UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS CORNÉLIO PROCÓPIO
DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ALTIERES DE MATOS

EFEITOS DO USO DO DESENVOLVIMENTO ORIENTADO POR TESTES EM CONJUNTO COM CRITÉRIOS DE TESTE NA INDÚSTRIA DE SOFTWARE ÁGIL

DISSERTAÇÃO – MESTRADO

CORNÉLIO PROCÓPIO
2020

ALTIERES DE MATOS

EFEITOS DO USO DO DESENVOLVIMENTO ORIENTADO POR TESTES EM CONJUNTO COM CRITÉRIOS DE TESTE NA INDÚSTRIA DE SOFTWARE ÁGIL

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do título de "Mestre em Informática".

Orientador: Prof. Dr. Marco Aurélio Graciotto Silva

Coorientador: Prof. Dr. Reginaldo Ré

CORNÉLIO PROCÓPIO
2020

TERMO DE LICENCIAMENTO

4.0 International

Esta licença permite que outros remixem, adaptem e criem partir do trabalho para fins não comerciais, desde que atribuam o devido crédito e que licenciem as novas criações sob termos idênticos.
Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Cornélio Procópio

ALTIERES DE MATOS

EFEITOS DO USO DO DESENVOLVIMENTO ORIENTADO POR TESTES EM CONJUNTO COM CRITÉRIOS DE TESTE NA INDÚSTRIA DE SOFTWARE ÁGIL

Trabalho de pesquisa de mestrado apresentado como requisito para obtenção do título de Mestre Em Informática da Universidade Tecnológica Federal do Paraná (UTFPR). Área de concentração: Computação Aplicada.

Data de aprovação: 13 de Novembro de 2020

Prof Marco Aurelio Graciotto Silva, - Universidade Tecnológica Federal do Paraná

Prof.a Katia Romero Felizardo Scannavino, Doutorado - Universidade Tecnológica Federal do Paraná

Prof Reginaldo Re, - Universidade Tecnológica Federal do Paraná

Prof Vinicius Humberto Serapilha Durelli, Doutorado - Universidade Federal de São João Del Rei (Ufsj)

Documento gerado pelo Sistema Acadêmico da UTFPR a partir dos dados da Ata de Defesa em 16/11/2020.

Dedico este trabalho a minha família.

Agradecimentos

Primeiramente a Deus que permitiu que tudo isso acontecesse, ao longo de minha vida, e não somente nestes anos como mestrando, mas que em todos os momentos é o maior mestre que alguém pode conhecer.

A esta universidade, seu corpo docente, direção e administração que oportunizaram a janela que hoje vislumbro um horizonte superior, eivado pela acendrada confiança no mérito e ética aqui presentes.

A todos os professores por me proporcionar o conhecimento não apenas racional, mas a manifestação do caráter e afetividade da educação no processo de formação profissional, por tanto que se dedicaram a mim, não somente por terem me ensinado, mas por terem me feito aprender. A palavra mestre nunca fará justiça aos professores dedicados aos quais sem nominar terão os meus eternos agradecimentos.

Aos meus pais, pelo amor, incentivo e apoio incondicional.

A minha mãe Geny Teixeira Verginio, heroína que me deu apoio, incentivo nas horas difíceis, de desânimo e cansaço.

Ao meu pai José Francisco de Matos, que apesar de todas as dificuldades me fortaleceu e que para mim foi muito importante.

A minha esposa Beatrice Ramos da Silva, pela paciência e apoio nos momentos em que eu mais precisei, seja durante a ausência devido a diversas viagens ou pelos momentos enquanto estive estudando.

Aos meus irmãos e sobrinhos, que nos momentos de minha ausência dedicados a este estudo, sempre fizeram entender que o futuro é feito a partir da constante dedicação no presente.

Ao meu orientador Marco Aurélio Graciotto Silva, pela paciência perante todo este período, pelo conhecimento compartilhado e por sempre acreditar em mim.

A todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

Resumo

de Matos, Altieres. **Efeitos do uso do desenvolvimento orientado por testes em conjunto com critérios de teste na indústria de software ágil**. 2020. 73 f. Dissertação – Mestrado, Programa de Pós-Graduação em Informática, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2020.

Contexto: O impacto da adoção de critérios de teste no desenvolvimento de software utilizando Desenvolvimento Orientado por Teste (TDD) não está claro. Não existem evidências suficientes para definir o impacto dessas novas atividades na produtividade e qualidade efetiva do software. Com o aumento do interesse de profissionais em automação de casos testes existem fatores que limitam a adoção do TDD na indústria de software. **Objetivo:** Propor a adoção do TDD em conjunto com critérios de teste, fornecendo uma ferramenta para análise da conformidade do processo do TDD, analisando os efeitos gerados na sua adoção e avaliando a existência do aumento da qualidade de produtos em indústrias de software inseridas no contexto ágil com modelo iterativo. **Método:** Neste estudo definiu-se um estudo de caso envolvendo profissionais de tecnologia da informação em uma indústria de software situada no noroeste do Paraná, com a finalidade de responder a questão de pesquisa: *Existem efeitos de melhoria no desenvolvimento de software durante a adoção de Desenvolvimento Orientado por Teste (TDD) e critérios de teste?* **Resultados:** Produzimos a ferramenta *Butterfly*, uma extensão para o IDE Eclipse, responsável por mensurar a conformidade do processo do TDD durante o ciclo do desenvolvimento de software. Com a execução do estudo de caso, alcançamos evidências quanto à adoção do TDD, melhoria da qualidade do produto de software e impacto no esforço de trabalho. Materializamos um pacote experimental derivado do estudo de caso realizado, provendo que outros pesquisadores possam realizar outros estudos em cenários similares, coletando novas evidências. **Conclusão:** Concluímos que existem indícios de efeitos de melhoria no desenvolvimento de software durante a adoção do TDD em conjunto com critérios, baseando-se nas evidências coletadas sobre a conformidade do processo de desenvolvimento e da qualidade do software desenvolvido. Em relação a conformidade do processo do TDD, tivemos redução no percentual da utilização do TDD, porém tivemos aumento do percentual da cobertura dos requisitos de teste funcionais. Em relação a qualidade do software tivemos a redução do número de *code smells* e defeitos, porém houve a redução do percentual de cobertura de linhas e de mutação.

Palavras-chaves: TDD. desenvolvimento orientado por testes. critérios de teste. métodos ágeis. ferramenta Butterfly

Abstract

de Matos, Altieres. Effects of using test-driven development with test criteria in the agile software industry. 2020. 73 f. Dissertation (Graduation Program in Informatics), Federal University of Technology – Paraná. Cornélio Procópio, PR, Brazil, 2020.

Context: The impact of adopting test criteria on software development using Test Driven Development (TDD) is unclear. There is insufficient evidence to define the impact of these new activities on the productivity and effective quality of the software. With the increase in the interest of professionals in automation of test cases, there are factors that limit the adoption of TDD in the software industry. **Goal:** Propose the adoption of TDD in conjunction with testing criteria, providing a tool for analyzing the compliance of the TDD process, analyzing the effects generated in its adoption and evaluating the existence of the increase in the quality of products in software industries inserted in the agile context with an iterative model. **Method:** In this study, a case study was defined involving information technology professionals in a software industry located in northwestern Paraná, in order to answer the research question: *There are improvement effects in the software development during the adoption of Test Driven Development (TDD) and test criteria?* **Outcomes:** We produced the *Butterfly* tool, an extension to the IDE Eclipse, responsible for measuring the compliance of the TDD process during the software development cycle. With the execution of the case study, we achieved evidence regarding the adoption of TDD, improving the quality of the software product and impacting the work effort. We materialized an experimental package derived from the case study carried out, providing that other researchers can carry out other studies in similar scenarios, collecting new evidence. **Conclusion:** We conclude that there are signs of improvement effects on software development during the adoption of TDD in conjunction with criteria, based on the evidence collected on the conformity of the development process and the quality of the software developed. Regarding the compliance of the TDD process, we had a reduction in the percentage of use of TDD, but we had an increase in the percentage of coverage of the functional test requirements. Regarding the quality of the software, we had a reduction in the number of code smells and defects, but there was a reduction in the percentage of line coverage and mutation.

Keywords: TDD. test driven development. test criteria. agile methods. Butterfly tool

Lista de figuras

Lista de tabelas

Lista de acrônimos

Sumário

Capítulo 1

Introdução

Emergindo no final dos anos 90 em resposta a métodos complexos de desenvolvimento de software, métodos ágeis oferecem processos mais leves e disciplinados. Eles empregam o esforço e a experiência no centro do desenvolvimento de software, por meio do seu foco central em pessoas e interações (HODA et al., 2018). A presença de métodos ágeis exige que a responsabilidade da qualidade do software faça-se além da equipe de qualidade (CAUSEVIC et al., 2013b). Logo, é necessário aplicar práticas ágeis relacionadas a qualidade do software, como o TDD (BECK, 1999; JANZEN; SAIEDIAN, 2005; CAUSEVIC et al., 2013a).

O uso do TDD está alinhado a importância de testes automatizados para que o desenvolvimento de software trabalhe de forma ideal (SPINELLIS, 2017). O TDD provê aos desenvolvedores a possibilidade de escrever pequenos pedaços de software baseado em requisitos de software e iterações curtas. Implementam-se casos de testes antes dos código fonte da aplicação e em seguida são realizadas melhorias do código fonte, sempre de forma iterativa. Esse estilo de desenvolvimento possibilita que desenvolvedores mantenham o foco nos conjuntos de requisitos, garantindo que cada trecho de código fonte da aplicação escrito seja coberto por um caso de teste e não mantenham código fonte desnecessário, garantindo maior qualidade do código fonte produzido (BECK, 1999; JANZEN; SAIEDIAN, 2005; CAUSEVIC et al., 2013a; SPINELLIS, 2017; FUCCI et al., 2017).

Com a introdução do TDD no *eXtreme Programming* (XP), o TDD obteve aumento de sua popularidade (BECK, 1999; CAUSEVIC et al., 2013a). XP é um método ágil que utiliza pequenas iterações na construção de software (JANZEN; SAIEDIAN, 2005), semelhantes às iterações do TDD (FUCCI et al., 2017). Além da utilização do TDD no método XP (JANZEN; SAIEDIAN, 2005), é possível sua utilização em conjunto com o método ágil Scrum (HAMMOND; UMPHRESS, 2012). Em contrapartida, por décadas softwares desenvolvidos não foram testados ou utilizaram-se de testes de forma manual (SPINELLIS, 2017). Com o crescimento do tamanho e complexidade do software surge a necessidade da automatização de teste (RAULAMO-JURVANEN et al., 2017).

Entretanto, o foco do TDD reside no *design* do software e não esta relacionado à técnicas de teste de software propriamente dito (Pachulski Camara; Graciotto Silva, 2016). Técnicas de verificação e teste de software necessitam de um conjunto de entradas e saídas, que podem ser aprimoradas com a utilização de critérios de teste em conjunto com testes automáticos (BRIAND et al., 2017). Não existe preocupação na criação de conjunto de casos de testes efetivos considerando técnicas e critérios de teste durante a utilização do TDD (Pachulski Camara; Graciotto Silva, 2016). Existem diversos desafios para criar suítes de testes robustas que apoiam desenvolvedores a identificarem defeitos (SHELTON et al., 2012). Escrever testes baseados em critérios de teste possibilitam deixar o caso de teste mais completo possível. Dessa forma, critérios de teste de software têm como objetivo diminuir a quantidade do domínio de entradas de um caso de teste, possibilitando o aumento de erros encontrados. Os critérios de teste são definidos a partir de técnicas de teste de software e são derivados dos requisitos de teste. É impossível criar testes completos, mesmo com a utilização de critérios de teste, porém é possível melhorar os casos de testes escritos (MYERS et al., 2013).

De outro lado, uma das discussões recentes na área de teste de software é a obrigatoriedade do teste de software no âmbito do desenvolvimento de software (BRIAND et al., 2017; RAULAMO-JURVANEN et al., 2017). Apesar disso, observa-se que a indústria e a acadêmia de software não estão fortemente ligadas e não possuem alta colaboração entre elas (BRIAND et al., 2017; GAROUSI et al., 2017). Outro fator motivante, é o aumento do interesse de profissionais em automação de casos testes (RAULAMO-JURVANEN et al., 2017). Em contrapartida ao aumento da procura por automação de testes (RAULAMO-JURVANEN et al., 2017), existem fatores que limitam a adoção do TDD na indústria de software (CAUSEVIC et al., 2011).

Segundo Briand et al. (2017) a Engenharia de Software (ES) não contempla forte relação entre pesquisadores e indústria de software. Um dos principais problemas é que pesquisadores não estão preocupados em resolver problemas das indústrias de software (GAROUSI et al., 2017). Várias razões tem sido discutidas por pesquisadores da área de ES, desde a diferença de objetivos entre as duas partes até desafios de escalabilidade e aplicabilidade dos problemas (GAROUSI et al., 2017).

Muitos estudos empíricos foram conduzidos para examinar os efeitos do TDD em diferentes atributos (TOSUN et al., 2018). Contudo, ainda não está claro o impacto da adoção de critérios de teste no desenvolvimento de software utilizando TDD (Pachulski Camara; Graciotto Silva, 2016; GHAFARI et al., 2020). Muitos pesquisadores não se preocupam com a evolução e melhoria da qualidade de casos de teste (CAUSEVIC et al., 2013b). Em contrapartida ao aumento da procura por automação de testes (RAULAMO-JURVANEN et al., 2017), existem fatores que limitam a adoção do TDD na indústria de software: (i) aumento no tempo de desenvolvimento, (ii) insuficiência de experiência/conhecimento com

TDD, (iii) insuficiência em design, (iv) insuficiência de habilidade de desenvolvimento de testes, (v) insuficiência na aderência do protocolo do TDD, (vi) limitação no domínio de negócio ou ferramentas, e (vii) código legado (CAUSEVIC et al., 2011).

Embora a indústria de desenvolvimento de software global e a comunidade de pesquisa acadêmica de software possuam grande quantidade de membros, a colaboração entre as duas é baixa (GAROUSI et al., 2017). Em 2017 o estado da arte de teste de software considerou que realizar testes manuais ou automáticos tornou-se obrigatório para a produção de produtos de software (BRIAND et al., 2017). Dessa forma, adotar TDD possibilita a automatização de testes de software. TDD provê a indústria possibilidade de que seus colaboradores deixem de realizar a escrita de testes automáticos após a escrita do código de produção (SPINELLIS, 2017).

Este estudo teve como objetivo propor a adoção do TDD em conjunto com critérios de teste em indústrias de software. Um aspecto fundamental do estudo é a aplicação na indústria de software no contexto ágil junto com a utilização do modelo iterativo. Este estudo também analisou a forma como a adoção de critérios de teste afetam o desenvolvimento de software utilizando TDD. Investigamos os efeitos relativos à conformidade do processo do TDD, redução do esforço de trabalho na construção de novas funcionalidades e aumento da qualidade de produtos de software baseado na diminuição da dívida técnica e defeitos. Baseado nos fatores que limitam a adoção do TDD, foi investigado o aumento no tempo de desenvolvimento, insuficiência na aderência do protocolo do TDD e insuficiência de habilidade de desenvolvimento de casos de testes, surge a seguinte questão de pesquisa:

- *Existem efeitos de melhoria no desenvolvimento de software durante a adoção do Desenvolvimento Orientado por Teste (TDD) com critérios de teste?*

Existem fatores que limitam a adoção do TDD na indústria de software (CAUSEVIC et al., 2011). Desta forma, com este estudo busca a redução de fatores que prejudicam a adoção de novas práticas. Ao diminuir a frequência de problemas encontrados, dizimar a criticidade dos problemas, minimizar o tempo para encontrar problemas e reduzir o esforço gerado pela correção de código fonte com problemas encontrados durante o ciclo de desenvolvimento (MATOS et al., 2018a), viabiliza-se maior qualidade do produto de software desenvolvido.

Este estudo contribui para esta área de pesquisa crescente por explorar a adoção de técnicas de teste de software durante o desenvolvimento de software, assim como no estudo de Pachulski Camara e Graciotto Silva (2016). Buscou-se, desta forma, o aumento da eficácia de casos de teste, resultando em redução de esforço para desenvolvimento de novas funcionalidades além do aumento da qualidade do produto de software desenvolvido em indústrias de software.

