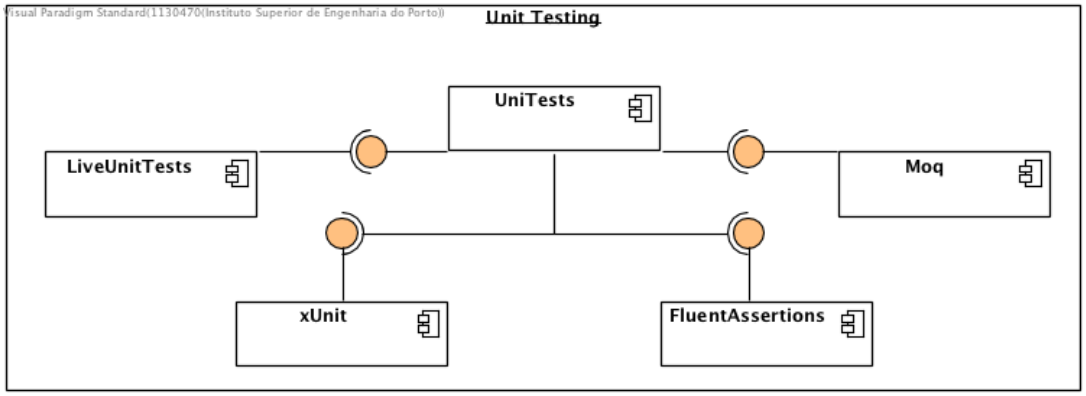

Figura 5.7: Diagrama de componentes - Testes de unidade

defende, referido em 2.1.1. Utiliza *FluentAssertions* para configurar um resultado esperado, facilitando o processo de escrita pois a ferramenta utiliza linguagem natural.

```
1 [Fact]
2 public void Convert_User_ConvertUser()
3 {
4     // Arrange
```

```
5       var user = new User
6       {
7           Email = "lorem@email.com",
8           Name = "Lorem Ipsum",
9           UserName = "lorem.ipsum",
10          PhoneNumber = "3290082"
11      };
12
13      var expected = new Dictionary<string, string>
14      {
15          { "FirstName", "Lorem" },
16          { "LastName", "Ipsum" },
17          { "PersonEmail", "lorem@email.com" },
18          { "PhoneNumber", "3290082" }
19      };
20
21      // Act
22      var actual = this.sut.Convert(user);
23
24      // Assert
25      expected.Should().Be(actual);
26  }
```

Listing 5.1: Exemplo de teste de unidade

A nível de unidade, quando surge a necessidade de simular um comportamento de um dependência, o *Mock Object* 5.7.1 é utilizado de forma a construir o *setup* do comportamento esperado, tal como podemos observar no excerto 5.2, onde o *Mock Object* está representado pela variável *apiClient*. Na linha 14, o seu *setup* é configurado de modo a simular o seu comportamento para a chamada ao método *Get* com o parâmetro *command*.

```
1   using Moq;
2
3   ...
4
5   [Fact]
6   public async Task GetUser_UserId_UserInformation()
7   {
8
9       // Arrange
10      var expected = new ApiResponse<User>();
11      var userId = "1";
12      var command = $"/Users/'{userId}'";
13      var apiClient = new Mock<IApiClient>();
14
15      apiClient.Setup(aC => aC.Get<ApiResponse<User>>(command)).
        ReturnsAsync(expected);
16
17      // Act
18      var actual = await this.usersApi.GetUser(userId);
19
20      // Assert
21      Assert.Equal(expected, actual);
22  }
```

Listing 5.2: Exemplo de teste de unidade com *MockObject*

5.7.3 Testes de integração

Os testes de integração surgem como peça chave no que diz respeito à validação com dependências externas da aplicação, testes de contrato e comunicação com componentes internos. O diagrama 5.8 apresenta os componentes utilizados na solução de testes de integração. Para além dos referidos em 5.7.2, surge o *Mongo2Go*, *MvcTesting* e *WireMock*.

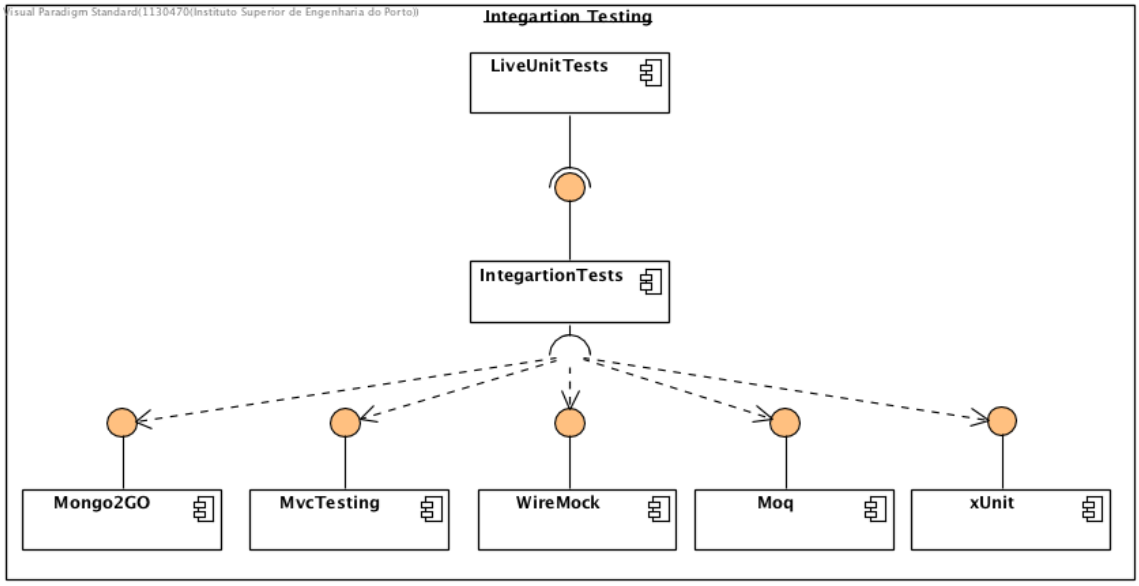

Figura 5.8: Diagrama de componentes - Testes de integração

O diagrama de sequência 5.9 exemplifica como um *developer* cria um novo teste de integração, utilizando os componentes presentes em 5.8. Seguindo a mesma linha de pensamento aplicada nos testes de unidade, relativamente ao padrão TDD *Assert first*, neste caso pode ser também aplicado de forma opcional. A instanciação dos componentes é muito simples, permitindo a que o *developer* construa rapidamente um ambiente à imagem da sua necessidade. De forma a complementar este diagrama, alguns excertos de código irão ser apresentados.

O excerto 5.3 apresenta um exemplo de um teste de integração, que à semelhança dos testes de unidade, segue a estrutura dos três As 2.1. No excerto, podemos observar os seguintes comportamentos:

- Criação do *FluentMockServer*, instância da ferramenta do *WireMock*. O teste pode usufruir dessa instância isolada, pois contém uma porta única. O teste é responsável também por enviar a simulação das chamadas que a aplicação irá fazer para o exterior (*server.StubRequest(...)*), que por sua vez o *WireMock* irá intercetar e responder com o que foi configurado nessa simulação.
- Um novo *client*, através do *WebApplicationFactory* que utiliza o *MVCTesting* 5.7.1. O teste utiliza uma nova instância da aplicação alvo, que pode utilizar de forma isolada.
- Criação de um pedido HTTP para a aplicação alvo *Endpoint*, com um *input* previamente definido.
- A validação do resultado
- Por fim, as instâncias *FluentMockServer* e *client* serão destruídas.

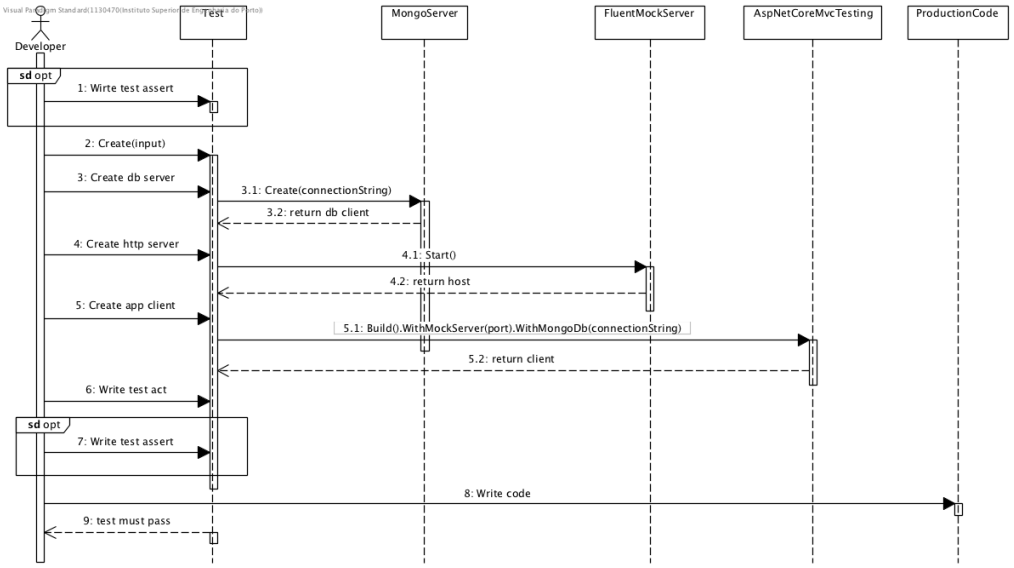

Figura 5.9: Diagrama de sequência - Criação de teste de integração

```
[Fact]
public async Task Users_UserEvent_PostUser()
{
    // Arrange
    var expected = 201;
    var input = new UserEvent();

    using (var server = FluentMockServer.Start())
    {
        server.StubRequest(new UserRequest());

        using (var client = new WebApplicationFactory().Build().
    WithMockServer())
        {
            // Act
            var response = await client.PostAsync("Users", input);

            // Assert
            Assert.True(response.StatusCode(), expected);
        }
    }
}
```