No Capítulo 2 relatam-se os principais conceitos sobre TDD e critérios de teste, enfatizando-se principalmente a sua utilização na indústria de software. De acordo com estes

conceitos, foi traçada uma visão geral a respeito dos trabalhos relacionados encontrados na literatura, de forma a abordar trabalhos que envolvam estudos empíricos na indústria de software e que busquem melhorar a eficiência de casos de teste durante o processo do TDD. No Capítulo 3 apresentamos o plano de trabalho, sendo descritas as principais atividades a serem realizadas e o método a ser seguido. No Capítulo 4 exibimos os resultados obtidos com este estudo, apresentando a ferramenta Butterfly, uma ferramenta criada para avaliação da conformidade do processo do TDD, que teve seu desenvolvimento necessário para coletar métricas levantadas durante a elaboração desta Dissertação de Mestrado. Além da ferramenta Butterfly, apresentamos os resultados do estudo estudo experimental executado com participantes de uma indústria de software situada no noroeste do Paraná. Por fim, no Capítulo 5 apresentamos as conclusões finais desta Dissertação de Mestrado.

Capítulo 2

Referencial Teórico

Neste capítulo são apresentados os principais tópicos deste estudo, como desenvolvimento de software ágil, TDD, teste de software e critério de teste.

2.1. Desenvolvimento Ágil de Software

O Desenvolvimento Ágil de Software (DAS) emergiu como resultado do amplo descontentamento de profissionais com a abordagem tradicional de desenvolvimento de software (ELBANNA; SARKER, 2016), com altas taxas de falhas associadas ao desenvolvimento de software junto com a necessidade de entrega rápida do software desenvolvido (ELBANNA; SARKER, 2016). Como qualquer outra abordagem, os primeiros anos do DAS foram marcados pela exuberância e pelo ceticismo de muitos (DINGSøYR et al., 2012), gerando enorme impacto em como o software é desenvolvido em todo o mundo (DYBA; DINGSOYR, 2009). O DAS é adotado por muitos profissionais, e muitas vezes, entusiastas, acabam exaltando como a abordagem *mais rápida, melhor e mais barata* (ELBANNA; SARKER, 2016).

Na Tabela 2.1 apresentamos um comparativo das perspectivas do desenvolvimento de software tradicional e ágil. De forma geral, a perspectiva tradicional desenhada para um ambiente menos caótico, em que processos são essências e a gestão cumpre apenas um papel controlador. Do outro lado, a perspectiva ágil, uma perspectiva para ambiente caóticos, em que o foco em pessoas se torna maior, possibilitando que evolução dos times com sua própria colaboração.

Em fevereiro de 2001, um grupo com 17 pessoas se reuniu para falar sobre o DAS com defensores do XP, *SCRUM*, *DSDM*, *Adaptive Software Development*, *Crystal*, *Feature-Driven Development*, *Pragmatic Programming* e outros que simpatizavam com a necessidade de uma alternativa aos processos de desenvolvimento de software tradicional e orientados por documentação. Dessa conversa, surgiu o Manifesto Ágil de *Desenvolvimento de Software* (BECK et al., 2001). O manifesto é composto por 4 itens essenciais e 12 princípios (DIEBOLD;

Tabela 2.1. Perspectivas tradicional e ágil no desenvolvimento de software.

Fonte: Dyba e Dingsoyr (2009).

	Perspectiva Tradicional	Perspectiva Ágil
Design do Processo	Deliberada e formal, sequência linear de etapas, formulação e implementação separadas, orientada por regras	Emergente, interativo e exploratório, conhecimento e ação inseparáveis, além de regras formais
Objetivo	Otimização	Adaptação, flexibilidade, capacidade de resposta
Processo de resolução de problemas	Seleção dos melhores meios para realizar um determinado fim através de atividades bem planejadas e formalizadas	Aprendizagem através da experimentação e introspecção, reenquadrando constantemente o problema e sua solução
Visão do ambiente	Estável, previsível	Turbulento, difícil de prever
Tipo de aprendizado	Malha simples / adaptativa	Malha dupla / generativa
Características chaves	Controle e Direção; Evita conflitos; Formaliza a inovação; Gerente é controlador; O design precede a implementação	Colaboração e comunicação; integra diferentes visões de mundo Abraça o conflito e a dialética; Incentiva a exploração e criatividade; Oportunista; Gerente é facilitador; Design e implementação são inseparáveis e evoluem iterativamente
Racionalidade	Técnico / funcional	Substancial
Teórico e/ou raízes filosóficas	Positivismo lógico, método científico	Aprendizado de ação, pragmatismo de John Dewey, fenomenologia

DAHLEM, 2014; BECK et al., 2001) e, de acordo com BECK et al. (2001), o manifesto define que estão sendo descobertas maneiras melhores de desenvolver software, seja feito por nós mesmos ou ajudando outros a fazerem o mesmo, e que por meio deste trabalho, passando a valorizar:

- **Indivíduos e interações** mais que processos e ferramentas.
- **Software em funcionamento** mais que documentação abrangente.
- **Colaboração com o cliente** mais que negociação de contratos.
- **Responder a mudanças** mais que seguir um plano.

Mesmo havendo valor nos itens à direita, valorizam-se mais os itens à esquerda (BECK et al., 2001). Dessa forma, surgiram os 12 princípios:

- A prioridade é satisfazer o cliente, através da entrega adiantada e contínua de software de valor.
- Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adéquam a mudanças, para que o cliente possa tirar vantagens competitivas.
- Entregar software funcionando com frequência, na escala de semanas até meses, com preferência aos períodos mais curtos.

- Pessoas relacionadas à negócios e desenvolvedores devem trabalhar em conjunto e diariamente, durante todo o curso do projeto.
- Construir projetos ao redor de indivíduos motivados, dando a eles o ambiente e suporte necessários, e confiando que farão seu trabalho.
- O método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara.
- Software funcional é a medida primária de progresso.
- Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter, indefinidamente, passos constantes.
- Contínua atenção à excelência técnica e bom design, aumenta a agilidade.
- Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito.
- As melhores arquiteturas, requisitos e designs emergem de times auto-organizáveis.
- Em intervalos regulares, o time deve refletir em como ficar mais efetivo e, então, se ajustar e otimizar seu comportamento de acordo.

A implementação dos princípios do manifesto ágil deriva de diversas práticas ágeis em que a combinação define métodos ágeis (DIEBOLD; DAHLEM, 2014). Existem práticas ágeis classificadas como técnicas e não técnicas: as técnicas podem ser baseadas na codificação, como, por exemplo, o TDD; e as não técnicas podem ser baseadas na melhoria contínua, como, por exemplo, a retrospectiva (VALLON et al., 2018).

De acordo com o *Survey* divulgado pela VersionOne em 2018, algumas práticas ágeis técnicas demonstraram aumento da utilização nos padrões de codificação. Dentre as citadas estavam o teste de unidade, refatoração, integração continua, TDD e *Pair Programming* (VERSIONONE, 2018). Devido ao âmbito deste trabalho consideramos Teste de Software e Desenvolvimento Orientado por Teste (TDD).

2.2. Teste de Software

Teste de Software é a verificação dinâmica de que um programa fornece comportamentos esperados em um conjunto finito de casos de teste, adequadamente selecionados do domínio de execuções geralmente infinito (DELAMARO et al., 2016). Sendo assim, define-se os termos mais utilizados no âmbito do teste de software por meio do padrão IEEE número 24765-2010 (ISO et al., 2010): **defeito** (*fault*) – um passo, processo ou definição de dados incorreto no software; **engano** (*mistake*) – uma ação humana que produz um defeito; **erro** (*error*) - a diferença entre um valor ou condição, observada ou medida, especificada ou teoricamente correta, associada a um ou mais defeitos; e **falha** (failure) – um evento que um software ou componente de software não executa uma função requerida dentro dos limites previamente especificados. Concluímos que os termos defeito, engano e erro serão referenciados como erro (causa) e o termo falha (consequência) a um comportamento incorreto

do software. Apresentamos na Figura 2.1 a taxonomia de defeito, de acordo com a definição citada anteriormente. Desta modo, conseguimos caracterizar os erros detectados por critérios de teste.

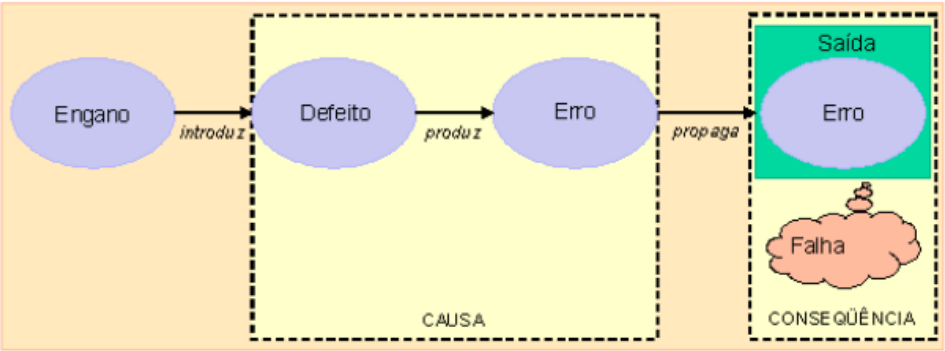

Figura 2.1. Taxonomia de defeito.

Fonte: Delamaro et al. (2016).

A atividade de teste de software é basicamente composta por quatro etapas: (i) planejamento de testes, (ii) projeto de casos de teste, (iii) execução e (iv) avaliação dos resultados dos testes. Cada etapa deve ser executada durante o processo de desenvolvimento de software, na qual concretizam-se em três fases: (i) teste de unidade, (ii) teste de integração e (iii) teste de sistema. O teste de unidade busca identificar erros de lógica e de implementação na menor unidade do projeto de software, para cada módulo de forma separada. O teste de integração consiste em encontrar erros associados às interfaces entre os módulos durante a integração da estrutura. O teste de sistema caracteriza-se por identificar erros de funções e características de desempenhos que não estejam de acordo com as especificações do software (DELAMARO et al., 2016).

Casos de teste são um conjunto de entradas de teste, condições de execução e resultados esperados desenvolvidos para um objetivo específico, como para exercer um determinado caminho do programa ou verificar a conformidade com um requisito específico (ISO et al., 2010). Para Delamaro et al. (2016), um caso de teste é um par formado por um dado de teste mais o resultado esperado para a execução do programa com aquele dado de teste. Por exemplo, no programa que computa x^Y, teríamos os seguintes casos de teste: ((2, 3), 8), ((4, 3), 64), ((3, - 1), "Erro"). Ao conjunto de todos os casos de teste usados durante uma determinada atividade de teste chamamos de conjunto de casos de teste (DELAMARO et al., 2016).

Critérios de teste têm como objetivo a redução do domínio de entrada de casos de teste enquanto possibilitam o aumento da probabilidade de detecção de erros. Critérios de teste são definidos baseados em técnicas de teste que utilizam fonte de informação para derivar requisitos de teste (Pachulski Camara; Graciotto Silva, 2016). As principais técnicas de teste são: funcional, estrutural e baseada em erro (MALDONADO, 1991; DELAMARO et

al., 2016). As técnicas são diferenciadas pela origem da informação utilizada na avaliação e construção dos conjuntos de casos de teste (MALDONADO, 1991). É importante salientar que essas técnicas de testes são complementares (VINCENZI et al., 2010).

Na técnica funcional, os critérios e requisitos do testes são definidos a partir da especificação do software, sem conhecimento sobre a estrutura e implementação do software (DELAMARO et al., 2016). A técnica funcional consiste em dois passos principais: (i) identificar as funcionalidades implementadas que o produto que deveria realizar (VINCENZI et al., 2010) e (ii) criar casos de teste que são capazes de verificar se cada funcionalidade é realizada corretamente de acordo com a especificação do software (VINCENZI et al., 2010). Exemplos desses critérios são: particionamento em classes de equivalência, análise do valor limite e grafo de causa-efeito (DELAMARO et al., 2016).

O critério de particionamento em classes de equivalência são realizados dois passos que tem como objetivo reduzir a quantidade de casos de testes e aumentar a possibilidade sobre a presença ou abstenção de erros. No primeiro passo são identificadas as classes de equivalência, divididas em classes de equivalência válida, que representam as entradas de dados válidas, e classes de equivalência inválidas, que representam as entradas de dados inválidas. No segundo passo são definidos os casos de teste, que compreende em criar casos de teste que tenham coberto cada classe de equivalência, seja válida ou inválida, com ao menos um caso de teste para cada classe inválida (MYERS et al., 2013).

O critério de análise do valor limite difere-se do critério de particionamento em classes de equivalência em dois aspectos. O primeiro aspecto compreende em, ao invés de selecionar qualquer elemento em uma classe de equivalência, são selecionados um ou mais elementos que representam o limite da classe de equivalência. O segundo aspecto, ao invés de focar somente nas condições de entrada para derivação de casos de teste, também considera as classes de equivalência de saída (MYERS et al., 2013).

O critério de grafo de causa-efeito minimiza a fraqueza de combinação de entradas dos critérios de particionamento em classes de equivalência e análise do valor limite, uma vez que estes critérios não realizam a combinação de entradas. O critério de grafo de causa-efeito utiliza um formato sistemático para definição de casos de teste, em virtude que realizar a combinação de muitas entradas pode gerar uma quantidade de combinações astronômicas (MYERS et al., 2013). Para Myers et al. (2013) o processo de derivação de casos de teste compreende em 6 fases, sendo elas: (i) divisão da especificação em pedaços viáveis, (ii) identificação dos efeitos e causas da especificação, (iii) análise e transformação para um grafo booleano que vinculam causa e efeito, (iv) anotação no grafo com as restrições descritas nas combinações das causas e/ou efeitos, (v) conversão do grafo para uma tabela de decisão e (vi) conversão de cada coluna da tabela de decisão para um caso de teste.

A técnica estrutural leva em consideração a implementação ou aspectos estruturais para determinar os requisitos do teste (DELAMARO et al., 2016; VINCENZI et al., 2010).

Em geral, a maioria dos critérios de testes estruturais usam uma abstração das estruturas internas do software, representados por um grafo (VINCENZI et al., 2010). Exemplos desses critérios são: baseados em fluxo de controle e baseados na complexidade (DELAMARO et al., 2016). No critério baseado em fluxo de controle utilizam-se apenas características de controle de execução do software, tais como comandos ou desvios para determinar quais estruturas são necessárias. Os critérios mais conhecidos são: *Statement Coverage*, que exige que a execução do software passe ao menos uma vez em cada instrução do software; *Decision Coverage*, o qual requer que todas as instruções do software sejam executadas pelo menos uma vez e cada decisão do software tomou todos os resultados possíveis em ao menos uma vez; *Condition Coverage*, o qual requer que todas as instruções do software foram executadas pelo menos uma vez e todas as condições em cada decisão tomaram todos os resultados possíveis ao menos uma vez; e *Path Coverage* que requer que todos os caminhos possíveis do software sejam executados (DELAMARO et al., 2016). No critério baseado na complexidade são utilizadas informações sobre a complexidade do software para derivar casos de teste. Um critério bastante conhecido é o McCabe (ou teste de caminho básico), que utiliza-se da complexidade ciclomática do Grafo de Fluxo de Controle (GFC). A complexidade ciclomática é uma métrica de software que proporciona uma medida quantitativa da complexidade lógica de um software. O valor da complexidade ciclomática estabelece o número de caminhos linearmente independentes do conjunto básico de um software, oferecendo um limite máximo para o número de casos de teste que devem ser derivados a fim de garantir que todas as instruções sejam executadas pelo menos uma vez (DELAMARO et al., 2016).

A técnica baseada em erro utiliza a informação sobre falhas que são frequentemente encontradas durante o processo de desenvolvimento de software (DELAMARO et al., 2016; VINCENZI et al., 2010). Exemplos desses critérios são: semeadura de erros e análise de mutantes (DELAMARO et al., 2016). No critério de semeadura de erros são introduzidos um número conhecido de defeitos artificiais no software a ser testado. Após o teste, a partir do número total de falhas encontras e da taxa entre os defeitos naturais e artificiais é possível estimar o número de defeitos naturais remanescentes (VINCENZI et al., 2010). No critério de análise de mutantes, referente à técnica de teste de mutação, defeitos são introduzidos em um software criando várias versões do software, cada um com um defeito diferente, conforme a aplicação de operadores de mutação. Os softwares defeituosos que têm como objetivo causar falhas são denominados mutantes, e quando um caso de teste falha significa que um mutante foi morto. Com este cenário o mutante satisfez seu requisito de identificar se um caso de teste é útil. Caso o mutante não falhe com um determinado caso de teste, mas o programa original falhe, o mutante revelou um defeito no software, satisfazendo o requisito da atividade de teste de software (OFFUTT et al., 1996).