Listing 5.3: Exemplo de teste de integração com *WireMock*

O teste de integração 5.4 mostra um exemplo do uso do *Mongo2Go*, utilizado para criar uma base de dados *MongoDB* para o efeito de teste. A sua utilização é semelhante aos outros componentes mencionados, o *MongoServer* é responsável por criar uma nova instância da base de dados utilizando uma determinada conexão (*ConnectionString*), que por sua vez a instância da aplicação alvo irá utilizar. Após o *Assert*, a instância é destruída tal como todas as dependências criadas.

```
[Fact]
public async Task User_UserEvent_SaveUserIntoDB()
{
    // Arrange
```

```
5       var connectionString = "connStringTestSaveUserIntoDB";
6       var input = new UserEvent();
7
8       using(var dBserver = MongoServer.Create(connectionString))
9       using(var client = new WebApplicationFactory().Build().WithMongoDb(
        connectionString))
10          {
11              // Act
12              var response = await client.PostAsync("LocalUsers", input);
13
14              // Assert
15              var actual = await dBserver.GetCollection<Users>("users").
        CountDocumentsAsync();
16
17              Assert.Single(actual);
18          }
19      }
20  }
```

Listing 5.4: Exemplo de teste de integração com base de dados

5.7.4 Testes de aceitação

Os testes de aceitação (ou *End-2-End*) tem um papel muito importante na validação dos requisitos da aplicação. O diagrama de sequência 5.10 apresenta os passos que o *developer* executa na escrita deste tipo de testes. O *developer* após a criação do teste de aceitação, escreve a notação BDD utilizando *Gherkin*, os *Step definitions* (passos do teste) que representam cada linha do texto escrito previamente em *Gherkin*. Seguindo o mesmo princípio do TDD, o teste deve falhar, e de seguida o *developer* é responsável por escrever o código de produção até que o teste passe.

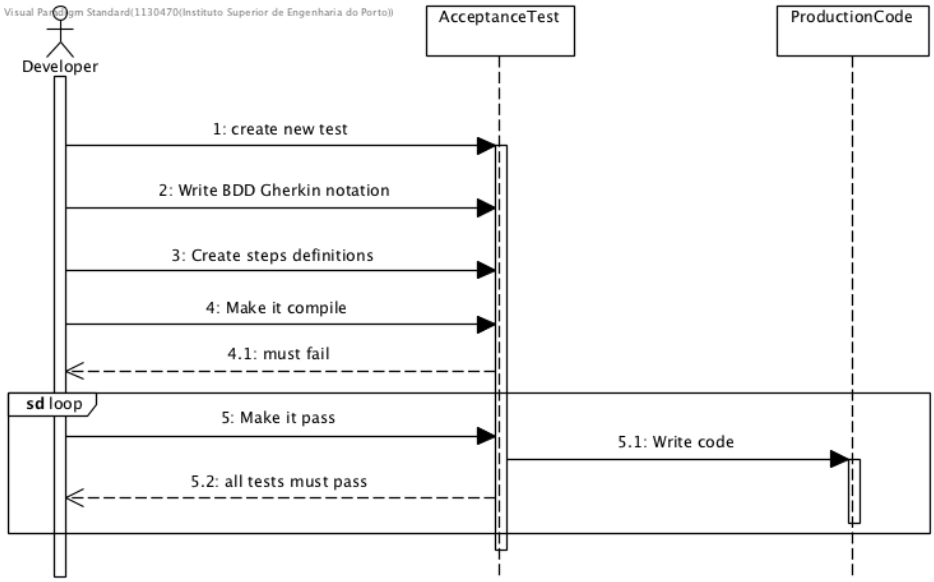

Figura 5.10: Diagrama de sequência - Testes de aceitação

O diagrama de componentes 5.11 apresenta os componentes utilizados na solução de testes de aceitação. Relativamente aos testes de integração 5.7.3, surgem dois novos componente, o *SpecFlow* e o *Gherkin*.

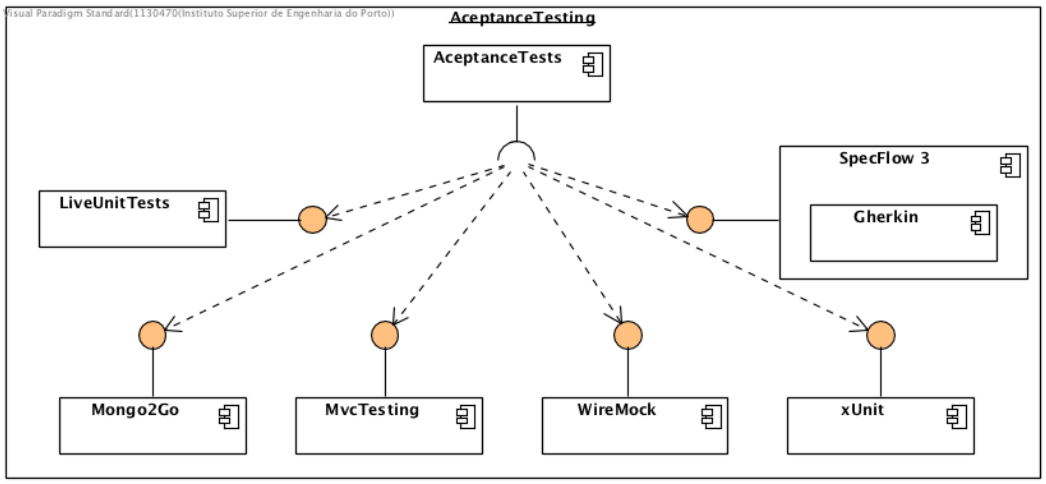

Figura 5.11: Diagrama de componentes - Testes de aceitação

O *SpecFlow* é o componente que permite a utilização do BDD, com o auxilio do *Gherkin*. A figura 5.5 apresenta um exemplo de um teste de aceitação com a utilização do *Gherkin*. A *Feature* é definida com base no contexto dos testes a realizar e o *Scenario* representa o critério de aceitação.

```
Feature: User
  As a API client consumer
  I want to manage the users
  So that the system can process the users

Scenario: Get user
  Given A 'request' user event
  When I call the 'Users/1' route
  Then the user should be 'retrieved'

Scenario: Create user
  Given A 'creation' user event
  When I call the 'Users' route
  Then the user should be 'created'

Scenario: Delete user
  Given A 'delete' user event
  When I call the 'Users/1' route
  Then the user should be 'deleted'
```

Listing 5.5: Exemplo de teste de aceitação

O excerto de código 5.6 mostra a correspondência dos passos definidos em 5.5. O *SpecFlow* ajuda no processo de criação e correlação entre o *Gherkin* e o código correspondente. Estes passos podem ser reutilizáveis de modo a reduzir redundância e duplicação de código.

```
[Binding]
public class UserTestsSteps
{
    private EventType @event;
```

```
5       private HttpResponseMessage result;
6
7       ...
8
9       [Given(@"A '(.*)' user event")]
10      public void GivenAUserEvent(EventType eventType)
11      {
12          this.@event = eventType;
13      }
14
15      [When(@"I call the '(.*)' route")]
16      public void WhenICallTheRoute(string route)
17      {
18          using (var dBserver = MongoServer.Create(connectionString))
19          using (var server = FluentMockServer.Start())
20          using (var client = new WebApplicationFactory()
21              .Build()
22              .WithMockServer()
23              .WithMongoDb(connectionString)
24              .Create())
25          {
26              this.result = await this.CallRoute(this.@event, route);
27          }
28      }
29
30      [Then(@"the user should be '(.*)'")]
31      public void ThenTheUserShouldBe(ActionType actionType)
32      {
33          var actual = this.VerifyAction(actionType, this.result);
34          Assert.IsTrue(actual, this.GetErrorMessage(actionType));
35      }
36  }
```

Listing 5.6: Exemplo de teste de aceitação - Passos

Um dos desafios desta implementação foi perceber como e qual a melhor forma de guardar o contexto entre *step definitions*. Como o contexto não pode ser guardado no navegador web (como acontece em aplicações gráficas), o contexto tem de ser gerido internamente e após a conclusão do teste, tem de ser destruído. No excerto 5.6 podemos observar a existência de duas variáveis privadas de classe que são utilizadas ao longo da execução dos *step definitions*, nomeadamente o *@event* que representa o tipo de evento a considerar (request, creation e delete) e *result*, que representa o resultado da operação efectuada. Após a instanciação dos *step definitions*, a classe *UserTestsSteps* guarda em memória o valor das variáveis, e à medida que a execução dos *step definitions* é realizada, o valor das variáveis é partilhado, de modo a ser utilizado no respetivo *step*.

5.8 Integração com CI/CD Pipeline

A solução desenvolvida pela equipa está integrada sobre uma *Pipeline* de CI/CD, e para além disso, a bateria de testes foi também adicionada a esta *pipeline* de modo a executar todas as camadas de teste de forma automática, garantido que todos os passos de validação foram executados com sucesso antes de qualquer implantação.

A estratégia utilizada foi segmentar as diferentes camadas de testes (de unidade, integração e aceitação) em diferentes *stages* (estágios) da *Pipeline*, de modo a que seja possível identificar facilmente qual camada de testes falhou, estando cada *stage* responsável por executar determinada camada de testes. A figura 5.12 representa uma versão simplificada da *Pipeline* executada no *Jenkins* 2.4.2, com os diferentes *stages* responsáveis por executar os diferentes tipos de testes.