2.3. Desenvolvimento Orientado por Testes (TDD)

Desenvolvimento Orientado por Testes (BECK, 2002) é uma técnica de desenvolvimento de software iterativo que concilia a implementação de novas funcionalidades e a construção de casos de teste (FUCCI et al., 2017; Pachulski Camara; Graciotto Silva, 2016; SHELTON et al., 2012). No processo do TDD, cada nova iteração consiste na implementação de uma *feature* (FUCCI et al., 2017). Três fases compõem o processo do TDD: (i) escrita do caso de teste de unidade, (ii) implementação do código de produção e (iii) refatoração (FUCCI et al., 2017; Pachulski Camara; Graciotto Silva, 2016; SHELTON et al., 2012). A iteração inicia-se pela escrita do caso de teste de unidade, seguida da fase de implementação do código de produção, e finaliza-se na fase de refatoração (FUCCI et al., 2017; SHELTON et al., 2012). Encerra-se a iteração quando todas as fases do processo são executadas e os testes de unidade são executados com sucesso (FUCCI et al., 2017). A principal regra do TDD é: *"If you can't write a test for what you are about to code, then you shouldn't even be thinking about coding"* (GEORGE; WILLIAMS, 2004). TDD provê ao desenvolvimento de software testes de regressão contínuos, que possibilitam a melhoria da qualidade do software (BECK, 2002). Durante o ciclo do TDD, é possível conciliar técnicas de teste de software ao seu processo, como por exemplo a utilização de critérios de teste (Pachulski Camara; Graciotto Silva, 2016).

Os efeitos do TDD têm sido estudados amplamente (FUCCI et al., 2017). Desta forma, os trabalhos relacionados têm como foco auxiliar desenvolvedores com a utilização do TDD e melhorar o design do software, de modo que seja possível obter a melhor solução para o cenário desejado (Pachulski Camara; Graciotto Silva, 2016). Nosso estudo consiste em dois aspectos principais: conformidade do processo do TDD e qualidade dos casos de teste utilizando critérios de teste. Quando considerados relacionados ao corpo de conhecimento:

1. Impactos na qualidade e produtividade a partir das quatro características dos processos do TDD: sequência, granularidade, uniformidade e esforço de refatoração (FUCCI et al., 2017);
2. Análise empírica da adoção do TDD na indústria de software, com a finalidade de evidenciar se a organização estava apta a adotar TDD (SANTOS et al., 2018);
3. Eficácia do uso de critérios de teste na elaboração de casos de teste de alta qualidade no contexto da academia utilizando TDD (Pachulski Camara; Graciotto Silva, 2016);
4. Aumento da qualidade dos casos de testes escritos durante o ciclo do TDD utilizando técnicas de design de teste de software (CAUSEVIC et al., 2013a);
5. Investigação sobre a existência do viés de testes positivos no contexto da indústria utilizando TDD, possibilitando mensurar a taxa de defeitos encontrados com testes positivos e negativos (CAUSEVIC et al., 2013b);

6. Geração de casos de testes adicionais como parte do ciclo do TDD, utilizando como critério para inclusão de novos casos de teste a análise de mutantes (SHELTON et al., 2012);
7. Eficácia de casos de testes gerados durante o TDD utilizando indicadores como *test smells*, *code coverage* e *mutation score indicator* (TOSUN et al., 2018).

No estudo de Fucci et al. (2017), os autores apresentaram um amplo estudo sobre os processos do TDD. Como objetivo os autores buscaram encontrar quais os impactos dos efeitos que as características do processo do TDD podem resultar na qualidade externa do software e na produtividade dos desenvolvedores. Os autores identificaram quatro características no processo do TDD: (i) granularidade, (ii) uniformidade, (iii) sequência e (iv) esforço de refatoração, conforme detalhadas na Tabela 2.2. Os autores conduziram um *quasi-experimento* no contexto da indústria de software. Para produção dos dados, os autores executaram quatro *workshops* com temas sobre testes unitários, TDD, *Test-First* (TF), *Test-Last* (TL) e processo iterativo de testes unitários. Cada execução do *workshop* durou cinco dias. Para obter os dados dos ciclos de desenvolvimento, os autores utilizaram a ferramenta Besouro[1]. Os dados gerados pela ferramenta foram utilizados para calcular métricas que representam as características comentadas na Tabela 2.2. Com o estudo, foi possível concluir que os benefícios do TDD não são providos apenas da dinâmica do TF: TDD como processo encoraja os desenvolvedores a seguirem o desenvolvimento com ciclos curtos e uniformes, melhorando o foco e fluxo de desenvolvimento.

Tabela 2.2. Características correspondentes aos processos do TDD.

Fonte: Fucci et al. (2017).

	Interpretação
(i) Granularidade	Caracterizado por um processo de desenvolvimento curto, onde cada ciclo dura tipicamente entre 5 e 10 minutos. Um valor pequeno indica um processo granular. Um valor grande indica um processo não granular ou não aperfeiçoado.
(ii) Uniformidade	Caracterizado por ciclos de desenvolvimento que duram aproximadamente o mesmo tempo. Um valor próximo a zero indica um processo uniforme. Um valor distante de zero indica um processo não uniforme ou instável.
(iii) Sequencia	Indica a prevalência da sequencia TF durante o processo de desenvolvimento. Um valor próximo de zero indica a violação dinâmica do TF. Um valor distante de zero indica predominância de TF.
(iv) Esforço de Refatoração	Indica a prevalência da atividade de refatoração no processo de desenvolvimento. Um valor próximo a zero indica a negligencia do esforço de refatoração. Um valor distante de zero indica o esforço de refatoração.

[1] https://github.com/brunopedroso/besouro

No estudo de Santos et al. (2018), os autores auxiliaram gestores a conhecer o TDD no âmbito de suas instalações. O objetivo era comprovar de forma empírica que o TDD supera o formato tradicional de codificação na qualidade externa do software. Caso o objetivo fosse comprovado os gestores adotariam o TDD na organização. Conduziram um experimento controlado envolvendo o desenvolvimento de software utilizando maneira tradicional, TDD e TL. O experimento controlado contou com a participação de 15 colaboradores, na qual os participantes desenvolveram em grupos três softwares: *Bowling Score Keeper*, *Mars Rovers* e *Spread Sheet*. Os participantes também responderam um *Survey* com a finalidade de mensurar sua experiência. Após a execução do experimento controlado e a análise dos dados, os autores constataram que a empresa não deveria adotar TDD imediatamente, visto que TDD superou as outras abordagens, porém com uma diferença de desempenho pequena. Contudo, os autores sugeriram que os gestores capacitem colaboradores com TDD e futuramente realizem um novo experimento controlado, uma vez que, os participantes não possuíam experiência suficiente com TDD.

No estudo de Pachulski Camara e Graciotto Silva (2016), os autores tiveram como objetivo verificar a eficácia de critérios de teste para produzir casos de testes de alta qualidade com TDD. Os autores propuseram uma abordagem que altera a fase de refatoração no fluxo do processo do TDD. Na fase alterada, incluíram uma nova atividade, que é responsável por adicionar um caso de teste baseado em critérios de teste e realizar a validação da cobertura de código fonte conforme exibido na Figura 2.2. Para este estudo, foram adotados dois critérios de teste: *statement coverage* e *branch coverage*. O estudo contemplou um experimento controlado com estudantes. O experimento controlado consistiu em três iterações, sendo que cada iteração tinha como objetivo incluir uma nova *feature* ao software. Para coletar os dados dos estudantes, os autores estenderam o *plugin* Eclemma[2] e desenvolveram um novo *plugin* que, a cada execução dos casos de testes pelos estudantes, enviava as informações para um servidor responsável por identificar os estudantes e armazenar os dados. Os autores observaram que o uso de critérios de teste melhorou a qualidade dos casos de teste sem causar distúrbios no processo do TDD.

[2] http://www.eclemma.org/

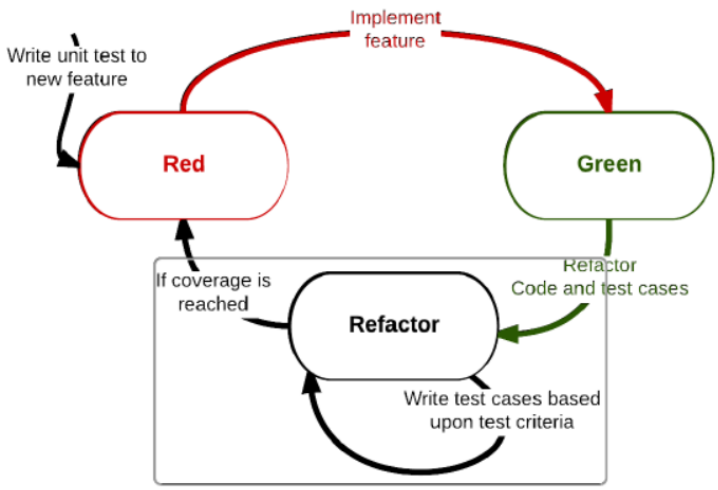

Figura 2.2. Fluxo do Processo do TDD utilizando critérios de teste.
Fonte: Pachulski Camara e Graciotto Silva (2016).

No estudo de Causevic et al. (2013a), os autores apresentaram o conceito do Desenvolvimento Orientado por Teste - Alta Qualidade (TDD^{HQ}). O TDD^{HQ} consiste em alcançar alta qualidade dos casos de testes desenvolvidos durante a utilização do TDD. Com a utilização do TDD^{HQ} os desenvolvedores não focam apenas na funcionalidade de acordo com os requisitos, mas também no aumento da segurança, robustez, performance e outros aspectos de melhoria de qualidade. Os autores adicionaram novos processos ao fluxo do TDD, conforme exibido na Figura 2.3. Na alteração do fluxo do processo, foram adicionados quatro processos antes da adição de um novo caso de teste: (i) escolha do aspecto de melhoria da qualidade, (ii) escolha da técnica de design de teste, (iii) verificação da existência de mais casos de testes a serem adicionados para que a técnica de design de teste satisfaça o aspecto de melhoria da qualidade, (iv) verificação da existência de mais aspectos de melhoria da qualidade ou adição de novas *features*. Contudo, mesmo com a adição de novos processos no fluxo do TDD e casos de testes com maior qualidade, não houve interferência no clássico *"red-green-refactor"* do TDD. O estudo também contemplou um experimento controlado na academia. Durante o experimento controlado, foram analisados resultados de dois grupos: um grupo seguindo a abordagem tradicional do TDD e outro grupo seguindo a abordagem proposta pelos autores. Foi possível constatar que o grupo que utilizou TDD^{HQ} criou menos casos de testes e detectou mais defeitos comparado ao grupo que usou o TDD de forma clássica.

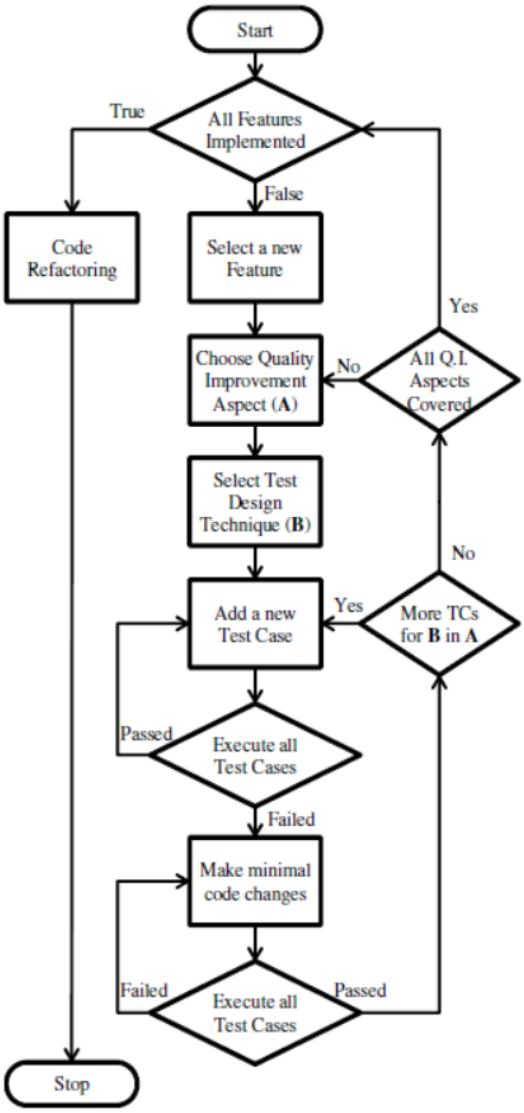

Figura 2.3. Fluxo do Processo do TDD^{HQ}.

Fonte: Causevic et al. (2013a).

No estudo de Causevic et al. (2013b), os autores realizaram uma investigação sobre a existência de viés de casos de testes positivos durante o TDD. Também mensuraram a habilidade de detecção de defeitos para casos de testes positivos e negativos. Os autores se preocuparam em criar medidas para auxiliar a evolução e melhoria da qualidade dos testes. A habilidade de detecção de defeitos representa o número total de defeitos encontrados por um caso de teste. A habilidade foi medida pelos autores utilizando a qualidade do código e dos casos de teste. Foi conduzido um experimento controlado no contexto da indústria. O experimento teve muitos objetivos, sendo o principal uma investigação sobre os efeitos do viés de testes positivos identificado na indústria durante a utilização da abordagem de TL ou TDD. Durante a execução do experimento os profissionais foram divididos em grupos para construção de um software utilizando TL, TDD e Desenvolvimento Orientado por Teste com

Suporte para Casos de Testes Negativos (TDD+). Os autores concluíram que existe diferença entre o número total de casos de testes positivos e negativos criados pelos participantes. Também constatou-se que existe diferença entre o número de detecção de asserções que falham e qualidade dos casos de testes positivos e negativos.

No estudo de Shelton et al. (2012), os autores apresentaram resultados de um estudo observacional, no qual profissionais adicionavam casos de testes baseados em critérios de teste como parte do processo do TDD. Utilizaram o critério de teste de análise de mutação e geração de casos de testes com objetivo de matar os mutantes que não foram mortos pelo TDD. Os autores adicionaram ao fluxo do processo do TDD a geração de casos de teste dirigidos por mutação após o ciclo *red-green-refactor*, conforme exibido na Figura 2.4. O estudo teve três principais objetivos: (i) determinar a cobertura de mutação dos casos de testes escritos por desenvolvedores, (ii) determinar se a adição de casos de testes derivados por meio do uso da análise de mutantes potencializa a detecção de defeitos ou possíveis deficiências, (iii) determinar se a geração de casos de testes derivados por meio do uso da análise de mutantes pode dificultar o processo do TDD. Para alcançar os objetivos os autores convidaram programadores profissionais para desenvolver um software usando TDD em duas iterações. Três profissionais convidados participaram do desenvolvimento do software proposto pelos autores. Algumas métricas foram coletadas durante o estudo, como número de classes criadas e número de defeitos encontrados pelos casos de teste. Com o estudo foi possível constatar que a cobertura do código fonte aumentou em todos os cenários de mutação. Os critérios de teste não revelaram apenas defeitos, mas também deficiências que podem ser melhoradas por programadores. Além disso, os participantes indicaram que a inclusão de critérios de teste são bem vindas e não interrompem o ritmo do TDD. Os participantes também reportaram que analisar casos de testes adicionais e cobertura de código fonte requer esforço adicional, esforço que é considerado positivo para melhoraria softwares.

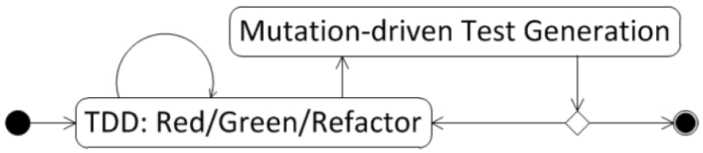

Figura 2.4. Fluxo do Processo do TDD com Geração de Casos de Teste Dirigidos por Mutação. Fonte: Shelton et al. (2012).