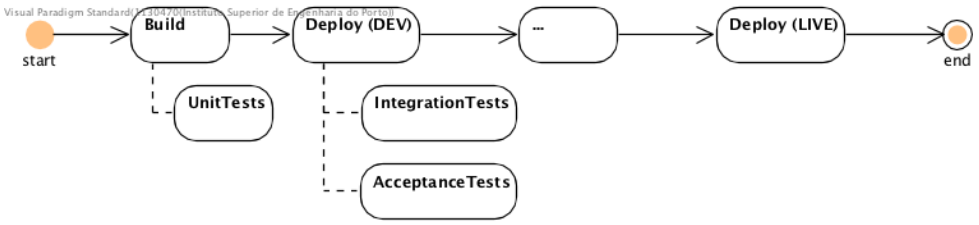

Figura 5.12: Pipeline - *Stages*

Após a execução do *stage build*, os testes de unidade *UnitTests* são executados e seguidamente, após um *Deploy* sobre um ambiente de desenvolvimento isolado, são executados os testes de integração *IntegrationTests*. Os teste de aceitação *AcceptanceTests* são executados sobre uma réplica do ambiente de desenvolviemnto. Após serem executados todos os outros *stages* (sinalizados com "...") é feita a implantação nos servidores de produção.

Capítulo 6

Avaliação

O capítulo de avaliação tem como objetivo descrever o processo de avaliação de resultados deste estudo de caso. As secções que compõe este capítulo são: definição da hipótese central, definição das grandezas, definição das hipóteses, apresentação das metodologias de avaliação e atribuição aos respetivos indicadores e por último, a avaliação de resultados.

6.1 Hipótese

Será que a aplicação do desenvolvimento orientado aos testes, em contexto empresarial, permitirá melhorar a qualidade dos produtos e do processo de desenvolvimento de software?

A hipótese central deste estudo de caso tem como objetivo determinar se de facto, o TDD e BDD desenhado e adaptado à realidade empresarial, ajuda a melhorar a qualidade do software produzido, bem como o processo de desenvolvimento. Para que a resposta seja determinada, serão definidas grandezas e hipóteses.

6.2 Definição das grandezas

A definição das grandezas tem como propósito ajudar a avaliar o resultado deste estudo de caso. A sequência seguinte representa o conjunto de grandezas que serão utilizadas:

- Satisfação da equipa: o objetivo é analisar se a equipa se sente confortável com a nova solução
- Adoção da nova solução: analisar se a equipa usou de forma contínua a solução desenhada.
- Produtividade da equipa: o objetivo é analisar se a produtividade da equipa aumentou ou diminuiu com a implementação da nova solução.
- Defeitos no código: esta grandeza irá medir o número de defeitos no código em produção. Será um forte indicador para perceber a eficácia desta nova solução.
- Cobertura do código: a cobertura do código servirá para medir a percentagem de código coberto pela bateria de testes.
- Qualidade dos testes: indicador importante no que diz respeito à qualidade da bateria de testes, refletindo o quão forte é a nossa bateria de testes e a confiança que podemos ter sobre ela.

- Tempo de execução da bateria de testes: o tempo de execução da bateria de testes deverá ser o mais curto o quanto possível, para que o *feedback* seja imediato.

6.3 Definição das hipóteses

Após definidas as grandezas, as hipóteses presentes nesta secção visam definir o que se pretende efetivamente avaliar.

- Aumento da satisfação da equipa: É expectável que a equipa se sinta satisfeita com a nova solução, sendo que a percentagem de satisfação deve ser superior a 90%.
- Adoção continua da nova solução: É esperado que a equipa adote a solução de forma natural e de forma constante, e por fim, a considere eficaz e adequada ao problema. A percentagem de utilização deverá ser superior a 80%.
- Aumento da produtividade da equipa: É esperado que a produtividade da equipa diminua nos primeiros tempos de introdução da nova metodologia, mas que aumente ao longo do tempo. É esperado que este valor aumente relativamente ao modelo base.
- Diminuição de defeitos em produção: É expectável uma diminuição do número de defeitos (*bugs*) em código de produção com a introdução da nova solução.
- Aumento da cobertura do código: Atualmente a cobertura do código é de cerca de (60%) em todas as soluções de software que a equipa trabalha. É esperado um aumento entre 20% a 30%.
- Aumento da qualidade dos testes: A percentagem de mutantes deve ser inferior a 5%, ou seja, é esperado que os testes aumentem a sua qualidade, para a bateria de testes fique mais forte.
- Diminuição do tempo da bateria de testes: É esperada uma redução do tempo da bateria de testes, que incluem todo o tipo de testes (unidade, integração e aceitação). Este valor deve ser inferior a 100 segundos.

6.4 Metodologias de avaliação

Esta secção tem como objetivo apresentar os diferentes métodos de avaliação, nomeadamente os inquéritos de satisfação e os testes estatísticos.

6.4.1 Inquéritos de satisfação

Os inquéritos de satisfação serão utilizados com o objetivo de avaliar as grandezas de ordem subjectiva. Este caso de estudo envolve a alteração do processo de desenvolvimento e consequentemente, todos os intervenientes desse processo. Para que grandezas diretamente ligadas a este processo possam ser avaliadas, os inquéritos serão a abordagem a seguir.

Os questionários serão compostos por 5 tipos de resposta, que seguindo o modelo de Likert Scales (Allen e Seaman 2007) (H. N. Boone e D. A. Boone 2012) (J. A. Gliem e R. R. Gliem 2003), são de 1 a 5. A resposta 1 representa "Muito Insatisfeito", 2 representa

"Insatisfeito", 3 representa "Neutro", 4 representa "Satisfeito"e por último, 5 representa "Muito Satisfeito".

6.4.2 Testes estatísticos

Para avaliar a solução final de acordo com as hipóteses e grandezas definidas, podemos optar pelo teste *One-Tailed Test* (paramétrico) ou *Two-Tailed Test* (não-paramétrico).

Segundo (Stone 2019), antes de optar por um dos testes devemos analisar o que pretendemos testar. A sua sugestão recai sobre duas questões:

1. O resultado é maior (ou menor que) um determinado valor?
2. O resultado está dentro ou fora de um determinado intervalo de valores?

Se estivermos presentes sobre o primeiro cenário, a opção recai para o teste paramétrico, enquanto que o segundo cenário de enquadra no teste não-paramétrico.

Paramétrico (Ruxton e Neuhäuser 2010) e (University 2018) referem que um teste paramétrico resulta de uma hipótese alternativa que especifica uma direção. Ou seja, quando a hipótese alternativa afirma que o parâmetro é de fato maior ou menor que o valor especificado na hipótese nula.

O teste paramétrico pode ser à esquerda, $H1 : \mu < \mu 0$ onde é usado quando a hipótese alternativa afirma que o valor verdadeiro do parâmetro especificado na hipótese nula é menor do que a hipótese nula reivindicada. Um teste paramétrico à direita $H1 : \mu > \mu 0$ é usado quando a hipótese alternativa afirma que o valor verdadeiro do parâmetro especificado na hipótese nula é maior do que as hipóteses de hipótese nula.

Não-Paramétrico Um teste não-paramétrico, segundo (Ruxton e Neuhäuser 2010) e (University 2018), resulta de uma hipótese alternativa que não especifica uma direção. ou seja, quando a hipótese alternativa afirma que a hipótese nula está errada. A principal diferença entre os testes paramétrico e não-paramétrico é que os testes paramétrico terão apenas uma região crítica, enquanto os testes não-paramétrico terão duas regiões críticas. Se precisarmos de um % de intervalo de confiança, temos que fazer alguns ajustes ao usar um teste não-paramétrico. O intervalo de confiança deve permanecer sobre um tamanho constante, portanto, se estivermos perante um teste não-paramétrico, como há duas vezes mais regiões críticas, essas regiões críticas devem ter metade do tamanho.

É importante perceber qual o caminho a seguir, e adaptar as duas opções às necessidades, pois a decisão sobre o uso de testes paramétrico ou não-paramétrico pode influenciar se uma ou mais hipóteses são aceites ou rejeitadas (Kock 2015).

Escolha do tipo de teste Analisando as duas alternativas e o propósito desta dissertação, fará sentido usar o teste paramétrico, visto que o que se pretende avaliar é se o novo processo é melhor que o processo anterior.

Para testes paramétricos, existem dois tipos de hipóteses: nulas e as alternativas (University 2018).

- $H0$: A hipótese nula é a hipótese considerada que o analista espera rejeitar.
- $H1$: A hipótese alternativa é a hipótese considerada que é apoiada por rejeitar a hipótese nula.

Neste tipo de testes, é necessária a definição de um grau de confiança. Por norma, o valor é de 95% (University 2018).

6.5 Atribuição dos métodos de avaliação

A metodologia de avaliação aqui documentada servirá de apoio à concretização das hipóteses. Foram escolhidos diferentes métodos para servir diferentes necessidades. A sequência seguinte apresenta as diferentes metodologias.

- Análise da satisfação da equipa: serão utilizados inquéritos e testes de hipóteses que servirão para avaliar a satisfação da equipa em relação à solução desenhada.
- Análise à adoção da nova solução: serão utilizados inquéritos, dados de monitorização e testes de hipóteses para perceber a percentagem de adoção da solução por parte da equipa.
- Monitorização da velocidade da equipa: teste de hipóteses. Será analisada a velocidade da equipa a cada *sprint* após o uso da nova solução. Será feita uma comparação com o modelo base.
- Monitorização de defeitos: teste de hipóteses. Será feita uma análise ao número de erros ocorridos em produção com o uso da nova solução. Em comparação, será utilizado o modelo base.
- Monitorização da cobertura do código: teste de hipóteses. Será analisada a cobertura do código produzido. O indicador presente será o *CodeCoverage* 2.3.1 utilizando o método de *Statement* referido na secção 2.3.1.
- Monitorizar a percentagem de mutantes vivos: teste de hipóteses. Será utilizada a metodologia de *MutantionTests*, referida na secção 2.3.2, para este efeito. Será criada uma rotina de análise à percentagem de mutantes vivos. Este valor deve ser próximo de nulo (inferior a 5%).
- Monitorização do tempo de execução da bateria de testes: teste de hipóteses. Para cumprir a métrica estabelecida, será feita uma análise constante ao tempo de execução da bateria de testes quando uma nova *build* entrar para a *pipeline*. Este indicador foi referido na secção 2.3.3.

6.6 Avaliação de resultados

Esta secção irá apresentar os resultados obtidos deste estudo de caso de acordo com as hipóteses definidas anteriormente. De modo a avaliar de forma coerente e precisa, algumas premissas foram definidas para que fosse seguida uma lógica de avaliação comum entre os indicadores.

- Período temporal: O período temporal foi definido com base na data de início da implementação deste estudo de caso na equipa de trabalho até 22 semanas depois, ou seja, o período temporal utilizado foi entre 11/02/2019 a 15/07/2019, que reflete um total de 11 *sprints*. Como alguns dos indicadores de avaliação requerem um período de comparação para ser possível comprar resultados, foi utilizado o mesmo período de tempo, 11 *sprints*, antes da implementação deste estudo de caso, ou seja, de 10/09/2018 a 11/02/2019. De modo a facilitar a referencia a estes dois períodos temporais ao longo desta secção, o período referente à aplicação deste caso de estudo será referenciado como **Período Estudo de caso** e o período referente a antes da aplicação como **Período Modelo Tradicional**.
- Inquiridos: Relativamente ao inquérito realizado A, os inquiridos responderam às questões com base no mesmo período temporal. Todos eles fazem parte da equipa que integrou este estudo de caso, tendo a oportunidade de experiênciar todos os conceitos aplicados ao longo deste período.

Por fim, a figura 6.1, apresenta a matriz de respostas por parte dos inquiridos. No total, 7 pessoas responderam ao questionário, sendo que todas elas responderam também a todas as questões. Perante este cenário, podemos considerar que todas as amostras são quantitativas, independentes, mas não podemos assumir uma distribuição normal, sendo a amostra inferior a 30 resultados.

Questão	Resposta				
	1	2	3	4	5
1	0	0	0	1	6
2	0	0	0	0	7
3	0	0	0	3	4
4	0	0	0	0	7
5	0	0	0	3	4
6	0	0	0	4	5

Figura 6.1: Matriz de respostas

6.6.1 Satisfação da equipa

Para avaliar este indicador, foram utilizadas as seguintes questões do questionário realizado aos membros da equipa:

- Questão 1 - Consideras que a solução desenhada se enquadra no âmbito da equipa?
- Questão 2 - Consideras que a solução desenhada ajuda de forma contínua a resolver os problemas identificados?
- Questão 4 - Consideras o uso do TDD uma mais valia durante o processo de desenvolvimento de forma a melhorar o software produzido?
- Questão 5 - Consideras o uso do BDD benéfico durante o processo de desenvolvimento de forma a validar o comportamento esperado do software?

Para avaliar a satisfação da equipa com base na métrica definida, onde refere que é expectável que a equipa se sinta confortável, com um grau de satisfação superior a 90%, irão ser avaliadas as questões referidas de forma singular de acordo com a métrica definida.

Questão 1 Como já referido anteriormente, apenas 7 respostas compõem a lista de resultados, logo não podemos assumir que estamos perante uma distribuição normal. Irá ser utilizado um teste *Shapiro-Wilk* para verificar o tipo de distribuição.

*H*0: A população segue uma distribuição normal
*H*1: A população não segue uma distribuição normal

O código presente em 6.1 apresenta o teste de *Shapiro-Wilks* realizado com a amostra de respostas relativas à questão 1.

```
> q1<-c(5,4,5,5,5,5,5)
> shapiro.test(q1)

  Shapiro-Wilk normality test

data:  q1
W = 0.45297, p-value = 4.136e-06
```

Listing 6.1: Teste Shapiro-Wilks - Questão 1

De acordo com 6.1, com intervalo de confiança de 95%, podemos concluir que como o *p-value* é inferior a 5% (nível de significância), a amostra não segue uma distribuição normal, logo um *t-test* não pode ser usado, sendo que irá ser aplicado um *Wilcoxon* teste, que é um teste não paramétrico.

De acordo com a definição de um grau de satisfação superior a 90% para esta questão, podemos considerar as seguintes hipóteses:

*H*0: Média de respostas é menor ou igual a 4.5
*H*1: Média de respostas é maior que 4.5

```
> wilcox.test(q1, mu = 4.5, alternative = "greater")

  Wilcoxon signed rank test with continuity correction

data:  q1
V = 24, p-value = 0.0363
alternative hypothesis: true location is greater than 4.5
```

Listing 6.2: Teste Wilcoxon - Questão 1

Após analisar o cálculo 6.2,como o *p-value* é inferior a 0.05, a hipótese nula é rejeitada, logo com 95% de confiança podemos concluir que a equipa considera que a solução se enquadra no seu âmbito.

Questão 2 Relativamente à questão 2, todos os inquiridos responderam com o valor 5, sendo a nossa amostra q2(5,5,5,5,5,5,5) estamos perante uma distribuição discreta, com um único valor com probabilidade 1, logo podemos concluir que a equipa considera, sem qualquer dúvida, que a solução ajuda de forma contínua a resolver os problemas identificados.

Questão 4 Sobre a questão 4, todos os inquiridos responderam com o valor 5, sendo a nossa amostra q3(5,5,5,5,5,5,5) estamos perante uma distribuição discreta, com um único valor com probabilidade 1, logo podemos concluir que a equipa considera que uso do TDD é sem dúvida alguma uma mais valia durante o processo de desenvolvimento de forma a melhorar o software produzido.

Questão 5 Irá ser utilizado um teste *Shapiro-Wilk* para verificar o tipo de distribuição, pois não podemos assumir que estamos perante uma distribuição normal.

*H*0: A população segue uma distribuição normal
*H*1: A população não segue uma distribuição normal

O código presente em 6.3 apresenta o teste de *Shapiro-Wilks* realizado com a amostra de respostas relativas à questão 5.

```
> q1<-c(4,4,5,5,5,4,5)
> shapiro.test(q1)

  Shapiro-Wilk normality test

data:  q1
W = 0.66444, p-value = 0.001497
```

Listing 6.3: Teste Shapiro-Wilks - Questão 5

De acordo com 6.1, com intervalo de confiança de 95%, podemos concluir que como o *p-value* é inferior a 5% (nível de significância), a amostra não segue uma distribuição normal, logo um *t-test* não pode ser usado, sendo que irá ser aplicado um *Wilcoxon* teste, que é um teste não paramétrico.

De acordo com a definição de um grau de satisfação superior a 90% para esta questão, podemos considerar as seguintes hipóteses:

*H*0: Média de respostas é menor ou igual a 4.5
*H*1: Média de respostas é maior que 4.5

```
> wilcox.test(q1, mu = 4.5, alternative = "greater")

  Wilcoxon signed rank test with continuity correction

data:  q1
V = 16, p-value = 0.3884
alternative hypothesis: true location is greater than 4.5
```

Listing 6.4: Teste Wilcoxon - Questão 5

Após analisar o cálculo 6.4, como o *p-value* é superior a 0.05, a hipótese nula não pode ser rejeitada, logo com 95% de confiança podemos concluir que a equipa não considera o uso do BDD benéfico durante o processo de desenvolvimento de forma a validar o comportamento esperado do software.

6.6.2 Adoção da nova solução

Para medir este indicador, foram utilizados os seguintes métodos:

- Questão: Foi utilizada questão número 3 - Adotaste a solução desenhada de forma contínua durante o processo de desenvolvimento?
- Monitorização: Para obter métricas mais precisas, foi realizada uma monitorização ao longo do tempo de forma a conseguir contabilizar quais os desenvolvimentos (*stories*) ou correções de código (*bugs*) realizados com o uso da nova solução. Para isso, foi criada uma etiqueta que simboliza o uso da nova solução, e o responsável pela tarefa marcava-a com essa etiqueta. O período temporal utilizado foi o **Período Estudo de Caso**.

Questão 3 Para avaliar esta questão, irá ser utilizado um teste *Shapiro-Wilk* para verificar o tipo de distribuição.

*H*0: A população segue uma distribuição normal
*H*1: A população não segue uma distribuição normal

O código presente em 6.5 apresenta o teste de *Shapiro-Wilks* realizado com a amostra de respostas relativas à questão 3.

```
> q1<-c(4,4,5,4,5,5,5)
> shapiro.test(q1)

  Shapiro-Wilk normality test

data:  q1
W = 0.66444, p-value = 0.001497
```

Listing 6.5: Teste Shapiro-Wilks - Questão 3

De acordo com 6.5, com intervalo de confiança de 95%, podemos concluir que como o *p-value* é inferior a 5%, a amostra não segue uma distribuição normal, logo um *Wilcoxon* teste irá ser usado.

De acordo com a definição de um grau de satisfação superior a 80% para esta questão, podemos considerar as seguintes hipóteses:

*H*0: Média de respostas é menor ou igual a 4
*H*1: Média de respostas é maior que 4

```
> wilcox.test(q1, mu = 4, alternative = "greater")

  Wilcoxon signed rank test with continuity correction

data:  q1
V = 10, p-value = 0.03593
alternative hypothesis: true location is greater than 4
```

Listing 6.6: T Test - Questão 3

Após analisar o resultado de 6.6, como o *p-value* é inferior a 0.05, a hipótese nula pode ser rejeitada, logo com 95% de confiança podemos concluir que a equipa adotou de forma contínua a solução desenhada durante o processo de desenvolvimento.