No estudo de Tosun et al. (2018), os autores investigaram os impactos do TDD na eficácia de casos de teste de unidade comparados com desenvolvimento utilizando TL em um contexto industrial. Os autores discutiram sobre três características para mensurar a eficácia da qualidade dos casos de teste: (i) *test smells*, (ii) *code coverage* e (iii) *mutation score indicator*. A primeira medida descrita foi *Test Smells* é similar ao termo *Code Smells*, ou seja, quando algo não parece certo no código de produção. Foram citados como *Test Smells*:

mistério para convidados, otimização de recursos, guerra com testes, equipamentos gerais, teste ansioso, teste lento, roleta de afirmações, testes indiretos, apenas para testadores, igualdade sensível e duplicação de código de teste. A segunda medida exposta foi *Code Coverage*, que é uma medida comumente utilizada e que compreende em quais partes o código de produção são exercitados/executados durante o teste. Existem diferentes maneiras de mensurar *Code Coverage*, as mais comuns são a cobertura por linha, instrução e ramos. A terceira medida relatada foi o *Mutation Score Indicator*, que é uma maneira de aferir a habilidade de casos de testes capturarem defeitos no código utilizando testes de mutação. Diferente de muitos estudos, os autores conduziram um experimento controlado com profissionais localizados em três diferentes locais de uma empresa de software. Para coletar os dados de todos os participantes foram utilizadas as ferramentas *EclEmma* e *Judy*. Os 24 participantes desenvolveram utilizando TDD o *Bowling Score Keeper* e com TL desenvolveram *Mars Rover API*. Para validar a hipótese levantada utilizou-se o Wilcoxon Rank Sum Test, na qual foi possível constatar que existe uma diferença significante na eficácia dos casos de testes desenvolvidos com TDD frente aos casos de testes desenvolvidos com TL.

2.4. Considerações Finais

Neste capítulo foi possível contextualizar todos os objetos deste estudo, iniciando com o desenvolvimento ágil de software, envolvendo práticas ágeis voltadas para qualidade do software como Desenvolvimento Orientado por Teste (TDD) e finalizando com a importância do teste de software e critérios de teste. Abordamos também trabalhos relacionados que buscam evoluir o mesmo campo de conhecimento aplicado neste estudo. Mesmo com diversos estudos publicados sobre a utilização do TDD, existe uma lacuna entre TDD e Teste de Software, de forma mais específica, em relação a empregabilidade de técnicas de Teste de Software durante a utilização do TDD. Em virtude desta lacuna, direcionamos o foco deste estudo para Teste de Software utilizando critérios de teste funcional, onde, aplicaremos critérios de teste durante a utilização do TDD, onde avaliaremos aspectos da conformidade do processo do TDD e qualidade do casos de teste utilizando critérios de teste.

Capítulo 3

Método

Neste capítulo descreve-se com detalhes o método utilizado. Aborda-se também o cenário em que se enquadra o estudo além dos objetivos a serem alcançados. Inicia-se pela definição do escopo de aplicação do estudo, na qual englobam-se empresas de software que utilizam desenvolvimento ágil.

3.1. Processo Organizacional

Antes de iniciar a descrição do processo metodológico, é necessário entender o fluxo do processo na qual se aplica este estudo. O método ágil considerado é o Scrum (SCHWABER, 1995; SCHWABER; SUTHERLAND, 2017), o qual é composto por ciclos de entregas de valor para o cliente. Essa escolha se deve ao fato de ser o método tipicamente utilizado na indústria de software (RODRíGUEZ et al., 2012; NAZIR et al., 2016; DOLEZEL et al., 2019; Digital.ai, 2020) e inclusive, na empresa em que o estudo foi aplicado. O fluxo do processo é divido em três fases:

1. Priorização das *features* a serem desenvolvidas.
2. Iterações do desenvolvimento de conjunto de *features*.
3. Atividades diárias de desenvolvimento.

Para melhor compreensão das fases do processo organizacional, é necessário conhecer todos os papéis praticados durante o processo de desenvolvimento ágil. Os nomes das funções de cada colaborador podem sofrer alterações de acordo com a organização, no entanto, o papel exercido é semelhante.

- ***Product Owner* (PO):** responsável por organizar e priorizar a lista de funcionalidades que deverão contemplar o produto de software, também conhecido *Product Backlog* (PB). Normalmente assume a função de analista de requisitos ou analista de negócios.

- ***Scrum Team* (ST):** composto por desenvolvedores, testadores, líderes técnicos e arquitetos, no qual todos são responsáveis pelo desenvolvimento e qualidade da funcionalidade do produto de software que será entregue, caracterizado como *Product Increment* (PI).
- ***Scrum Master* (SM):** responsável por liderar e apoiar o ST. Garante que o time não será interrompido por fatores externos. Normalmente é representado por um agilista, podendo em alguns casos ser representado por algum membro do time de desenvolvimento e também por assistentes de projeto. O SM também é um intermediador do ST com PO ou *Stakeholders*.
- ***Stakeholders*:** pessoas interessadas no PI, geralmente são lideres formais dentro da organização. Podem ser representados por coordenadores, gerentes de projeto, gestores de produto e clientes finais.

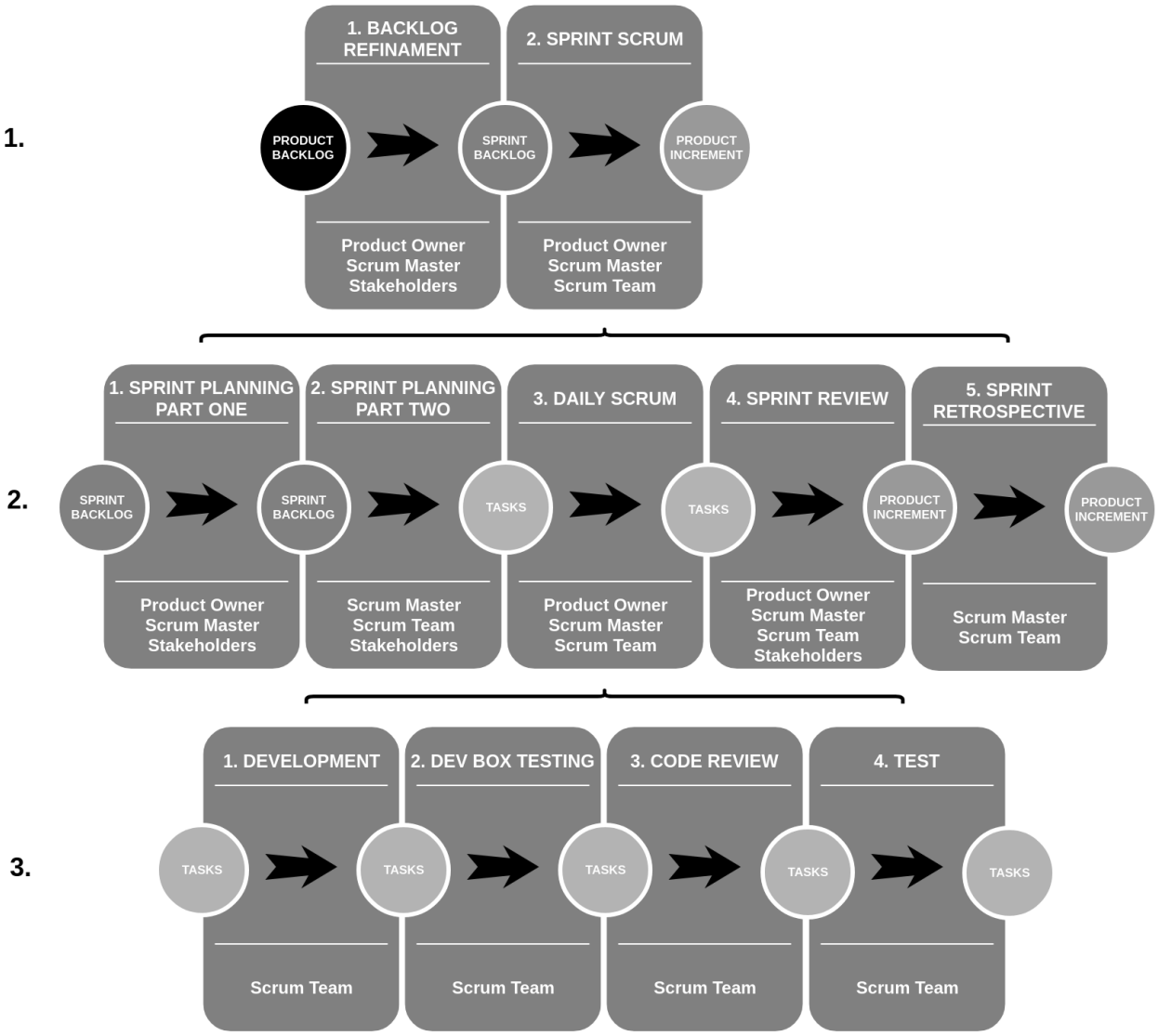

Figura 3.1. Fluxo do Processo.

Fonte: Autoria própria.

Dessa forma, apresenta-se na Figura 3.1 o fluxo do processo. Na primeira fase (1), obtêm-se todas as *features* que deverão ser entregues para determinado produto. Dentre as *features*, encontram-se novas funcionalidades, melhorias, deficiências funcionais e erros. Esse conjunto de *features* é definido como PB. Com o PB definido, inicia-se o processo de *Backlog Refinament* (BR). PO, SM e *Stakeholders* se reúnem para selecionar as *features* que serão trabalhadas na *Sprint Scrum* (SS). As *features* que compõem o *Sprint Backlog* (SB) dependem do tamanho e da quantidade de recursos disponíveis na SS. Caso o tamanho das *features* escolhidas pelo PO, SM e *Stakeholders* ultrapassem a quantidade de recursos, os *Stakeholders* são responsáveis por encontrar recursos para que o SB seja entregue. Para que as *features* sejam selecionadas para o SB é necessário que as mesmas possuam requisitos de negócio e técnico, e itens necessários para execução das atividades (servidores, bases de dados, ferramentas, etc) definidos. Finalizado o processo de BR, pode-se iniciar a SS.

Na segunda fase (2), inicia-se a SS. Normalmente uma SS tem duração de duas semanas e é compostas de 4 e 10 membros, que fazem parte do ST. Cada membro do time tem seu papel específico, podendo realizar atividades de outros papéis. Inicia-se a SS com a *Sprint Planning - Part One* (SP1), onde PO e *Stakeholders* são responsáveis por informar ao ST quais *features* desejam receber ao final da SS no PI. Após a finalização da SP1, o ST se reúne para planejar como desenvolverão as *features* elencadas na SP1. Então, inicia-se a *Sprint Planning - Part Two* (SP2), onde o ST realiza a quebra das *features* em tarefas que possibilitem o paralelismo e entrega antecipada. Após feita a quebra de tarefas, é realizada aceitação das *features* que serão desenvolvidas. Caso o ST não aceite alguma *feature*, o SM é encarregado de negociar as *features* com o PO e *Stakeholders*. Após a finalização da SP2 inicia-se o desenvolvimento das *features*. Diariamente o ST se reúne para verificar o andamento das atividades, levantando impedimentos e alinhando as expectativas de cada membro do time. Ao fim do desenvolvimento, acontece a *Sprint Review* (SR). Na SR o ST demonstra para PO, SM e *Stakeholders* o PI desenvolvido. Durante a SR é possível que erros sejam encontrados e, caso isso ocorra, esses erros são adicionados ao *backlog* de erros. Por fim, o ST se reúne com o SM para realizar a última atividade SS, a *Sprint Retrospective*. Nesta reunião são levantados os pontos que podem ser melhorados pelo time, com intuito de aumentar o potencial do ST.

Na terceira fase (3), são realizadas as atividades diárias da SS. Estas atividades são compreendidas pelo desenvolvimento das *features*, com intuito de atingir o objetivo da SS. O ST realiza o desenvolvimento e teste das *features* selecionadas no SB. Após o fim do desenvolvimento, o desenvolvedor convida um testador para fazer uma validação rápida do que foi desenvolvido, a fim de encontrar problemas impeditivos para realização dos testes. Após essa validação, o ST realiza o *code review*, com intuito de melhorar a qualidade do código e encontrar defeitos. Após isso, o teste da *feature* é executado pela equipe de testes,

garantindo a qualidade do PI. Os defeitos encontrados durante as atividades diárias são corrigidos pelo ST.

3.2. Questão de Pesquisa

O objetivo deste estudo é propor a adoção do TDD em conjunto com critérios de teste, analisando efeitos gerados na adoção e avaliando a melhoria do desenvolvimento de produtos em indústrias de software inseridas no contexto ágil com modelo iterativo. A partir desse objetivo, surge a seguinte questão de pesquisa:

- *Existem efeitos de melhoria no desenvolvimento de software com a adoção do Desenvolvimento Orientado por Teste (TDD) com critérios de teste?*

De acordo com alguns autores, a abordagem *Goal Question Metric* (GQM) é recomendada para definição de estudos experimentais (WOHLIN et al., 2000; JURISTO; MORENO, 2001). O paradigma GQM é um mecanismo para definir e evoluir um conjunto de objetivos utilizando métricas (BASILI, 1992). Para que uma organização possa mensurar de maneira proposital, é necessário especificar objetivos para ela mesma e para seus produtos, além de definir para os objetivos pretendidos, e fornecer ferramentas que possibilitem a interpretação dos dados dos objetivos (BASILI, 1992). Para que estudos empíricos nos ambientes industriais sejam efetivos com a aplicação de métricas e modelos, é essencial possuir três características (BASILI et al., 1994):

1. Focado em objetivos específicos;
2. Aplicado para todo o ciclo de vida de produtos, processos e recursos;
3. Interpretado baseado na caracterização e entendimento do contexto organizacional, ambiente e objetivos.

Logo, utilizou-se neste trabalho a abordagem GQM para definição do estudo experimental. Desenvolveu-se o objetivo em volta do TDD e critérios de teste, que são objetos deste estudo, possibilitando mensurar esses objetivos. Também elencou-se questões com intuito de definir completamente os objetivos de forma quantificável. Foi necessário especificar medidas objetivas para responder as questões identificadas, garantindo assim que o objetivo inicial seja alcançado. O modelo GQM elaborado para responder a questão de pesquisa deste estudo consiste de uma meta, duas questões e cinco métricas, conforme descritas no restante desta seção e na Tabela 3.1. A relação entre os níveis é apresentada na Figura 3.2.

Apresenta-se a seguir a meta definida no modelo GQM. Esta meta auxiliará a responder a questão de pesquisa deste estudo. A meta corresponde na avaliação da conformidade do processo do TDD, possibilitando a avaliação da utilização do TDD em conjunto com critérios de teste, de forma a evidenciar que ao utilizar critérios de teste a

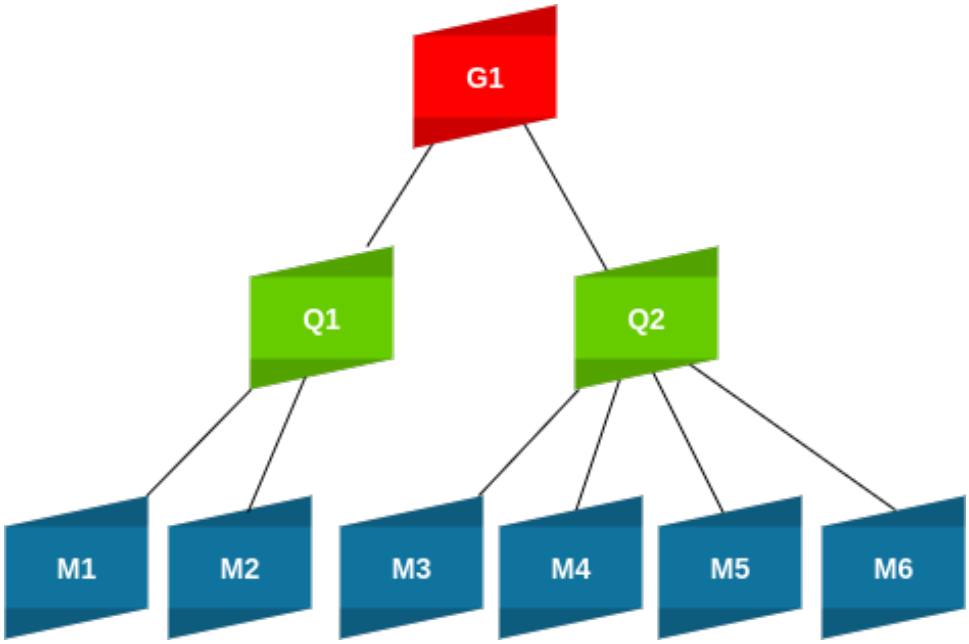

Figura 3.2. Modelo GQM utilizado para definição do estudo experimental.
Fonte: Autoria própria.

conformidade do TDD poderá ser potencializada. A meta refere-se aos efeitos de melhoria da qualidade do software produzido, garantindo que o propósito de qualidade do código fonte do TDD seja correspondido mesmo com a adição de uma técnica de teste. Essa meta também apoiará na análise dos reflexos do TDD em conjunto com critérios de teste na qualidade externa do software, avaliando seus reflexos ao adicionar uma técnica de teste.

O propósito deste estudo é avaliar os efeitos com foco na qualidade de software durante a adoção do TDD em conjunto com critérios de teste, no ponto de vista de desenvolvedores, testadores, arquitetos e gerentes de projeto, na indústria de software em um contexto ágil com a utilização do modelo iterativo.