Tabela 6.1: Monitorização do uso da nova solução

Período	**Estudo de caso**	**Modelo Tradicional**
Tarefas	70	13
Percentagem de utilização	84%	16%

Tabela 6.2: Velocidade da equipa

Período	**Estudo de caso**	**Modelo Tradicional**
Sprints	11	11
Story Points	382	353

Monitorização do uso da nova solução Para medir este indicador, foi utilizado o período temporal Estudo de Caso.

Foi realizada uma pesquisa a todas as tarefas realizadas pela equipa dentro desse período temporal e a amostra total de tarefas foi de 83. Após aplicar filtros sobre a amostra, nomeadamente o valor da etiqueta que reflete o uso da nova solução, foram retirados os números correspondeste ao uso da nova solução e também ao uso do processo tradicional. A tabela 6.1 apresenta com mais detalhe os números da amostra.

De acordo com a tabela 6.1, podemos concluir que cerca de 84% das tarefas foram marcadas com a etiqueta que refere a solução que este estudo de caso propõe, sendo este valor superior as 80% defendidos pela hipótese, podemos considerar que a solução foi utilizada e aplicada dentro do esperado.

6.6.3 Produtividade da equipa

Uma das dimensões a avaliar é a produtividade da equipa, ou seja, analisar o impacto da nova solução relativamente à capacidade de entrega da equipa. De acordo com a maioria dos estudos de caso semelhantes a este realizados industrialmente 2.7.1, seria expectável que numa fase inicial a produtividade diminuísse mas que, ao longo do tempo, aumentasse.

Para medir este indicador utilizou-se os dois períodos temporais referidos inicialmente, Período Estudo de caso e Período Modelo Tradicional. A tabela 6.2 apresenta o total de *Story Points* entregues durante estes dois períodos temporais.

Como podemos observar na tabela 6.2, o número de *Story Points* aumentou com o uso da nova solução, ou seja, a velocidade da equipa aumentou cerca de 8%. Não foi possível determinar se o uso da nova solução teve impacto na velocidade da equipa em uma fase inicial. Posto isto, podemos concluir que o uso da nova solução aumentou a velocidade de entrega.

6.6.4 Defeitos no código

O número de defeitos no código em produção é uma importante dimensão a ter em consideração, visto que a nova solução visa diminuir este problema. É expectável que o número de defeitos diminua com o uso da nova solução. Para medir este indicador, foi utilizado os mesmos período temporais, Período Estudo de Caso e Período Modelo Tradicional, com

Tabela 6.3: Número de defeitos no código

Período	**Estudo de caso**	**Modelo Tradicional**
Número de defeitos	1	17

Tabela 6.4: Cobertura do código por aplicação

Aplicação	**Percentagem de cobertura**
Aplicação A	84.74%
Aplicação B	34.14%
Aplicação C	41.75%
Aplicação D	82.06%
Média	61.44%

base nos projetos que a equipa tem na sua posse. A tabela 6.3 apresenta o número de defeitos no código relativos aos dois períodos temporais.

Analisando os dados presentes na tabela 6.3, o número de defeitos no código em produção diminuiu cerca de 94% com o uso da nova solução, conclui-se que a nova metodologia aplicada neste estudo de caso foi uma autêntica mais valia a diminuir o número de defeitos no código em produção.

6.6.5 Cobertura do código

Com a introdução e utilização da nova solução, orientada ao *Test-first*, o expectável é o aumento da cobertura do código. Para medir este indicador, o método de *Code Coverage* utilizado foi o *Statement coverage* (cobertura sobre instrução) 2.3.1.

A tabela 6.4 apresenta a percentagem de cobertura de código, utilizando o método *Statement coverage*, das aplicações pertencentes à equipa, antes da introdução da nova solução.

A tabela 6.5 apresenta a percentagem de código coberto pela aplicação que adotou a nova solução.

Após analisar as duas tabelas 6.4 e 6.5, relativamente à cobertura média de código das aplicações antes da introdução da nova solução, houve um aumento de cerca de 28% de cobertura de código comparativamente à aplicação que adotou a nova solução.

O hipótese delineada para este indicador era um valor de cobertura de código entre 80% e 90%, sendo que podemos concluir que o valor de 89.91% satisfaz a nossa condição. A nova solução contribui para um aumento da cobertura de código.

Tabela 6.5: Cobertura do código - Aplicação estudo de caso

Período	**Percentagem de cobertura**
Estudo de caso	89,91%

Tabela 6.6: Tempo de execução da bateria de testes (segundos)

Pipeline	**Máquina local**
88	43
93	42
91	44
80	41
88	42
82	43
83	43
80	42
93	42
86	39
99	38
83	38
89	30
89	41
91	34
86	33
82	33
99	34
86	34
87	35
Média = 87.75	Média = 38.55

6.6.6 Tempo de execução da bateria de testes

Sendo um dos pontos chave desta solução a utilização de testes, o tempo de execução da bateria de testes tem um grande impacto no desenvolvimento, *feedback* contínuo e na velocidade de entrega de valor. Para garantir que o tempo total de execução não ultrapassa-se os 100 segundos, foi realizada uma monitorização da execução da bateria de testes nas maquinas dos membros da equipa e nas maquinas onde são executadas as *pipelines*, e para além disso, foi utilizada uma questão, de modo a perceber se os membros da equipa consideram a bateria de testes rápida, escalável, eficiente e eficaz a garantir a qualidade da solução.

Monitorização da execução da bateria de testes As características das maquinas locais são iguais, sendo compostas pelo sistema operativo *Windows*, processador *Intel Core i7-7820HQ 2.90GHz*, memória RAM de 16GB e o disco é um *SSD*. As maquinas onde a *pipeline* é executada são compostas pelo sistema operativo *Linux* e as suas especificações gerais são *Standard DS13 v2 (8 vcpus, 56 GiB memory)*.

A tabela 6.6 apresenta uma lista de tempos de execução em segundos da bateria de testes nos dois ambientes diferentes, maquinas locais e *pipeline*.

Pipeline Para avaliar estes valores, irá ser utilizado um teste *Shapiro-Wilk* para verificar o tipo de distribuição para o tempo de execução da *pipeline*.

*H*0: A população segue uma distribuição normal
*H*1: A população não segue uma distribuição normal

```
> q1<-c(88,93,91,80,88,82,83,80,93,86,99,83,89,89,91,86,82,99,86,87)
> shapiro.test(q1)

  Shapiro-Wilk normality test

data:  q1
W = 0.94179, p-value = 0.2591
```

Listing 6.7: Teste Shapiro-Wilks - Tempo de execução da *pipeline*

De acordo com 6.7, com intervalo de confiança de 95%, podemos concluir que como o *p-value* é superior a 5% a amostra segue uma distribuição normal, logo um *t-test* pode ser usado.

De acordo com a definição tempo máximo de execução de 100 segundos, podemos considerar as seguintes hipóteses:

*H*0: Média de tempo de execução é igual ou superior a 100 segundos
*H*1: Média de tempo de execução é inferior a 100 segundos

```
> t.test(q1, mu= 100, alternative = "less", conf.level=0.95)

  One Sample t-test

data:  q1
t = -9.9693, df = 19, p-value = 2.765e-09
alternative hypothesis: true mean is less than 100
95 percent confidence interval:
     -Inf 89.8747
sample estimates:
mean of x
    87.75
```

Listing 6.8: T Test - Tempo de execução da *pipeline*

Analisando os resultados presentes em 6.8, como o *p-value* é inferior inferior a 5%, a hipótese nula é rejeitada, logo podemos concluir com 95% de confiança que o tempo de execução da *pipeline* é inferior a 100 segundos.

Máquina local O mesmo método irá ser utilizado, um teste *Shapiro-Wilk* para verificar o tipo de distribuição para o tempo de execução nas maquinas locais.

*H*0: A população segue uma distribuição normal
*H*1: A população não segue uma distribuição normal

```
> q1<-c(43,42,44,41,42,43,43,42,42,39,38,38,30,41,34,33,33,34,34,35)
> shapiro.test(q1)

  Shapiro-Wilk normality test

data:  q1
W = 0.88708, p-value = 0.02377
```

Listing 6.9: Teste Shapiro-Wilks - Tempo de execução nas maquinas locais

De acordo com 6.9, com intervalo de confiança de 95%, podemos concluir que como o *p-value* é inferior a 5% a amostra não segue uma distribuição normal, logo um *Wilcoxon* teste terá de ser usado.

De acordo com a definição tempo máximo de execução de 100 segundos, podemos considerar as seguintes hipóteses:

*H*0: Média de tempo de execução é igual ou superior a 100 segundos
*H*1: Média de tempo de execução é inferior a 100 segundos

```
> wilcox.test(q1, mu = 100, alternative = "less")

  Wilcoxon signed rank test with continuity correction

data:  q1
V = 0, p-value = 4.645e-05
alternative hypothesis: true location is less than 100
```

Listing 6.10: T Test - Tempo de execução nas maquinas locais

Analisando os resultados presentes em 6.10, como o *p-value* é inferior inferior a 5%, a hipótese nula é rejeitada, logo podemos concluir com 95% de confiança que o tempo de execução nas máquinas locais é inferior a 100 segundos.

Por último, podemos concluir que em ambos os ambientes, o tempo de execução da bateria de testes é inferior a 100 segundos.

Questão 6 Consideras a bateria de testes rápida, escalável, eficiente e eficaz a garantir a qualidade da solução?

Para avaliar esta questão, irá ser utilizado um teste *Shapiro-Wilk* para verificar o tipo de distribuição.

*H*0: A população segue uma distribuição normal
*H*1: A população não segue uma distribuição normal

O código presente em 6.11 apresenta o teste de *Shapiro-Wilks* realizado com a amostra de respostas relativas à questão 6.

```
> q1<-c(4,4,5,5,4,5,5)
> shapiro.test(q1)

  Shapiro-Wilk normality test

data:  q1
W = 0.66444, p-value = 0.001497
```

Listing 6.11: Teste Shapiro-Wilks - Questão 6

De acordo com 6.5, com intervalo de confiança de 95%, podemos concluir que como o *p-value* é inferior a 5% (nível de significância), a amostra não segue uma distribuição normal, sendo que irá ser aplicado um *Wilcoxon* teste.