- **Objeto de estudo:** TDD
- **Propósito:** avaliar os efeitos da adoção de critério de teste
- **Foco:** conformidade do processo do TDD e qualidade do software
- **Interessados:** desenvolvedores, testadores, arquitetos e gerentes de projeto
- **Contexto:** indústria de software, contexto ágil, modelo iterativo

Neste estudo, duas questões foram estabelecidas pelo modelo GQM. A questão Q1 responderá se existe aumento significativo da conformidade do processo do TDD ao incluir critérios de teste em seu processo. A questão Q2 verificará se métricas de qualidade do código fonte terão seus indicadores melhorados com a utilização de critérios de teste, além de responder se a redução de defeitos no produto de software acontece de fato. Com estas respostas será possível atingir a meta estabelecida no modelo GQM.

- **Q1:** Existe aumento da conformidade do processo do TDD em conjunto com critérios de teste?
- **Q2:** Existe aumento da qualidade do software com a utilização do TDD em conjunto com critérios de teste?

Na Tabela 3.1 apresenta-se as cinco medidas originadas a partir das questões do modelo GQM, as quais auxiliarão a responder as questões de forma quantitativa. As medidas percentual de utilização do TDD (M1) e percentual de cobertura dos requisitos de teste funcional (M2) são responsáveis por coletar as informações sobre a conformidade do processo do TDD, em conjunto com a utilização de critérios de teste. A medida correspondente ao número de *code smells* (M3) e o percentual de cobertura de linhas (M4) serão coletadas por meio da ferramenta *SonarQube*, software responsável por armazenar e atualizar essas informações. A medida de percentual de cobertura de mutação (M5) será colhida com auxílio da ferramenta Pitest[1]. A medida número de defeitos (M6) será coletada manualmente durante a execução de casos de testes estabelecidos de acordo com critérios de teste funcionais. Todas as medidas citadas serão retiradas da fase 3 do processo ilustrado na Figura 3.1.

Tabela 3.1. GQM: Medidas utilizadas para responder a questão de pesquisa.

Fonte: Autoria própria.

Medida	**Descrição**	**Questão**
M1	Percentual de utilização do TDD	Q1
M2	Percentual de cobertura dos requisitos de teste funcional	Q1
M3	Número de *code smells*	Q2
M4	Percentual de cobertura de linhas	Q2
M5	Percentual de cobertura de mutação	Q2
M6	Número de defeitos	Q2

As métricas M1 e M2 foram utilizadas para responder a questão Q1, uma vez que será possível comparar as métricas M1 e M2, mensurando a conformidade do processo do TDD. Na questão Q2, foi possível apurar se com o aumento da conformidade do processo do TDD obtém-se o aumento da qualidade do software dispondo das métricas M3, M4, M5 e M6. Para ambas as questões, as métricas foram comparadas com ou sem uso de critérios de teste durante a utilização do TDD.

3.3. Abordagem do TDD com critério de teste

Para alcançar o objetivo mencionado e responder a questão de pesquisa deste estudo, adiciona-se critérios de teste na fase de criação de casos de teste (ou seja, na fase *1. red*) no processo do TDD, conforme ilustrado na Figura 3.3. Toda vez que o desenvolvedor iniciar um novo ciclo do TDD, deverá escrever um novo caso de teste utilizando critérios de teste. As

[1] Ferramenta de análise dinâmica de código fonte, disponível em <http://pitest.org/>.

demais fases do processo serão semelhantes ao processo atual do TDD, em que o desenvolvedor escreve somente o código necessário para que os casos de testes passem e ao final do ciclo evolua o código fonte sem alterar seu comportamento.

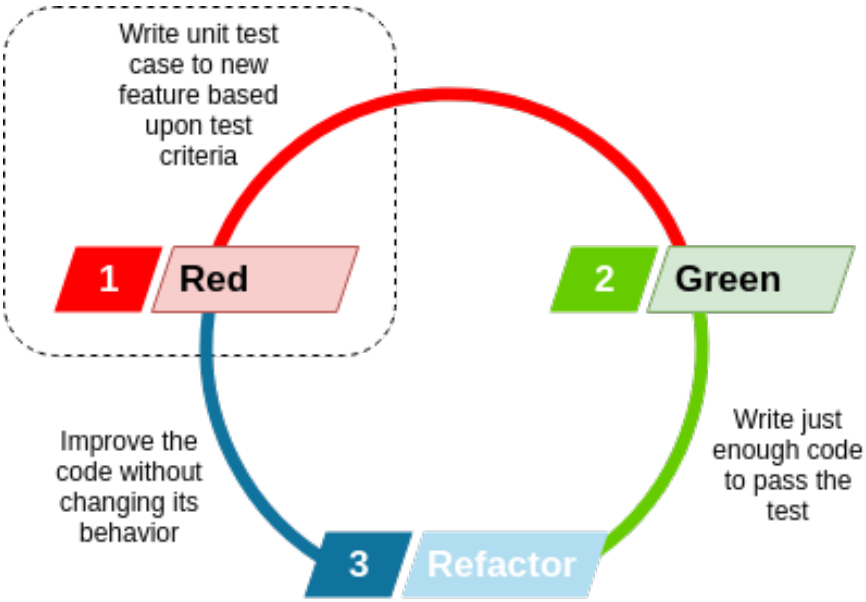

Figura 3.3. Fluxo do Processo do Desenvolvimento Orientado por Teste (TDD) utilizando critérios de teste.

Fonte: Autoria própria.

No passo de adição de casos de testes não foram utilizados critérios de teste estrutural e baseados em erro. Foram utilizados apenas critérios de teste funcional. Os critérios de teste funcional utilizados pelos desenvolvedores foram: particionamento em classes de equivalência e análise do valor limite.

3.3.1. Processo Organizacional Simplificado

Caso não seja possível estabelecer um cenário do processo organizacional semelhante ao mencionado na Seção 3.1, descrevemos como adicionar os elementos mínimos para execução do estudo. É necessário escolher participantes com conhecimento em programação, teste automatizado, ferramenta de testes automatizados. Também deverão ter participantes com e sem conhecimento de critérios de teste. É imprescindível a realização da caracterização dos participantes. É obrigatório realizar o treinamento sobre TDD, Teste de Software e critérios de teste. Durante a execução, são utilizados problemas ou requisitos de software para serem resolvidos utilizando programação com seus critérios de teste funcionais definidos. Também é necessário que os participantes do estudo experimental possuam conhecimento em linguagem de programação e em alguma *Integrated Development Environment* (IDE). Para avaliar os resultados é necessário ter ferramenta de conformidade do processo do TDD, ferramenta de análise estática de código e ferramenta para análise de mutantes.

3.3.2. Sujeitos

O estudo foi planejado para aplicação com desenvolvedores de empresas produtoras de software. Para a organização dos grupos que participaram do experimento, foi desenvolvido um questionário para medir o conhecimento e habilidades declarados em programação e teste de software. O questionário possui 14 perguntas, sendo 11 com respostas na escala Likert de 5 pontos e 3 com respostas dissertativas. As perguntas enviadas para os participantes questionaram sobre o conhecimento sobre a linguagem Java, teste automatizado e suas ferramentas, assim como TDD e critérios de teste. Para analisar a consistência interna, deve ser calculado o α de Cronbach para as perguntas com escala Likert.

3.3.3. Bowling Game Microservice

O Bowling Game Microservice segue as mesmas diretrizes disponibilizadas pela indústria de software. Com esta abordagem é possível aproximar os exemplos do Bowling Game de outros estudos com o contexto industrial. Estas diretrizes, originadas da indústria de software, são formadas pelo uso da Arquitetura de Camadas e REST, além da linguagem Java e Spring Framework. Este microserviço, contém as camadas de domínio, persistência, negócio e controle, onde são disponibilizados endpoints REST utilizando o Spring Framework. O exemplo deste microserviço pode ser encontrado no Github[2].

3.3.4. Planejamento da Execução

Para execução deste estudo de caso, planejou-se três passos a serem seguidos. O primeiro passo compreende pelo treinamento que será ministrado aos participantes do estudo. O segundo passo corresponde à execução de desenvolvimento, no entanto, com a utilização apenas do TDD no desenvolvimento de software. Por fim, o terceiro passo que compreende pela utilização do TDD e critérios de teste no desenvolvimento de software.

Antes de iniciarem o processo de desenvolvimento, os participantes receberam treinamento sobre TDD e critérios de teste. O treinamento contou com uma apresentação dos conceitos do TDD e critérios de teste, além de um exercício prático desenvolvendo o marcador de pontos de boliche, denominado *Bowling Game*, utilizando TDD e critérios de teste. O treinamento utilizou-se dos mesmos sujeitos e requisitos do experimento. Após o treinamento, foram atribuídas aos desenvolvedores somente atividades classificadas com grau de dificuldade média, que são representadas pela criação de um *endpoint* com Arquitetura REST, com lógica de negócio implementada na camada de serviço e acesso a alguma fonte de dados conhecida, como banco de dados ou *web services*.

[2] https://github.com/altitdb/tdd/tree/master/bowling-game-microservice

Posteriormente, durante a SS cada desenvolvedor construirá sua tarefa utilizando TDD e critérios de teste. As atividades serão priorizadas pelo PO e refinadas pelo ST. O tempo estimado para desenvolvimento de cada atividade é definida pelo ST. O número de horas estimados pode divergir do tempo gasto no desenvolvimento. Cada desenvolvedor recebeu 3 atividades para executar sem o uso do TDD e critérios de teste. Após a conclusão das 3 atividades sem uso do TDD e critérios de teste, os desenvolvedores receberão 3 novas atividades para realizar o desenvolvimento utilizando apenas TDD. Por fim, após o término das 3 atividades utilizando apenas TDD, os desenvolvedores receberam mais 3 atividades para serem desenvolvidas utilizando TDD e critérios de teste. No total serão desenvolvidas por cada desenvolvedor 9 atividades.

Ao fim de cada atividade os resultados gerados pela ferramenta Butterfly deverão ser anexados na atividade descrita no *Visual Studio Team Services* (VSTS). Também será colhido e anexado na atividade descrita no VSTS os dados providos pela ferramenta PITest e SonarQube. Após o fim do desenvolvimento, iniciam-se os testes exploratórios que serão realizados pela equipe de teste. A equipe é responsável por anexar o número de defeitos encontrado em cada *feature* no VSTS. Finalizadas todas as etapas do processo, os dados serão analisados. Por fim, será aplicado instrumento NASA-TLX, com a finalidade de analisar o esforço percebido pelos desenvolvedores na utilização do TDD e TDD em conjunto com critérios de teste. Este teste é uma ferramenta de avaliação multidimensional, subjetiva e amplamente utilizada. A ferramenta avalia a carga de trabalho percebida para avaliar a eficácia de uma tarefa. Disponibilizamos o exemplo do questionário NASA-TLX no Appendix A.

Capítulo 4

Resultados

Neste capítulo apresentamos os resultados deste estudo: a ferramenta Butterfly e a abordagem de TDD aliada a critérios de teste funcional. A ferramenta **Butterfly** (MATOS et al., 2018b), descrita na Seção 4.1, é capaz de mensurar a conformidade do ciclo de desenvolvimento iterativo de software, possibilitando que sejam evidenciados claramente fatores que limitam a adoção do TDD na indústria de software, como a insuficiência de experiência/conhecimento com TDD e insuficiência na aderência do protocolo do TDD (CAUSEVIC et al., 2011). A ferramenta Butterfly também suporta a coleta da medida do percentual de utilização do TDD (M1) descrita na tabela Tabela 3.1. Com o emprego desta ferramenta e outras medidas, conforme apresentado no capítulo anterior, foi avaliada a abordagem de TDD com critérios funcionais, aplicada em contexto similar àquele da indústria de software ágil, como apresentado na Seção 4.2.

4.1. Butterfly

A ferramenta Butterfly tem como objetivo avaliar a conformidade do ciclo de desenvolvimento iterativo de software. A ferramenta utiliza heurísticas para classificar qual a forma esta sendo utilizada pelo desenvolvedor: *Test Addition*, *Test-first*, *Test-last*, TDD, *Refactoring* e *Unknown*. As heurísticas são compostas por ações captadas pela ferramenta durante o desenvolvimento de software. A ferramenta é instalada na IDE e age de forma transparente, não afetando a produtividade dos desenvolvedores que a usam. A ferramenta também fornece um relatório sobre as ações e categorias classificadas de acordo com o formato de desenvolvimento utilizado, além da opção de comentários caso exista alguma dúvida da classificação realizada. Tais ações e heurísticas serão detalhadas a seguir, e foram evoluídas a partir dos estudos de Fucci et al. (2017) e Kou et al. (2010). Com esta ferramenta é possível obter a medida M1 descrita na Tabela 3.1, possibilitando mensurar o percentual de utilização do TDD durante o desenvolvimento de software iterativo.

4.1.1. Ações

Para definir cada heurística, foi necessário classificar as ações que são frequentemente executadas por desenvolvedores de software. Na Tabela 4.1 apresentamos cada ação e sua respectiva interpretação. São classificadas cinco ações essenciais para produzir as heurísticas necessárias, possibilitando a classificação de cada cenário utilizado no ciclo de desenvolvimento iterativo de software. A ação *Test Creation* compreende a criação de um caso de teste automático, seja antes ou depois da escrita do código fonte de produção. A ação *Test Pass* refere-se a execução de um ou mais casos de testes automáticos com sucesso. A ação *Test Failure*, ao contrário da ação *Test Pass*, é relacionada a execução de um ou mais casos de testes automáticos que falham. A ação de *Test Editing* engloba a inclusão, alteração ou remoção do código fonte dos casos de testes automáticos. A ação *Code Editing*, similar a ação de *Test Editing*, abrange a inclusão, alteração e remoção de código fonte de produção.

Tabela 4.1. Ações do ciclo de vida de desenvolvimento.

Ação	**Definição**
Test Creation	Caracterizado pela criação de um caso de teste automático.
Test Pass	Caracterizado pela execução dos casos de teste que resultam em sucesso.
Test Failure	Caracterizado pela execução dos casos de teste que resultam em falha.
Test Editing	Caracterizado pela inclusão/alteração/remoção de código fonte de teste.
Code Editing	Caracterizado pela inclusão/alteração/remoção de código fonte de produção.

Fonte: Autoria própria.

4.1.2. Categorias

O novo modelo de heurísticas é formado por 6 categorias e 16 tipos de episódios. O modelo foi construído devido a evolução das demais ferramentas de análise do processo do TDD. Cada episódio é representado pela sequência de ações que poderão ser classificadas utilizando as heurísticas definidas, conforme descrito na Tabela 4.2. Durante a definição das heurísticas foi necessário avaliar detalhadamente o ciclo de vida do desenvolvimento iterativo de software. A ferramenta contempla as seguintes categorias:

- *Test Addition* (TA)
- *Test-first* (TF)
- *Test-last* (TL)
- *Refactoring* (RF)
- *Test Driven Development* (TDD)
- *Unknown* (UK)

Tabela 4.2. Heurísticas utilizadas para inferir a classificação do ciclo de desenvolvimento.

Tipo	**Definição**
Test Addition	TA1. Test Creation → Test Pass TA2. Test Creation → Test Failure → Test Editing → Test Pass
Test-first	TF1. Test Creation → Code Editing → Test Pass TF2. Test Creation → Test Failure → Code Editing → Test Pass TF3. Test Creation → Code Editing → Test Failure → Code Editing → Test Pass
Test-last	TL1. Code Editing → Test Creation → Test Pass TL2. Code Editing → Test Creation → Test Failure → Test Editing → Test Pass
Refactoring	RF1. Code Editing → Test Pass RF2. Code Editing → Test Failure → Code Editing → Test Pass RF3. Test Editing → Test Pass RF4. Test Editing → Test Failure → Test Editing → Test Pass
Test Driven Develop-ment	TDD1. Test Creation → Test Failure → Code Editing → Test Pass TDD2. Test Creation → Test Failure → Code Editing → Test Pass → Test Editing → Test Pass TDD3. Test Creation → Test Failure → Code Editing → Test Pass → Test Editing → Test Failure → Test Editing → Test Pass TDD4. Test Creation → Test Failure → Code Editing → Test Pass → Code Editing → Test Failure → Code Editing → Test Pass
Unknown	UK1. None of the above → Test pass

Fonte: Autoria própria.

Test Addition

Compreende a adição de novos casos de teste. Nessa categoria não há alteração em código fonte de produção, somente em código fonte de testes. Os possíveis fluxos são vistos na Figura 4.1.

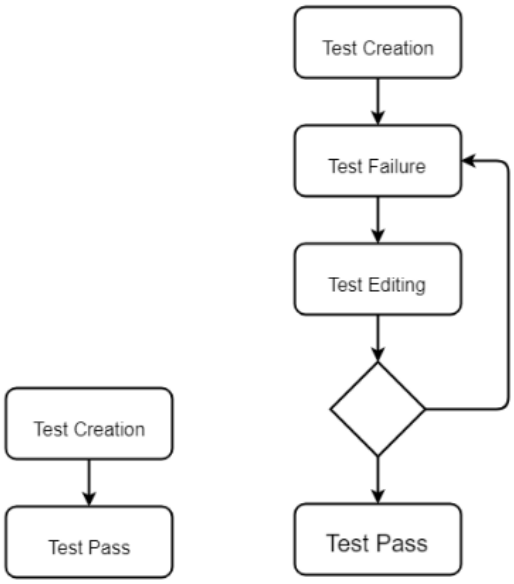

Figura 4.1. Fluxo do Test Addition.

Fonte: Autoria própria.

Test-first

Nesta categoria, o caso de teste deve ser criado antes da criação do código de produção. Os possíveis fluxos são demonstrados na Figura 4.2.

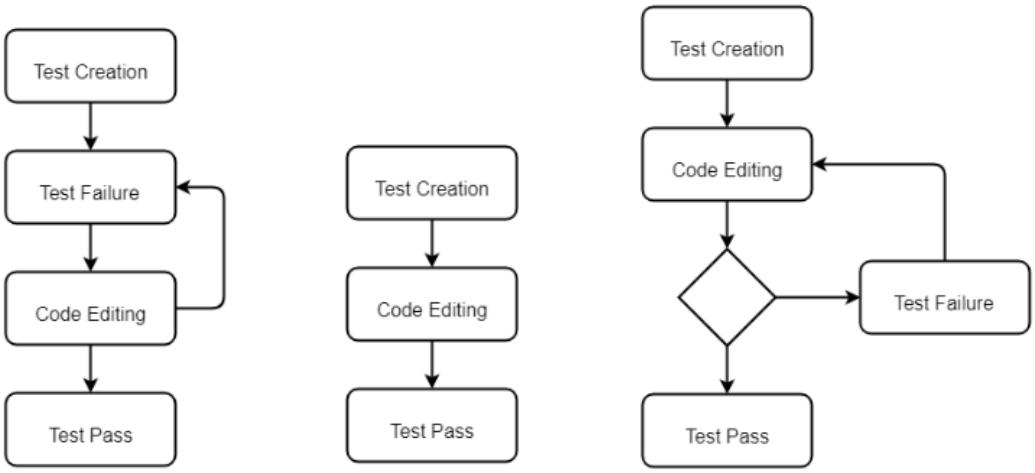

Figura 4.2. Fluxo do Test-first.

Fonte: Autoria própria.

Test-last

Nesta categoria, o caso de teste deve ser criado após a criação do código de produção. Os possíveis fluxos são demonstrados na Figura 4.3.

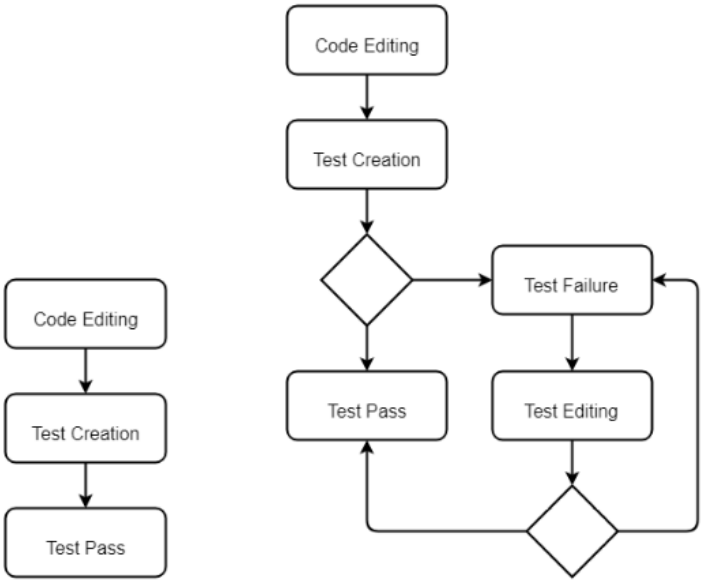

Figura 4.3. Fluxo do Test-last.

Fonte: Autoria própria.

Refactoring

Nesta categoria, é possível realizar a refatoração para código fonte de produção ou de teste. Compreende-se pela melhoria do código fonte, seja ela realizada no código fonte de produção ou teste. Os possíveis fluxos são apresentados na Figura 4.4.

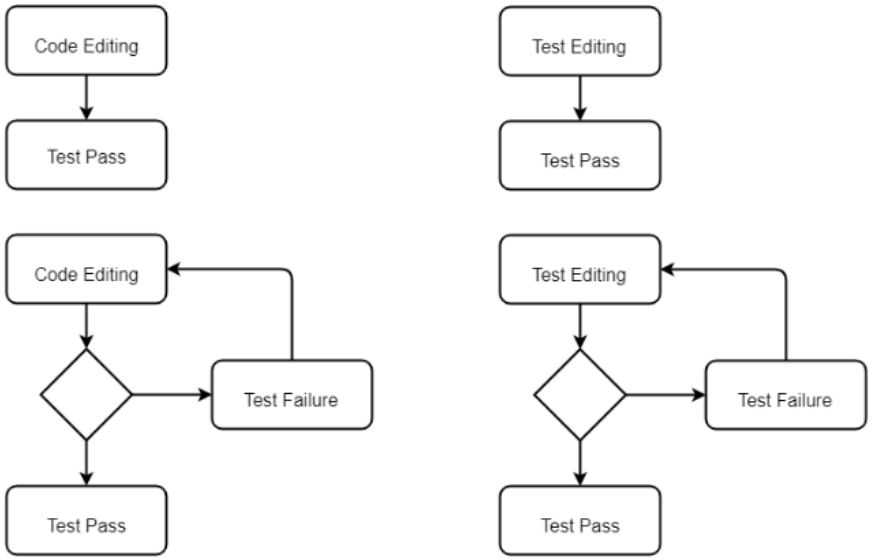

Figura 4.4. Fluxo da Refactoring.

Fonte: Autoria própria.

Test Driven Development

Nesta categoria, é necessário realizar a criação do caso de teste antes da criação do código fonte de produção. Após a criação do código de produção é necessário realizar o refatoração do código fonte de produção e teste. Os possíveis fluxos são apresentados na Figura 4.5.

Unknown

Para que essa categoria seja classificada pela ferramenta, nenhuma das categorias seja alcançada. Dessa forma, tudo que não contempla as heurísticas da Tabela 4.2 será classificado como *Unknown*.

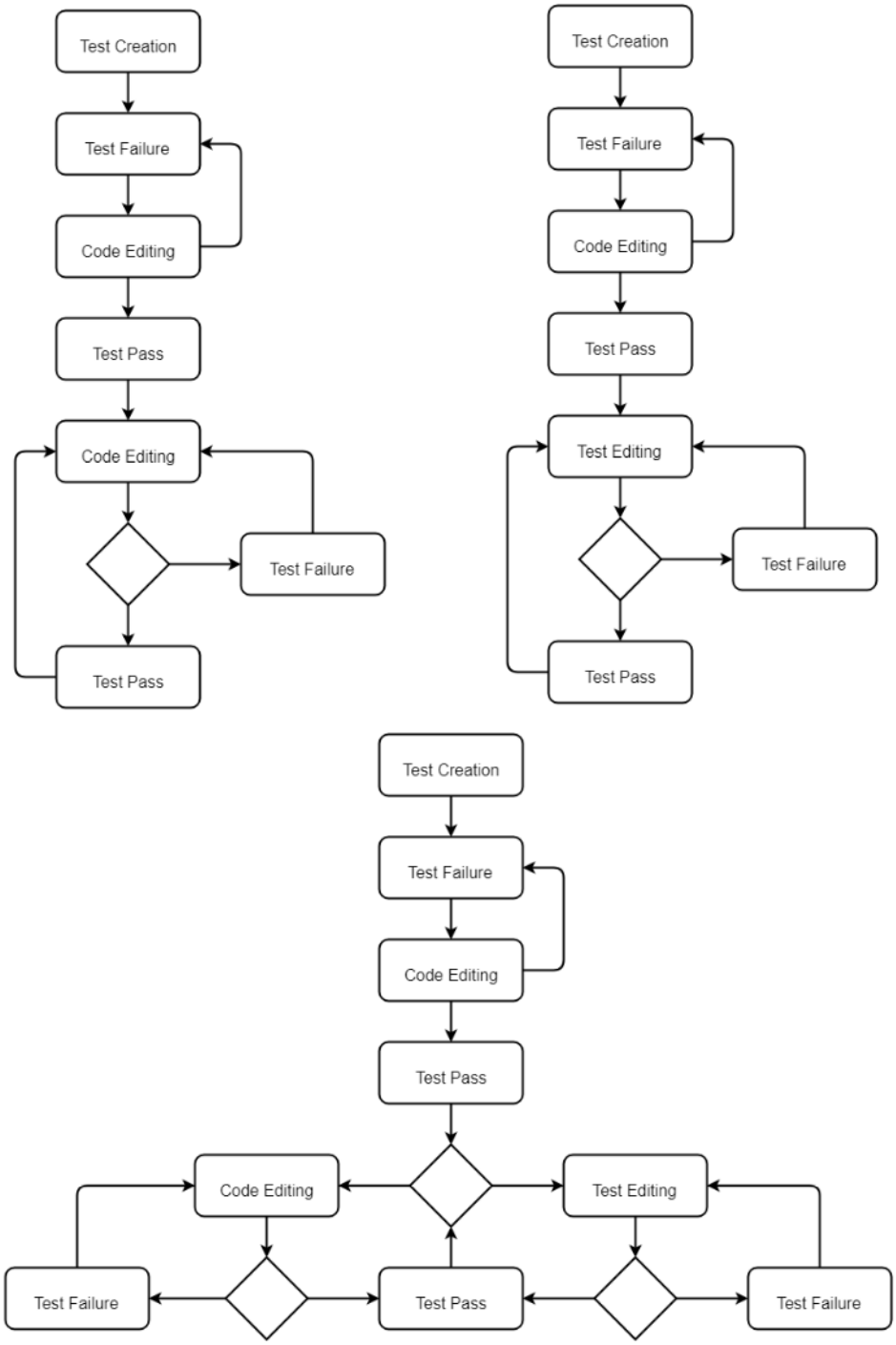

Figura 4.5. Fluxo do Test Driven Development.

Fonte: Autoria própria.

4.1.3. Ferramenta

As classificações realizadas pela ferramenta são apresentadas de forma similar àquela da Figura 4.6, tais como *Test Addition*, *Test-first*, *Test-last*, *Test Driven Development* e *Refactoring*. Podemos constatar na imagem como as ações do usuário são representadas e como cada classificação é exibida para o usuário da ferramenta.

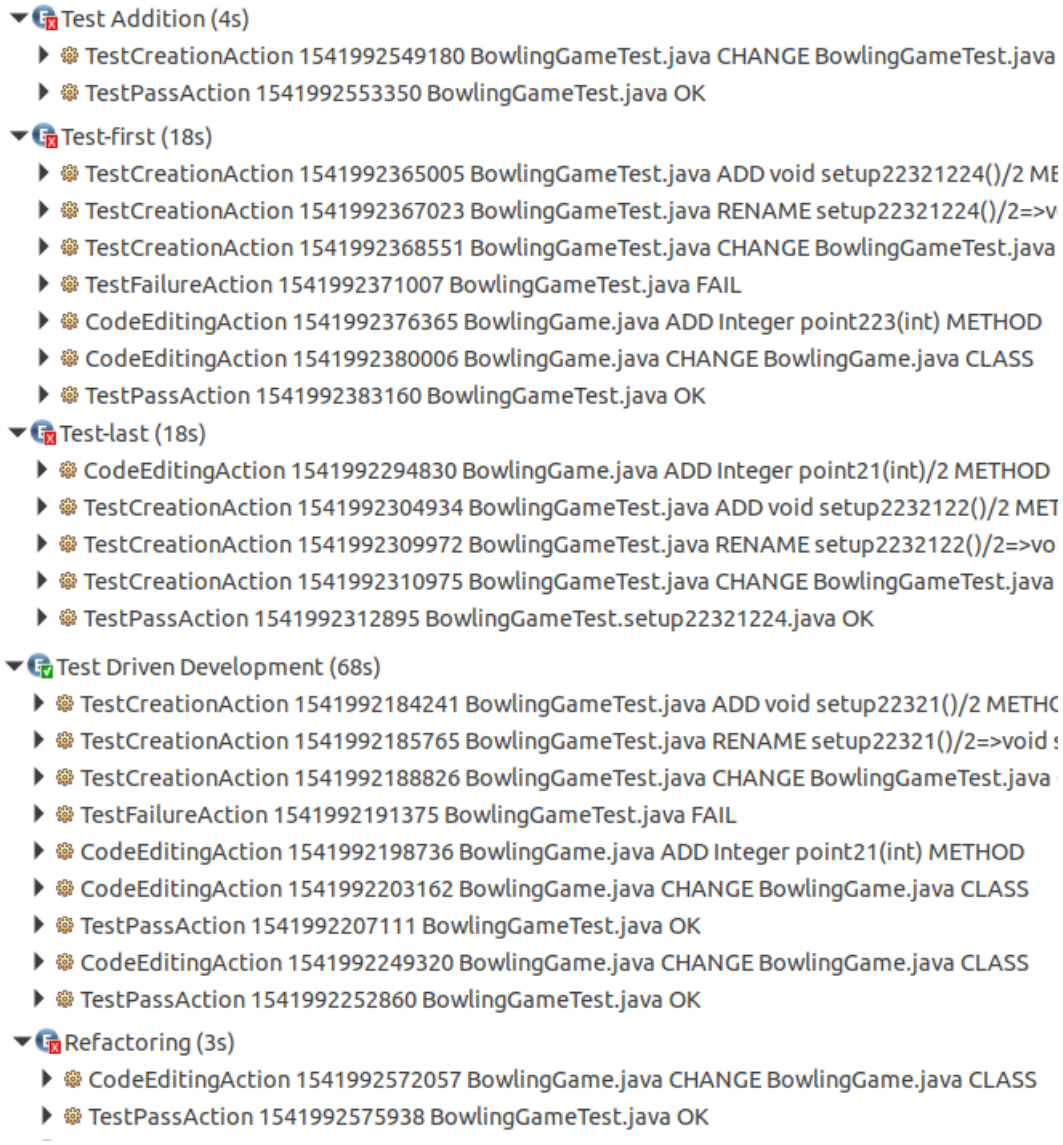

Figura 4.6. Classificações realizadas pela Butterfly.

Fonte: Autoria própria.

4.1.4. Avaliação

Conduzimos um estudo experimental (MATOS et al., 2018b) para avaliar a ferramenta em questão. O estudo contemplou um estudo de caso onde desenvolvedores de software realizaram a construção de um marcador de pontos de um jogo de boliche, comumente chamado

de *Bowling Game*, utilizando TDD. Contamos com a participação de sete desenvolvedores, onde cada um usou o tempo necessário para construir o marcador. A ferramenta conseguiu classificar todas as categorias esperadas, sendo elas: Test Addition, Test-first, Test-last, Test Driven Development, Refactoring e Unknown. Com a execução foi possível avaliar a ferramenta, analisando a corretude da mesma, além de garantir que a métrica M1 citada na Tabela 3.1, referentes ao percentual de conformidade do processo do TDD, seja coletada de forma eficaz.

4.1.5. Trabalhos Relacionados

Considerando a evolução da ferramenta Butterfly frente a outras ferramentas existentes, demonstramos as principais características na Tabela 4.3. As características são comparadas com as ferramentas Zorro (KOU et al., 2010) e Besouro (BECKER et al., 2015).