De acordo com a definição de um grau de satisfação superior a 80% para esta questão, podemos considerar as seguintes hipóteses:

*H*0: Média de respostas é menor ou igual a 4
*H*1: Média de respostas é maior que 4

```
> wilcox.test(q1, mu = 4, alternative = "greater")

  Wilcoxon signed rank test with continuity correction

data:  q1
V = 10, p-value = 0.03593
alternative hypothesis: true location is greater than 4
```

Listing 6.12: Teste Wilcoxon - Questão 6

Após analisar o cálculo 6.12,como o *p-value* é inferior a 0.05, a hipótese nula é rejeitada, logo com 95% de confiança podemos concluir que a equipa considera que a bateria de testes é rápida, escalável, eficiente e eficaz a garantir a qualidade da solução.

Capítulo 7

Conclusão

7.1 Introdução

O presente capítulo tem como propósito apresentar as conclusões finais de todo o trabalho desenvolvido ao longo deste estudo de caso, os resultados obtidos, e por fim, o trabalho futuro.

7.2 Resultados

O principal objetivo deste estudo de caso foi desenhar, desenvolver, implementar e monitorizar um novo processo de desenvolvimento e aplica-lo sobre a equipa de desenvolvimento, de forma a responder à hipótese central desta dissertação, *Será que a aplicação do desenvolvimento orientado aos testes, em contexto empresarial, permitirá melhorar a qualidade dos produtos e do processo de desenvolvimento de software?*.

Após uma análise ao capítulo 6, podemos concluir que:

- Com um grau de satisfação superior a 90%, a equipa considerou que a solução desenhada se enquadra no âmbito da equipa, foi implementada com sucesso e adotada de forma contínua ao longo do tempo, com 84% das tarefas a serem marcadas com o selo *TDD&BDD*. Com isto, o processo de desenvolvimento ficou mais robusto, pois a definição dos critérios e testes de aceitação foram definidos em conjunto por toda a equipa, garantindo que todos os membros estão na mesma linha de pensamento.
- A solução contribuiu para uma aumento da velocidade da equipa, cerca de 8% face ao processo tradicional. Com este aumento, a equipa foi capaz de entregar mais valor para os seus *stakeholders*.
- O número de defeitos no código diminuiu drasticamente com o uso da nova solução, onde podemos evidenciar uma diminuição de cerca de 94%. Concluímos assim que a nova solução aplicada foi um enorme sucesso no que diz respeito à garantia da qualidade do produto entregue, apresentado um grau de defeitos muito baixo. Isto permite a que a equipa fique focada em novos desenvolvimentos e não perca tempo a analisar e realizar correções sobre aplicações em produção, que implicam repetir todo o fluxo de desenvolvimento, teste, aprovação e *release*.

- Aumento significativo da cobertura de código, cerca de 29% face às aplicações que seguiram o processo tradicional. O *Test-first*, nomeadamente o *TDD* e *BDD* contribuíram e guiaram os membros da equipa para a escrita de testes que tornam a aplicação mais forte e robusta.
- Desenho e implementação de uma bateria de testes rápida e eficaz a executar todo o tipo de testes, de unidade, integração e aceitação, com um tempo total abaixo dos 100 segundos, contribuindo para um *feedback* rápido a todas as vertentes, permitindo ser rápido também a executar na *pipeline*, não impactando o processo de *release*. O grande ganho foi diminuir os tempos de execução dos testes de integração e aceitação, que são testes mais custosos, mas que teriam de participar no *feedback* contínuo que os membros da equipa precisam de ver ao longo do processo da escrita de *software*, seguindo o fluxo de *Red, Green, Refactor*. Por último, o facto de todos os testes poderem ser executados dentro do ambiente de desenvolvimento (IDE, Visual Studio) e da própria solução, torna-se uma enorme mais valia, pois o *feedback* da bateria de testes responde imediatamente após às alterações realizadas sobre o *software* de forma automática.
- A confiança no *software* produzido aumentou exponencialmente, sendo que a bateria de testes, responsável por cobrir todos os cenários que a aplicação deve realizar, transmite a solidez e confiança necessária para que um processo de *release* seja feito a qualquer momento (mesmo a uma sexta-feira às 6 horas da tarde) sem que exista receio no que poderá acontecer em ambientes de produção.
- O facto do *TDD* guiar o desenvolvimento, com o desenho do teste feito inicialmente, as decisões de *design* são tomadas e o comportamento esperado é definido. Quando, por algum motivo, a realização do teste fica impossibilitada ou é difícil de concretizar, estamos perante um sinal de que talvez tenhamos de repensar o nosso *design*, e este exercício ajuda no desenho contínuo, fazendo com que os componentes que compõe a aplicação sejam segregados de uma forma mais clara, contribuindo para a reutilização de código e segregação de responsabilidades, ou seja, ajuda a criar código mais forte e robusto.
- Relativamente ao uso do BDD, a equipa, com um grau de confiança de 90%, não o considerar uma mais valia. Os motivos para este resultado estão associados aos desafios que a implementação do BDD provocou 5.6, podendo estar relacionado com o facto de a aplicação desenvolvida ser uma API, e não ser o contexto usual para o BDD, criando uma curva de aprendizagem maior do que o esperado pela equipa. No entanto, interpretando resultado de outra forma, a média de respostas é de 91%, ou seja, estamos perante um indicador de que possivelmente a equipa com um pouco mais de maturidade na utilização do BDD, o poderá considerar uma mais valia.
- Pelo facto das mais valias apresentadas, a confiança no processo de *merge* manual aumentou, bem como o processo de *refactor*, pois toda a solução apresenta uma bateria de testes que ajuda a cobrir os cenários e também se apresenta com um bom *design*, facilitando estes dois processos.
- A divida técnica criada foi praticamente nula, ou seja, o uso da nova solução proporciona a que este factor não aconteça com regularidade.
- Regressões manuais diminuíram drasticamente, devido ao facto do aumento da confiança no *software* e nos testes de aceitação. Assim sendo, a equipa ficou mais disponível

para novos desenvolvimentos ao invés de dedicar tempo a regressões, entregando mais valor.

- O tempo gasto no processo de *debug* diminuiu bastante devido a um motivo principal, o ciclo iterativo do TDD. Com *feedback* constantemente, não é necessário recorrer ao *debug* para perceber o problema.
- A confiança no processo de *refactor* também aumentou e foi mais frequente, existindo confiança de que o comportamento iria ser mantido, pois a bateria de testes é suficientemente eficaz a garantir o comportamento esperado.

Concluindo, a técnica *Test-First* foi muito bem aceite, o uso do TDD foi um enorme sucesso, sendo mesmo considerado uma mais valia por unanimidade. A adoção do BDD foi também muito vantajoso, apenas terá este conceito de ser amadurecido. Por fim, a resposta à hipótese central da dissertação é positiva, o desenvolvimento aplicado aos testes melhora a qualidade dos produtos e melhora o processo de desenvolvimento de *software*.

7.3 Trabalho futuro

Como trabalho futuro, para além de continuar a melhorar o processo de desenvolvimento e melhorar a aplicação de todas as técnicas e temas introduzidos ao longo desta dissertação, será interessante abordar os dois grandes tópicos que apesar de serem contemplados no desenho da solução, não foram implementados, sendo eles:

- Testes de mutação não foram aplicados durante o novo processo de desenvolvimento, pois após a sua introdução, a empresa considerou que este conceito não se adaptava ao processo pelo facto de não considera uma mais valia, sendo que a equipa apresenta um grau de maturidade elevado, a equipa contém um elemento dedicado aos testes funcionais, e por último, prever um elevado gasto de tempo a manter os testes de mutação. Contudo, será interessante aplicar os testes de mutação em projetos em que estas condições não se apliquem, pois será um componente chave no que diz respeito à validação da qualidade da bateria de testes, que consequentemente irá evidenciar, consolidar e afirmar a sua solidez.
- Monitorização e medição da qualidade do código acabou por não acontecer pelo facto de que não houve tempo dentro da realização deste estudo de caso, contudo, será um tópico muito interessante de realizar no futuro para perceber se de facto o desenvolvimento orientado aos testes contribui para um código melhor, bem desenhado e robusto.

Contudo, apesar destes tópicos não serem aplicados, podemos retirar conclusões com base na avaliação feita a alguns indicadores e concluir que: apesar de não existirem testes de mutação, a bateria de testes desenhada apresenta-se robusta e eficaz, e um dos indicadores que pode comprovar isso é a diminuição do número de defeitos do código. Relativamente à qualidade do código, a confiança que a equipa apresenta no *software* produzido, a velocidade de entrega e a diminuição de defeitos podem ajudar a evidenciar a qualidade geral do código, contudo, uma ferramenta que possa comprovar a premissa servirá para poder ser feita uma conclusão mais precisa.

Concluindo, apesar de todo o trabalho futuro já evidenciado, será feita a divulgação deste estudo de caso para que possa servir de exemplo para outras equipas, de forma a puderem

usar esta solução, ou parte dela, para melhorar os seus processos, produtos, velocidade de entrega, qualidade, etc.

7.4 Feedback adicional

Feedback adicional dos membros da equipa sobre o estudo de caso aplicado.