- Propósito: Tem como propósito mensurar a conformidade e avaliar o ciclo de vida do TDD, o que não era possível em outras ferramentas. Com a ferramenta é possível verificar se o desenvolvedor está utilizando corretamente todas as fases do TDD, ou, se está apenas utilizando fases do TDD, como *Test-first* e *Refactoring*.
- Dependências: Na ferramenta Zorro e Besouro, existem dependências que são fundamentais para o funcionamento da ferramenta. A dependência do *Jess* era uma das

Tabela 4.3. Comparação de sistemas de reconhecimento automático do processo do Desenvolvimento Orientado por Teste (TDD)

Ferramenta	Propósito	Dependências	Avaliação de Conformidade	Feedback do Usuário	Relatório de Conformidade
Zorro (KOU et al., 2010)	Conformidade de TDD	Hackystat, SDSA, Jess	Conformidade sensível ao contexto	Não	Não
Besouro (BECKER et al., 2015)	Construção de sistemas para explorar e comparar definicões operacionais	Listeners, Jess, VCS	Variando, de acordo com o componente implementado	Sim	Não
Butterfly (MATOS et al., 2018b)	Conformidade e avaliação do ciclo de vida do TDD	Listener, VCS	Implementação padrão de acordo com heurísticas pré estabelecidas	Sim	Sim

Fonte: Autoria própria.

principais a serem utilizadas, porém não é uma dependência gratuita para utilização nas indústrias de software, sendo gratuita apenas no âmbito acadêmico. De outro lado, a ferramenta Butterfly não possui dependências que impossibilitem a utilização da mesma na indústria de software, além de ser compartilhada com a comunidade.

- Avaliação de Conformidade: Passou a ser voltada às ações conhecidas na literatura e não mais variando de acordo com contextos ou implementações como realizado nas ferramentas Zorro e Besouro.
- Feedback do Usuário: Tanto a ferramenta Besouro quanto a ferramenta Butterfly possuem a possibilidade de classificação de ações que não são consideradas corretas pelo usuário, possibilitando futuramente a melhoria da ferramenta.
- Relatório de Conformidade: Uma melhoria na ferramenta Butterfly é a sumarização das ações executadas pelo usuário. Com esse relatório é possível analisar o percentual de utilização do TDD de acordo com as ações do usuário.

A ferramenta Butterfly está disponível no Github[1], livre para o uso da comunidade, que permite também novas funcionalidades, visto que a ferramente possui código fonte aberto.

4.2. Execução do Estudo Experimental

De acordo com o desenho experimental e a abordagem do TDD em conjunto com critérios de teste funcional descritos no Capítulo 3, descrevemos os passos realizados durante a execução do estudo de caso, como ocorreu o treinamento dos participantes e detalhes das atividades executadas.

4.2.1. Participantes

Os participantes utilizaram o ambiente de desenvolvimento com Java na versão 13, IDE Eclipse, JUnit 5 e Butterfly. O estudo foi conduzido dentro de uma organização com foco em desenvolvimento de software com cerca de 550 profissionais da área da Tecnologia da Informação (TI) localizada no noroeste do Paraná. Dentre esses profissionais, 10 (dez) participaram. Os participantes pertencem a uma equipe de profissionais que estão cursando ou já concluíram cursos superiores de TI, como Análise e Desenvolvimento de Sistemas, Ciências da Computação, Sistemas de Informação, Informática e Engenharia de Software. Os profissionais participantes possuem funções de Analistas de Sistemas e Arquitetos de Software dentro da organização. Todos possuem conhecimento em linguagem Java, IDE Eclipse e ferramentas de automatização de teste.

Da amostra escolhida, 6 (seis) participantes responderam com a implementação do Bowling Game Microservice, 1 (um) dos participantes foi removido, devido o código-fonte

[1] https://github.com/altitdb/butterfly

entregue constar como incompleto e não implementava as regras estabelecidas pelo estudo. Dado os 5 (cinco) participantes, analisamos a consistência interna da amostra de participantes escolhida. Calculamos o α de Cronbach para cada pergunta com escala Likert. Como resultado para esta amostra de participantes, temos o valor de *0.85* para o α de Cronbach, valor que é representado como *"Bom"* de acordo com a faixa de confiabilidade do coeficiente de α de Cronbach, onde $0.80 \leq \alpha \leq 0.89$ (CRONBACH, 1951). Ainda sobre a classificação dos participantes, em ambos os grupos contamos com características semelhantes, como Desenvolvedores e Arquitetos em ambos os grupos.

Dos participantes selecionados, todos ocupam cargos/funções de Arquiteto de Software ou Desenvolvedor. Desta forma, selecionamos Arquitetos de Software e Desenvolvedores para os dois grupos criados, conforme Tabela 4.4. Os Arquitetos de Software são responsáveis por garantir que o time de Desenvolvedores produza software de qualidade, por meio de uma arquitetura evolutiva. Os Desenvolvedores são responsáveis por transformar requisitos de software em produtos de software, analisando, colaborando e mantendo o software desde a fase inicial até a entrega em produção.

Tabela 4.4. Cargo/função dos participantes.

Grupo	Participante	Qual é o seu cargo/sua função?
Com critérios	Participante 01	Arquiteto de Software
	Participante 02	Desenvolvedor
	Participante 03	Desenvolvedor
Sem critérios	Participante 04	Desenvolvedor
	Participante 05	Arquiteto de Software

Fonte: Autoria própria.

Na Tabela 4.5 apresentamos o conhecimento em linguagem de programação Java. Dos participantes, todos possuem conhecimento em linguagem Java. É evidente que os participantes que ocupam posições de Arquitetos de Software possuem maior tempo de experiência, entre 9 e 10 anos, em programação Java. Os Desenvolvedores ocupam a faixa de 4 ou 5 anos de experiência em programação Java. Unanimente, todos os participantes utilizam Java com frequência em seu trabalho.

Tabela 4.5. Conhecimento em linguagem de programação Java.

Grupo	Participante	Qual é seu nível de conhecimento na linguagem de programação Java?	Você possui quantos anos de experiência de programação em Java?	Com que frequência você utiliza Java no seu trabalho?
Com critérios	Participante 01	5	10	5
	Participante 02	4	05	5
	Participante 03	5	04	5
Sem critérios	Participante 04	5	05	5
	Participante 05	5	09	5

Fonte: Autoria própria.

Questionamos os participantes também sobre seu conhecimento em testes automatizados conforme apresentado na Tabela 4.6. Todos participantes possuem conhecimento em testes automatizados, onde ambos os grupos ficaram com média 4 do nível de conhecimento, de uma escala de 1 a 5. Todos os participantes utilizam testes automatizados com frequência no seu ambiente de trabalho, sendo que a maioria utiliza testes automatizados com muita frequência. Além desse conhecimento, todos os participantes utilizam como ferramenta de teste automatizado o *JUnit*. Outras ferramentas utilizadas pelos participantes, porém, não unimamente são: *Cucumber* (ferramenta que auxília na utilização de Behavior-driven development (BDD)), *Selenium* (*framework* para teste de aplicações web), *Mockito* (*framework* para utilização de *mocks* em conjunto com *JUnit*), *PowerMockito* (*framework* para utilização de *mocks* em conjunto com *JUnit*), *SpringTest* (*framework* que fornece diversas abstrações para simplificar a escrita de testes automatizados), *Maven* (ferramenta de gerenciamento e compreensão de projetos de software), *Azure Devops* (software de colaboração para desenvolvimento de software), *AssertJ* (biblioteca java que fornece um rico conjunto de asserções, mensagens de erro úteis, melhora a legibilidade do código de teste), entre outros dependendo do *framework* base.

Tabela 4.6. Conhecimento em testes automatizados.

Grupo	Participante	Qual é o seu nível de conhecimento em testes automatizados?	Com que frequência você utiliza testes automatizados em seu trabalho?
Com critérios	Participante 01	5	5
	Participante 02	3	4
	Participante 03	4	3
Sem critérios	Participante 04	5	5
	Participante 05	3	5

Fonte: Autoria própria.

Na Tabela 4.7 apresentamos o conhecimento dos participantes sobre Desenvolvimento Orientado por Teste (TDD). Dos participantes somente 1 (um) deles possuía conhecimento alto em TDD, 3 (três) participantes possuíam conhecimento moderado e 1 (um) possuía conhecimento baixo em TDD. Dos 5 (cinco) participantes, 2 (dois) utilizam TDD de forma moderada e 3 (três) utilizam pouco.

Tabela 4.7. Conhecimento em Desenvolvimento Orientado por Teste (TDD).

Grupo	Participante	Qual é o seu conhecimento em Desenvolvimento Orientado por Teste (TDD)?	Com que frequência você utiliza o Desenvolvimento Orientado por Teste (TDD) em seu trabalho?
Com critérios	Participante 01	5	3
	Participante 02	2	2
	Participante 03	3	2
Sem critérios	Participante 04	3	2
	Participante 05	3	3

Fonte: Autoria própria.

Analisamos também o conhecimento sobre critérios de teste dos participantes e apresentamos na Tabela 4.8. Os participantes do grupo “Sem critérios” não possuíam conhecimento em critérios de teste. Como estes participantes não possuem conhecimento, logo, eles não utilizam Critérios em seu trabalho. Já os participantes do grupo “Com critérios” possuíam conhecimento moderado ou alto e utilizam sempre, muito ou muito frequentemente em seu trabalho. Eles também conhecem critérios de teste funcionais, estruturais e baseados em erro.

Tabela 4.8. Conhecimento em critérios de teste.

Grupo	Participante	Qual é o seu conhecimento em critérios de teste?	Com que frequência você utiliza critérios de teste em seu trabalho?
Com critérios	Participante 01	5	5
	Participante 02	3	4
	Participante 03	3	3
Sem critérios	Participante 04	1	1
	Participante 05	1	1

Fonte: Autoria própria.

Por fim, verificamos o conhecimento em refatoração dos participantes, demonstrado na Tabela 4.9. Os participantes possuem conhecimento entre moderado e alto. Dos participantes, somente 1 (um) utiliza às vezes, 1 (um) utiliza muito e 3 (três) utilizam muito frequentemente a refatoração em seu ambiente de trabalho.

Tabela 4.9. Conhecimento em refatoração.

Grupo	Participante	Qual é o seu conhecimento sobre refatoração?	Com que frequência você faz refatoração do código em seu trabalho?
Com critérios	Participante 01	5	5
	Participante 02	4	2
	Participante 03	5	5
Sem critérios	Participante 04	3	4
	Participante 05	4	5

Fonte: Autoria própria.

4.2.2. Treinamento

Os participantes selecionados assistiram a um treinamento que contemplou o seguinte conteúdo: noções básicas de teste de software, critérios de teste, TDD, estratégias de desenvolvimento de software e a utilização da ferramenta **Butterfly**. Depois do conteúdo ministrado, foi realizado um exercício prático, que consistiu na implementação do *Bowling Game* utilizando TDD em conjunto com critérios de teste juntamente com a ferramenta **Butterfly**. O exemplo da implementação pode ser encontrado no Github[2] e os requisitos implementados na Tabela 4.10. Foi possível observar que os participantes que participaram

[2] https://github.com/altitdb/tdd/tree/master/bowling-game-training

da etapa de treinamento obtiveram resultados satisfatórios, ou seja, o conhecimento foi comprovado. Dessa forma, foi possível validar que só participaram do experimento indivíduos que possuíam proficiência em TDD, critérios de teste e na ferramenta **Butterfly**. Os slides do treinamento estão disponíveis no Github[3]. O treinamento durou cerca de 5 horas e contou com a participação de 10 profissionais.

Tabela 4.10. Requisitos do *Bowling Game*.

Requisito	Descrição
R1	A pontuação do jogo pode ser consultada a qualquer momento.
R2	O jogador tem direito a duas jogadas para atingir a pontuação máxima (10) em cada rodada.
R3	O jogo consiste em 10 rodadas.
R4	Se no primeiro tiro a pontuação máxima for atingida (*strike*), o jogador não terá direito ao segundo tiro.
R5	Se em ambas as tacadas a pontuação máxima for atingida (*spare*), o jogador terá como bônus a pontuação obtida na próxima jogada.
R6	O bônus de pontuação do *strike* é o valor dos próximos dois movimentos.
R7	O jogador terá duas jogadas extras se fizer um *strike* na décima rodada.
R8	Se ele atingir um *spare* nas duas jogadas após a décima rodada, o jogador terá direito a um tiro extra.

Fonte: Autoria própria.

Antes da execução do estudo, foi necessária a configuração do ambiente de desenvolvimento, onde, era necessária a instalação da IDE Eclipse e a ferramenta Butterfly. Para instalação da IDE Eclipse não necessitou-se de vídeos ou tutoriais, uma vez que, os participantes possuem experiência com a IDE. Para a ferramenta Butterfly, elaboramos vídeos na demonstram como realizar a instalação, configuração e utilização da ferramenta Butterfly. Os links estão disponíveis em:

- Instalando a Butterfly: https://www.youtube.com/watch?v=wcHey80JDYg
- Usando a Butterfly: https://www.youtube.com/watch?v=vt5JMlEdOA0

4.2.3. Grupos

Para a execução do estudo experimental, realizamos a caracterização dos participantes por meio de um questionário descrito na Seção 4.2.1, que visou conhecer melhor os participantes. Dada esta caracterização, realizamos a classificação dos participantes, onde

[3] https://github.com/altitdb/utfpr/blob/master/software-testing-training.pdf

classificamos em participantes com conhecimento em critérios de teste e participantes sem conhecimento em critérios de teste. Os grupos foram formados por conveniência, buscando balancear os grupos, onde selecionamos os participantes de acordo com o conhecimento e/ou experiência em linguagem de programação, testes automatizados, Desenvolvimento Orientado por Teste (TDD), critérios de teste e refatoração. Formamos 2 (dois) grupos com 3 (três) participantes cada.

4.2.4. Técnica de Teste de Software

Na execução experimental foi utilizada a estratégia descrita na Seção 3.2, na qual propõe a utilização do TDD em conjunto com critérios de teste. A estratégia empregada está definida na Figura 3.3 e novamente apresentada na Figura 4.7. A estratégia compreende adicionar critérios de teste na fase de criação de casos de teste (ou seja, na fase 1. *red*) no processo do TDD.

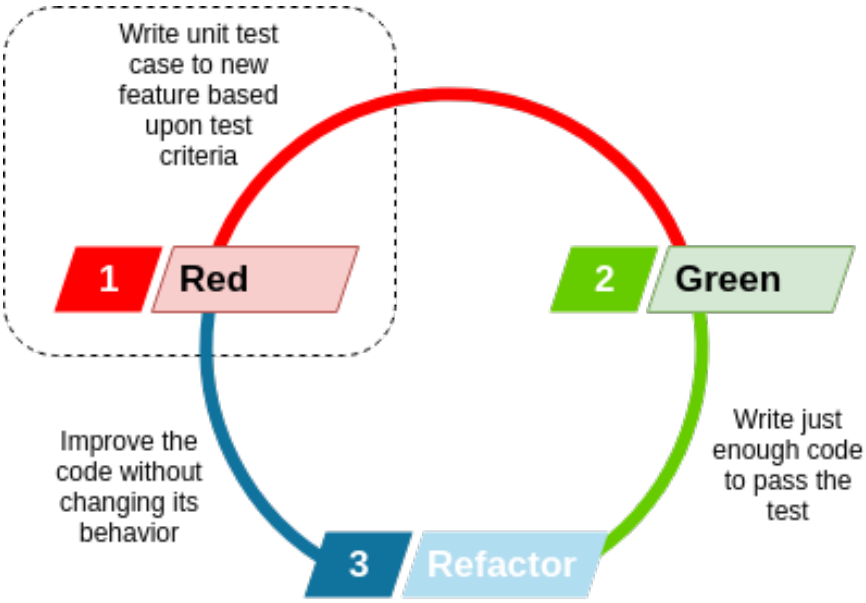

Figura 4.7. Fluxo do Processo do Desenvolvimento Orientado por Teste (TDD) utilizando critérios de teste.

Fonte: Autoria própria.

Os critérios de teste utilizados na fase *1. red* (vide Figura 4.7) foram as técnicas funcionais de particionamento em classes de equivalência e análise do valor limite. Como o foco do estudo para avaliar o uso do TDD em conjunto com critérios de teste, descrevemos os requisitos de teste obtidos com a aplicação dos critérios de teste a serem utilizados pelos participantes do estudo experimental.

Particionamento em classes de equivalência

Na Tabela 4.11 são apresentadas as classes de equivalência derivadas dos requisitos de software R1, R2 e R3 do *Bowling Game* descritos na Tabela 4.10. Estes requisitos foram utilizados no treinamento e também no estudo experimental.

Tabela 4.11. Classes de equivalência.

Condições de Entrada	Classes Válidas	Classes Inválidas
Pontuação total	$0 >= pt <= 100$ (1)	$pt < 0$ (2) $pt > 100$ (3)
Pontuação da primeira jogada	$0 >= p1 <= 10$ (4)	$p1 < 0$ (5) $p1 > 10$ (6)
Pontuação da segunda jogada	$0 >= (p1 + p2) <= 10$ (7)	$(p1 + p2) < 0$ (8) $(p1 + p2) > 10$ (9)
Quantidade de rodadas	$0 >= qr <= 10$ (10)	$qr < 0$ (11) $qr > 10$ (12)

Fonte: Autoria própria.