1. *A confiança da equipa no software desenvolvido aumentou consideravelmente e o refactor no projeto tornou-se muito mais distenso, pois existiam testes que garantiam que o comportamento esperado é mantido.*
2. *Acho que esta iniciativa trouxe muitos ganhos à nossa equipa. Os esclarecimentos sobre técnicas usadas foram muito úteis.*
3. *O presente caso de estudo trouxe, no meu entender, melhorias principalmente em 3 pontos:*
   (a) *Melhor qualidade das soluções desenvolvidas. A taxa de bugs tem diminuído consideravelmente;*
   (b) *Processo de desenvolvimento mais robusto, principalmente pela participação da equipa no desenho e na definição dos testes de aceitação;*
   (c) *Por último e consequentemente, a velocidade da equipa em tarefas é maior;*
4. *Para além de conseguir fazer release com mais confiança porque temos praticamente todos os cenários de teste cobertos, temos um CodeCoverage altíssimo desde o inicio das implementações. Desta forma evitamos ter dívida técnica e ter que recuar a implementações antigas que acabam por desgastar as pessoas. O uso do TDD, no meu ponto de vista, defende o desenvolvimento do software e defende o interesse da globalidade dos programadores, nunca voltar a mexer em código muito antigo.*
5. *Considero a adoção do TDD uma mais valia, tanto para para os elementos da equipa como para o projeto em questão do estudo de caso, já que proporciona bastante confiança à equipa de que a solução responde ao que é pedido de forma robusta. De forma pessoal, a mudança na forma de pensar para Test-First não foi imediata, e ainda é perdido algum tempo a considerar como desenhar o teste antes do código, mas acaba por ser positivo já que no final há confiança de que o código terá mais qualidade/robustez. Ainda não passou muito tempo desde a adoção deste paradigma mas a médio/longo prazo parece uma mais valia.*
6. *A confiança que temos nas implementações usando TDD é muito boa. Sempre tive intenção de começar a trabalhar desta forma. Só passado algum tempo é que começa a ser automático e também "indispensável". Muito obrigado pelos incentivos, sem duvida que é uma grande mais valia para a qualidade e segurança no software que produzimos.*

Bibliografia

Acceptance Test Driven Development (ATDD) (2018). url: https://www.agilealliance.org/glossary/atdd.

Adams, Tom (2007). «Better testing through behaviour». Em: *Open Source Developers' Conference*. Citeseer.

Alexandrova, Julia (2018). *Continuous Integration with TeamCity - TeamCity 2018.x Documentation*. url: https://confluence.jetbrains.com/display/TCD18/Continuous%20Integration%20with%20TeamCity.

Allen, I Elaine e Christopher A Seaman (2007). «Likert scales and data analyses». Em: *Quality progress* 40.7, pp. 64–65.

Anderson, Charles (2015). «Docker [software engineering]». Em: *IEEE Software* 32.3, pp. 102–c3.

Andersson, Johan, Geoff Bache e Peter Sutton (2003). «XP with Acceptance-Test Driven Development: A rewrite project for a resource optimization system». Em: *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, pp. 180–188.

Arachchi, SAIBS e Indika Perera (2018). «Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management». Em: *2018 Moratuwa Engineering Research Conference (MERCon)*. IEEE, pp. 156–161.

Beck, Kent (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Beck, Kent e Martin Fowler (2001). *Planning extreme programming*. Addison-Wesley Professional.

Beedle, Mike et al. (1999). «SCRUM: An extension pattern language for hyperproductive software development». Em: *Pattern languages of program design* 4, pp. 637–651.

Binder, Robert (2000). *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional.

Bissi, Wilson, Adolfo Gustavo Serra Seca Neto e Maria Claudia Figueiredo Pereira Emer (2016). «The effects of test driven development on internal quality, external quality and productivity: A systematic review». Em: *Information and Software Technology* 74, pp. 45–54.

Blog, Visual Studio (2017). *Live Unit Testing in Visual Studio 2017 Enterprise*. url: https://blogs.msdn.microsoft.com/visualstudio/2017/03/09/live-unit-testing-in-visual-studio-2017-enterprise/.

Bloom, James D (2018). *Mock Server*. url: http://www.mock-server.com/.

Boone, Harry N e Deborah A Boone (2012). «Analyzing likert data». Em: *Journal of extension* 50.2, pp. 1–5.

Burr, Kevin e William Young (1998). «Combinatorial test techniques: Table-based automation, test generation and code coverage». Em: *Proc. of the Intl. Conf. on Software Testing Analysis & Review*. Citeseer.

Butt, Farrukh Latif, Shahid Nazir Bhatti e Sohail Sarwar (2017). «Optimized order of software testing techniques in agile process–A systematic approach». Em:

Carvalho, Rogerio Atem de, Rodrigo Soares Manhães et al. (2010). «Filling the Gap between Business Process Modeling and Behavior Driven Development». Em: *arXiv preprint arXiv:1005.4975*.

Causevic, Adnan, Daniel Sundmark e Sasikumar Punnekkat (2011). «Factors limiting industrial adoption of test driven development: A systematic review». Em: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, pp. 337–346.

Cheon, Yoonsik e Gary T Leavens (2002). «A simple and practical approach to unit testing: The JML and JUnit way». Em: *European Conference on Object-Oriented Programming*. Springer, pp. 231–255.

De Carvalho, Rogério Atem e Renato De Campos (2006). «A development process proposal for the ERP5 system». Em: *Systems, Man and Cybernetics, 2006. SMC'06. IEEE International Conference on*. Vol. 6. IEEE, pp. 4703–4708.

De Lucia, Andrea e Abdallah Qusef (2010). «Requirements engineering in agile software development». Em: *Journal of emerging technologies in web intelligence* 2.3, pp. 212–220.

DeMillo, Richard A, Richard J Lipton e Frederick G Sayward (1978). «Hints on test data selection: Help for the practicing programmer». Em: *Computer* 11.4, pp. 34–41.

Docker overview (2019). url: `https://docs.docker.com/engine/docker-overview/`.

Elbaum, Sebastian, David Gable e Gregg Rothermel (2001). «The impact of software evolution on code coverage information». Em: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society, p. 170.

Fallah, Farzan, Srinivas Devadas e Kurt Keutzer (2001). «OCCOM-efficient computation of observability-based code coverage metrics for functional verification». Em: *IEEE Transactions on computer-aided design of integrated circuits and systems* 20.8, pp. 1003–1015.

Ficco, Massimo, Roberto Pietrantuono e Stefano Russo (2011). «Bug localization in test-driven development». Em: *Advances in Software Engineering* 2011, p. 2.

FluentAssertions (2019). *FluentAssertions*. url: `https://fluentassertions.com` (acedido em 30/06/2019).

Focus, Micro (2016). *Continuous Delivery: Automating the Deployment Pipeline*. url: `https://www.microfocus.com/media/white-paper/continuous_delivery_automating_the_deployment_pipeline_wp.pdf`.

Foundation, .NET. *About xUnit.net*. url: `https://xunit.github.io/`.

Fowler, Martin (2014). *UnitTest*. url: `https://martinfowler.com/bliki/UnitTest.html`.

Fowler, Martin e Matthew Foemmel (2006). «Continuous integration». Em: *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122, p. 14.

George, Boby e Laurie Williams (2003). «An initial investigation of test driven development in industry». Em: *Proceedings of the 2003 ACM symposium on Applied computing*. ACM, pp. 1135–1139.

Gliem, Joseph A e Rosemary R Gliem (2003). «Calculating, interpreting, and reporting Cronbach's alpha reliability coefficient for Likert-type scales». Em: Midwest Research-to-Practice Conference in Adult, Continuing, e Community ...

Gojare, Satish, Rahul Joshi e Dhanashree Gaigaware (2015). «Analysis and Design of Selenium WebDriver Automation Testing Framework». Em: *Procedia Computer Science* 50, pp. 341–346.

Guidelines for Test-Driven Development (2006). url: `https://msdn.microsoft.com/en-%20us/library/aa730844(v=vs.80).aspx`.

Gundecha, Unmesh (2012). *Selenium Testing Tools Cookbook*. Packt Publishing Ltd.

Hembrink, J e P Stenberg (2013). «Continuous integration with jenkins». Em: *Coaching of Programming Teams (EDA 270), Faculty of Engineering, Lund University, LTH*, p. 23.

Hendrickson, Elisabeth (2008). «Driving development with tests: ATDD and TDD». Em: *STARWest 2008*.

Hetzel, William C e Bill Hetzel (1988). *The complete guide to software testing*. QED Information Sciences Wellesley, MA.

Holmes, Antawan e Marc Kellogg (2006). «Automating functional tests using selenium». Em: *Agile Conference, 2006*. IEEE, 6–pp.

Hsia, Pei et al. (1994). «Behavior-based acceptance testing of software systems: a formal scenario approach». Em: *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International*. IEEE, pp. 293–298.

Introducing BDD (2006). url: `https://dannorth.net/introducing-bdd/`.

Janzen, David S (2005). «Software architecture improvement through test-driven development». Em: *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, p. 240.

Janzen, David S e Hossein Saiedian (2006). «On the influence of test-driven development on software design». Em: *Software Engineering Education and Training, 2006. Proceedings. 19th Conference on*. IEEE, pp. 141–148.

– (2008). «Does test-driven development really improve software design quality?» Em: *Ieee Software* 25.2, p. 77.

Jeffries, Ron, Ann Anderson e Chet Hendrickson (2001). *Extreme programming installed*. Addison-Wesley Professional.

Jeffries, Ron e Grigori Melnik (2007). «TDD: The Art of Fearless Programming». Em: *IEEE SOFTWARE* 2, p. 5.

Jenkins (2018). url: `http://jenkins.io/doc/`.

Jia, Yue e Mark Harman (2011). «An analysis and survey of the development of mutation testing». Em: *IEEE transactions on software engineering* 37.5, pp. 649–678.

Kaur, Rupinder e Jyotsna Sengupta (2013). «Software process models and analysis on failure of software development projects». Em: *arXiv preprint arXiv:1306.1068*.

Kim, Taesoo, Ramesh Chandra e Nickolai Zeldovich (2013). «Optimizing unit test execution in large software programs using dependency analysis». Em: *Proceedings of the 4th Asia-Pacific Workshop on Systems*. ACM, p. 19.

Kock, Ned (2015). «One-tailed or two-tailed P values in PLS-SEM?» Em: *International Journal of e-Collaboration (IJeC)* 11.2, pp. 1–7.

Koen, Peter et al. (2001). «Providing clarity and a common language to the "fuzzy front end"». Em: *Research-Technology Management* 44.2, pp. 46–55.

Kumar, Shaweta e Sanjeev Bansal (2013). «Comparative study of test driven development with traditional techniques». Em: *Int J Soft Comput Eng (IJSCE)* 3.1, pp. 2231–2307.

Layman, Lucas et al. (2006). «Essential communication practices for Extreme Programming in a global software development team». Em: *Information and software technology* 48.9, pp. 781–794.

Leung, Hareton KN e Lee White (1990). «A study of integration testing and software regression at the integration level». Em: *Software Maintenance, 1990, Proceedings., Conference on*. IEEE, pp. 290–301.

Lindstrom, Lowell e Ron Jeffries (2004). «Extreme programming and agile software development methodologies». Em: *Information systems management* 21.3, pp. 41–52.

Luo, Lu (2001). «Software testing techniques». Em: *Institute for software research international Carnegie mellon university Pittsburgh, PA* 15232.1-19, p. 19.

Mackinnon, Tim, Steve Freeman e Philip Craig (2000). «Endo-testing: unit testing with mock objects». Em: *Extreme programming examined*, pp. 287–301.

Marick, Brian et al. (1999). «How to misuse code coverage». Em: *Proceedings of the 16th Interational Conference on Testing Computer Software*, pp. 16–18.

Marri, Madhuri R et al. (2009). «An empirical study of testing file-system-dependent software with mock objects». Em: *Automation of Software Test, 2009. AST'09. ICSE Workshop on*. IEEE, pp. 149–153.

Martinez-Hernandez, Veronica (2003). «Understanding value creation: the value matrix and the value cube». Tese de doutoramento. University of Strathclyde.

Microsoft (2019a). *Create web APIs with ASP.NET Core*. url: `https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-2.2`.

– (2019b). *Integration tests in ASP.NET Core*. url: `https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-2.2`.

– (2019c). *Microsoft.AspNetCore.Mvc.Testing*. url: `https://www.nuget.org/packages/Microsoft.AspNetCore.Mvc.Testing` (acedido em 30/06/2019).

Minimal e Tom Akehurst (2019). *WireMock*. url: `http://wiremock.org/`.

Mongo2Go (2019a). *Mongo2Go*. url: `https://github.com/Mongo2Go/Mongo2Go`.

– (2019b). *Mongo2Go*. url: `https://github.com/Mongo2Go/Mongo2Go` (acedido em 30/06/2019).

Moq (2019). *Moq*. url: `https://github.com/moq/moq4` (acedido em 30/06/2019).

Nagappan, Nachiappan et al. (2008). «Realizing quality improvement through test driven development: results and experiences of four industrial teams». Em: *Empirical Software Engineering* 13.3, pp. 289–302.

Offutt, A Jefferson (1994). «A practical system for mutation testing: help for the common programmer». Em: *Test Conference, 1994. Proceedings., International*. IEEE, pp. 824–830.

Osterwalder, Alexander e Yves Pigneur (2010). *Business model generation: a handbook for visionaries, game changers, and challengers*. John Wiley & Sons.

– (2011). «Aligning profit and purpose through business model innovation». Em: *Responsible management practices for the 21st century*, pp. 61–75.

Pan, Jiantao (1999). «Software testing». Em: *Dependable Embedded Systems* 5, p. 2006.

Pereira, Lauriane et al. (2018). «Behavior-driven development benefits and challenges: reports from an industrial study». Em: *Proceedings of the 19th International Conference on Agile Software Development: Companion*. ACM, p. 42.

Prouse, Rob et al. (2018). *Unit testing C with NUnit and .NET Core - .NET Core*. url: `https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit`.

Reese, John, Andy De George e Maira Wenzel (2018). *Best practices for writing unit tests - .NET Core*. url: `https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices`.

Rendell, Andrew (2008). «Effective and pragmatic test driven development». Em: *Agile 2008 Conference*. IEEE, pp. 298–303.

Rpetrusha (2017). *Live Unit Testing - Visual Studio*. url: `https://docs.microsoft.com/en-us/visualstudio/test/live-unit-testing?view=vs-2017`.

– (2019). *Live Unit Testing - Visual Studio*. url: `https://docs.microsoft.com/en-us/visualstudio/test/live-unit-testing?view=vs-2017`.

Ruxton, Graeme D e Markus Neuhäuser (2010). «When should we use one-tailed hypothesis testing?» Em: *Methods in Ecology and Evolution* 1.2, pp. 114–117.

Santelices, Raul et al. (2009). «Lightweight fault-localization using multiple coverage types». Em: *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, pp. 56–66.

Sauvé, Jacques Philippe, Osório Lopes Abath Neto e Walfredo Cirne (2006). «EasyAccept: a tool to easily create, run and drive development with automated acceptance tests». Em: *Proceedings of the 2006 international workshop on Automation of software test*. ACM, pp. 111–117.

Schwaber, Ken e Mike Beedle (2002). *Agile software development with Scrum*. Vol. 1. Prentice Hall Upper Saddle River.

Shull, Forrest et al. (2010). «What do we know about test-driven development?» Em: *IEEE software* 27.6, pp. 16–19.

Solis, Carlos e Xiaofeng Wang (2011). «A study of the characteristics of behaviour driven development». Em: *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. IEEE, pp. 383–387.

Sommerville, Ian (1996). «Software process models». Em: *ACM computing surveys (CSUR)* 28.1, pp. 269–271.

Specflow (2019). *SpecFlow Documentation*. url: `https://specflow.org/documentation/faq/`.

Stolberg, Sean (2009). «Enabling agile testing through continuous integration». Em: *Agile Conference, 2009. AGILE'09*. IEEE, pp. 369–374.

Stone, David (2019). *Stats Tutorial - Instrumental Analysis and Calibration*. url: `https://www.chem.utoronto.ca/coursenotes/analsci/stats/12tailed.html`.

Thomas, Christopher M (2014). «An Overview of the Current State of the Test-First vs. Test-Last Debate». Em: *Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal* 1.2, p. 2.

Thomas, Dave e Andy Hunt (2002). «Mock objects». Em: *IEEE Software* 19.3, pp. 22–24.

Tsai, Wei-Tek et al. (2001). «End-to-end integration testing design». Em: *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*. IEEE, pp. 166–171.

Umesh, Nitin e Amar Saraswat (2015). «Automation Testing: An Introduction to Selenium». Em: *International Journal of Computer Applications* 119.3.

University, Newcastle (2018). *Hypothesis Testing*. url: `https://internal.ncl.ac.uk/ask/numeracy-maths-statistics/statistics/hypothesis-testing/one-tailed-and-two-tailed-tests.html`.

Value Analysis (2005). url: `http://www.innovation-portal.info/wp-content/uploads/Value-Analysis.pdf`.

Wappler, Stefan e Joachim Wegener (2006). «Evolutionary unit testing of object-oriented software using strongly-typed genetic programming». Em: *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, pp. 1925–1932.

WireMock-Net (2019). *WireMock-Net/WireMock.Net*. url: `https://github.com/WireMock-Net/WireMock.Net`.

Woodall, Tony (2003). «Conceptualising 'value for the customer': an attributional, structural and dispositional analysis». Em: *Academy of marketing science review* 12.1, pp. 1–42.

Wright, Joe (2009). *The Coding Dojo*. url: `https://code.joejag.com/2009/the-coding-dojo.html`.

Wynne, Matt, Aslak Hellesoy e Steve Tooke (2017). *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf.

Yurtoğlu, Nadir (2018). «http://www.historystudies.net/dergi//birinci-dunya-savasinda-bir-asayis-sorunu-sebinkarahisar-ermeni-isyani20181092a4a8f.pdf». Em: *History Studies International Journal of History* 10.7, pp. 241–264. doi: 10.9737/hist.2018.658.

Zhang, Lin (2012). «Software testing: A case study of a small Norwegian software team». Tese de mestrado. Universitetet i Agder; University of Agder.

Appendix A

Apêndice A

Inquérito de satisfação

Consideras que a solução desenhada se enquadra no âmbito da equipa? *

	1	2	3	4	5	
Discordo totalmente	○	○	○	○	○	Concordo totalmente

Consideras que a solução desenhada ajuda de forma contínua a resolver os problemas identificados? *

	1	2	3	4	5	
Discordo totalmente	○	○	○	○	○	Concordo totalmente

Adotaste a solução desenhada de forma contínua durante o processo de desenvolvimento? *

	1	2	3	4	5	
Discordo totalmente	○	○	○	○	○	Concordo totalmente

Consideras o uso do TDD uma mais valia durante o processo de desenvolvimento de forma a melhorar o software produzido? *

	1	2	3	4	5	
Discordo totalmente	○	○	○	○	○	Concordo totalmente

Consideras o uso do BDD benéfico durante o processo de desenvolvimento de forma a validar o comportamento esperado do software? *

	1	2	3	4	5	
Discordo totalmente	○	○	○	○	○	Concordo totalmente

Consideras a bateria de testes rápida, escalável, eficiente e eficáz a garantir a qualidade da solução? *

1 = Very disatisfied 5 = Very satisfied

	1	2	3	4	5	
Discordo totalmente	○	○	○	○	○	Concordo totalmente

Feedback adicional sobre o estudo de caso aplicado. (Keywords: TDD, BDD, Software-Development, Test-First, Software-Tests, Automation, Unit-Tests, Code-Quality, Software-Confidence, Scalability)

Long-answer text