Análise de valor limite

Na Tabela 4.12 são apresentados os valores limites derivados das classes de equivalência descritos na Tabela 4.11, onde para cada limite temos a indicação da classe de equivalência e requisito de teste entre parenteses. Conforme mencionado anteriormente, foram descritas todas as classes de equivalência dos requisitos R1, R2 e R3 do *Bowling Game*.

Tabela 4.12. Análise de valor limite.

Condições de Entrada	Limites Válidos	Limites Inválidos
Pontuação total	$pt = 0$ (1,13) $pt = 100$ (1,14)	$pt =$ -1 (2,15) $pt = 101$ (3,16)
Pontuação da primeira jogada	$p1 = 0$ (4,17) $p1 = 10$ (4,18)	$p1 =$ -1 (5,19) $p1 = 11$ (6,20)
Pontuação da segunda jogada	$p1 + p2 = 0$ (7,21) $p1 + p2 = 10$ (7,22)	$p1 + p2 =$ -1 (8,23) $p1 + p2 = 11$ (9,24)
Quantidade de rodadas	$qr = 0$ (10,25) $qr = 10$ (10,26)	$qr =$ -1 (11,27) $qr = 11$ (12,28)

Fonte: Autoria própria.

Casos de Teste

Na Tabela 4.13 são apresentados os casos de teste derivados a partir dos critérios de teste funcional de particionamento em classes de equivalência e análise do valor limite.

Dependendo da implementação do *Bowling Game* alguns requisitos de teste não serão executados, como por exemplo nesta implementação disponível no Github[4]. Estes casos de teste foram utilizados para analisar a cobertura dos requisitos de teste funcional (M2).

Tabela 4.13. Casos de Teste

Caso de Teste	Entrada	Saída	Requisitos de Teste
CT-01	Nenhuma tacada	0 pontos, 0 rodada	1, 10, 25
CT-02	*p1* = -1	Pontuação inválida	2, 5, 19
CT-03	*p1* = 11	Pontuação inválida	3, 6, 20
CT-04	*p1* = 0 e *p2* = -1	Pontuação inválida	2, 8, 23
CT-05	*p1* = 0 e *p2* = 11	Pontuação inválida	3, 9, 24
CT-06	*p1* = 0 e *p2* = 10	10 pontos, 1 rodada	1, 4, 7, 22
CT-07	20 tacadas de 0 pontos	0 pontos, 10 rodadas	1, 4, 7, 10, 13, 17, 21, 26
CT-08	10 tacadas de 10 pontos	100 pontos, 10 rodadas	1, 4, 10, 14, 18, 26
CT-09	11 tacadas de 10 pontos	Rodada inválida	3, 12, 16, 28

Fonte: Autoria própria.

4.2.5. Atividade Desenvolvida

O participantes desenvolveram uma versão avançada do *Bowling Game*, descrita na Seção 3.3.3, utilizando definições utilizadas no contexto industrial, possibilitando que a atividade desenvolvida tivesse o mesmo grau de dificuldade das entregas realizadas pelos colaboradores da empresa no dia a dia. O exemplo da atividade desenvolvida pode ser encontrada no Github[5], no *Bowling Game* de referência. Nesse exemplo, todas as características e dificuldades são semelhantes às do contexto industrial. No exemplo é possível ver como são implementadas uma arquitetura em camadas, arquitetura REST e acesso a banco de dados relacional utilizando Java 13 e Spring Boot. Nessa aplicação são disponibilizados *endpoints* REST que permitem a entrada e saída de dados utilizando JSON[6]. São implementadas as regras de negócio R1, R2 e R3 descritas na Tabela 4.10. Os dados manipulados pela aplicação são armazenados em banco de dados relacional. Para cada camada da aplicação são implementados testes automatizados a fim de validar os requisitos de software.

Avaliamos também o *Bowling Game* de referência com as mesmas ferramentas na qual avaliamos os desenvolvimentos dos participantes. Constatamos que não foram gerados *code smells* e não foram criados defeitos no *Bowling Game* de referência. Em relação ao percentual de cobertura de linhas, obtivemos o valor de 74,8%. Para o percentual de cobertura de

[4] https://github.com/altitdb/tdd/tree/master/bowling-game-microservice
[5] https://github.com/altitdb/tdd/tree/master/bowling-game-reference
[6] https://www.json.org/

mutação obtivemos o valor de 70,0%. Tanto para o percentual de cobertura de linhas quanto para o percentual de cobertura de mutação não obtivemos o valor de 100% de cobertura. Em análise mais detalhada, verificamos que a cobertura de 100% não ocorreu pois existiam código fonte gerado para suprir as necessidades da linguagem, de forma mais clara, houve a necessidade de utilizar os métodos *equals*, *hashCode* e *toString* da linguagem de programação Java. Excluindo o código fonte escrito por necessidade da linguagem, atingiríamos o percentual de cobertura de linhas e de mutação de 100%.

4.2.6. Análise dos Resultados

A análise dos resultados deste experimento foi feita com base no paradigma GQM definido na Seção 3.2. A análise iniciou-se pela conferência da conformidade do processo do TDD. Em seguida, foi feita a análise da qualidade do software desenvolvido pelos participantes. Todos os dados e *scripts* estão disponíveis no Github[7].

Conformidade do Processo do TDD

Durante a análise dos dados foi possível notar que os participantes inseridos no contexto industrial não seguiram o fluxo fielmente do TDD de acordo com as heurísticas estabelecidas na Tabela 4.2. Os principais diferenciais percebidos foram a ausência da execução dos casos de teste após a escrita de um novo Caso de Teste, não demonstrando a fase "Red" do processo do TDD, e a alteração do Caso de Teste após realizar o processo de codificação da funcionalidade, logo após criar um novo Caso de Teste. Tais ações ocorreram quando haviam casos de testes incompletos ou errados. Retiramos a sequência de ações "*TestCreationAction -> CodeEditingAction -> TestEditingAction -> TestFailureAction -> CodeEditingAction -> TestPassAction*" da execução dos participantes. Nesta sequência podemos verificar a ausência da execução do casos de teste após a criação de um novo caso de teste e também a edição de um caso de teste novo após o início da implementação da funcionalidade. Dado tais circunstâncias, realizamos a análise manual das ações dos participantes, onde entendemos que estas ações se encaixam no processo do TDD, onde obtemos o percentual de utilização do TDD (M1). O cálculo para obter o percentual de utilização do TDD foi realizado utilizando o total de episódios com TDD (TET) e total de episódios (TE) registrados pela ferramenta Butterfly, onde temos:

$$M1 = TET * 100/TE \tag{4.1}$$

Analisamos os casos de testes escritos pelos participantes, confrontando os casos de teste presentes na Tabela 4.13. A partir disso obtemos o percentual de cobertura dos requisitos de teste funcional (M2). O cálculo para obter o percentual de cobertura dos

[7] https://github.com/altitdb/utfpr/tree/master/estudo-experimental-tdd-ct

requisitos de teste funcional foi realizado utilizando o total de requisitos de teste funcional cobertos pelos participantes (RTC) divididos pelo total de requisitos de teste funcional (TRT) sem os requisitos de teste funcional não alcançáveis (RTNA), conforme apresentado na Equação (4.2).

$$M2 = RTC * 100/(TRT - RTNA) \tag{4.2}$$

Na Tabela 4.14 são apresentados os casos de testes escritos pelos participantes e seus respectivos requisitos de teste. Constatamos que os participantes que estavam no grupo "Com critérios" e possuíam maior conhecimento em critérios de testes atingiram maior percentual de cobertura dos requisitos de teste funcional. De acordo com a implementação do Bowling Game Microservice descrito na Seção 3.3.3, os requisitos de teste 11, 15 e 27 não são alcançáveis.

Tabela 4.14. Relação entre participantes e casos de teste.

Grupo	Participante	Casos de Teste	Requisitos de Teste Cobertos
Com critérios	Participante 01	CT-01, CT-02, CT-03, CT-05, CT-09	1, 2, 3, 5, 6, 9, 10, 12, 16, 19, 20, 24, 25, 28
	Participante 02	CT-01, CT-02, CT-03, CT-05, CT-06, CT-07, CT-08	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 17, 19, 20, 21, 22, 23, 24, 25, 26
	Participante 03	CT-01, CT-08, CT-09	1, 3, 4, 10, 12, 14, 16, 18, 25, 26, 28
Sem critérios	Participante 04	CT-01, CT-09	1, 3, 10, 12, 16, 25, 28
	Participante 05	CT-06	1, 4, 7, 22

Fonte: Autoria própria.

Na Tabela 4.15 apresentamos os resultados das métricas M1 e M2 para responder a ***Q1:*** *Existe aumento da conformidade do processo do TDD em conjunto com critérios de teste?* do modelo GQM. Na tabela, apresentamos os dois grupos formados durante o estudo experimental e seus participantes. O percentual de utilização do TDD (M1) médio do grupo "Com critérios" foi de 69,14% contra o valor médio do grupo "Sem critérios" de 81,35%, diferença de 12,21%. O percentual de cobertura dos requisitos de teste funcional (M2) médio do grupo "Com critérios" foi de 20% contra o valor médio do grupo "Sem critérios" de 06%, diferença de 14%.

De acordo com o percentual de utilização do TDD (M1) e cobertura de dos requisitos funcionais (M2) conseguimos responder a ***Q1:*** *Existe aumento da conformidade do processo do TDD em conjunto com critérios de teste?*. Não constatamos aumento da conformidade do processo ao utilizar TDD em conjunto com critérios de teste, visto que ao utilizar critérios de teste o percentual de utilização do TDD foi 12,21% menor. O percentual médio de cobertura

Tabela 4.15. Resultado da conformidade do processo do TDD.

Grupo	Participante	Percentual de utilização do TDD (M1)	Percentual de cobertura dos requisitos de teste funcional (M2)
Com critérios	Participante 01	76.08	20
	Participante 02	66.66	28
	Participante 03	64.70	12
	Média	69.14	20
Sem critérios	Participante 04	93.75	08
	Participante 05	68.96	04
	Média	81.35	06

Fonte: Autoria própria.

dos requisito de teste funcional do grupo "Com critérios" foi de 20% contra 06% do grupo "Sem critérios". Concluímos que com a diferença do percentual de cobertura dos requisitos de teste funcional a utilização TDD teve aumento significativo de 14%.

Qualidade do Software com a utilização do TDD

Analisamos o número de *code smells* (M3) gerados durante o desenvolvimento da atividade, onde o grupo "Sem critérios" acabou produzindo uma quantidade maior que o grupo "Com critérios". O Participante 02 do grupo "Com critérios" gerou um número elevado de *code smells*, porém, analisando estes *code smells*, averiguamos que esta quantidade elevada corresponde a itens que podem ser retirados automaticamente pela IDE e outros pela falta do uso de refatoração. Os *code smells* deste participante poderiam ser removidos automaticamente pela IDE ou pelo uso de refatoração são: *Unnecessary imports should be removed*, *JUnit assertTrue/assertFalse should be simplified to the corresponding dedicated assertion* e *Unused "private" fields should be removed.*

Observamos o percentual de cobertura de linhas (M4) e percentual de cobertura de mutação (M5), onde não tivemos diferença significativa entre os grupos. De certa forma, não foi possível aumentar o percentual de cobertura de linhas e cobertura de mutação utilizando TDD em conjunto com critérios de teste. Para o número de defeitos, tivemos a diminuição de defeitos pelo grupo "Com critérios".

Na Tabela 4.16 apresentamos os resultados das métricas M3, M4, M5 e M6 para responder a ***Q2:*** *Existe aumento da qualidade do software com a utilização do TDD em conjunto com critérios de teste?* do modelo GQM. Na tabela, apresentamos os dois grupos formados durante o estudo. Para o grupo "Com critérios" tivemos uma média de 7 *code smells* por participante contra uma média de 5 *code smells* do grupo "Sem critérios".

Os *code smells* gerados pelos participantes são detalhados na Tabela 4.17. Os participantes 01 e 03 do grupo "Com critérios" não geraram *code smells.* Tivemos diversos *code smells* gerados pelos participantes, na qual o *code smell "Unnecessary imports should be removed"* teve destaque com 08 ocorrências.

Tabela 4.16. Resultado da qualidade do software com uso do TDD.

Grupo	Participante	Número de *code smells* (M3)	Percentual de cobertura de linhas (M4)	Percentual de cobertura de mutação (M5)	Número de defeitos (M6)
Com critérios	Participante 01	00	74.3	68	0
	Participante 02	21	64.7	48	3
	Participante 03	00	74.6	84	3
	Média	07	71.2	66	2
Sem critérios	Participante 04	04	85.60	83	3
	Participante 05	06	64.70	59	4
	Média	05	75.15	71	3.5

Fonte: Autoria própria.

Tabela 4.17. *Code smells* gerados pelos participantes.

Grupo	Participante	Qtde	*Code Smells*
Com critérios	Participante 02	08	Unnecessary imports should be removed
		04	JUnit assertTrue/assertFalse should be simplified to the corresponding dedicated assertion
		03	Unused “private” fields should be removed
		02	Boolean checks should not be inverted
		01	Lambdas should be replaced with method references
		01	Cognitive Complexity of methods should not be too high
		01	Assignments should not be made from within sub-expressions
		01	Generic exceptions should never be thrown
Sem critérios	Participante 04	02	Field names should comply with a naming convention
		02	Public constants and fields initialized at declaration should be “static final” rather than merely “final”
	Participante 05	02	Source files should not have any duplicated blocks
		01	Class variable fields should not have public accessibility
		01	Track uses of “TODO” tags
		01	Unused “private” methods should be removed
		01	Raw types should not be used

Fonte: Autoria própria.

Na Tabela 4.18 apresentamos os defeitos encontrados nas aplicações desenvolvidas pelos participantes. O participante 01 do grupo “Com critérios” não gerou defeitos. Os defeitos foram constatados após realizarmos a execução dos casos de teste apresentados na Tabela 4.13.

De acordo com o número de *code smells* (M3), percentual de cobertura de linhas (M4), percentual de cobertura de mutação (M5) e número de defeitos (M6), conseguimos

Tabela 4.18. Defeitos gerados pelos participantes.

Fonte: Autoria própria.

Grupo	Participante	Qtde	*Code Smells*
Com critérios	Participante 02	03	Não limita o jogo em 10 rodadas
			Não apresenta o número da rodada corretamente
			Não são retornados os dados corretamente quando feita somente a primeira tacada
	Participante 03	03	Não limita o jogo em 10 rodadas
			Não apresenta o número da rodada corretamente
			Não retorna a pontuação correta
Sem critérios	Participante 04	03	Não limita o jogo em 10 rodadas
			Não apresenta o número da rodada corretamente
			Não valida a pontuação da tacada (valores menores que 0 e maiores que 10)
	Participante 05	04	Não limita o jogo em 10 rodadas
			Não apresenta o número da rodada corretamente
			Não valida a pontuação da tacada (valores menores que 0 e maiores que 10)
			Pontuação calculada incorretamente

responder a ***Q2:*** *Existe aumento da qualidade do software com a utilização do TDD em conjunto com critérios de teste?*. Percebemos que houve um leve aumento de qualidade em relação ao número de *code smells* gerados quando se utiliza TDD em conjunto com critérios de teste, mesmo com a média do grupo "Com critérios" sendo maior que "Sem Critérios", tivemos apenas um participante que gerou *code smells* e acreditamos isso devido ao fato do participante não ter conhecimento em refatoração e não utilizar refatoração com frequência no seu trabalho, diferente dos outros participantes. Por outro lado, não percebemos aumento de qualidade em relação a cobertura de linhas e cobertura de mutação. Em relação ao número de defeitos, apenas um participante do grupo "Com critérios" não gerou defeitos, os demais geraram uma quantidade semelhante de defeitos.

Esforço dos Participantes

O objetivo de utilizar o instrumento NASA-TLX foi analisar o esforço percebido pelos desenvolvedores na utilização do TDD e TDD em conjunto com critérios de teste. Apresentamos os dados da análise de esforço dos participantes na Tabela 4.19.

Examinando o conjunto de dimensões, a mediana de todas as dimensões obteve o valor de 330 (de um total de 600). Com esse resultado, os participantes apontam que o esforço percebido durante o desenvolvimento da atividade teve grau de esforço moderado, valor desejado pelo estudo. Analisando os resultados de cada participante, os participantes relataram que a demanda mental foi acima do esperado, com um valor total de 500. Os participantes salientaram que seu desempenho não foi satisfatório e que foi necessário esforço considerável para desenvolvimento da atividade, onde ambos atingiram um valor total de

Tabela 4.19. Resultado da análise de esforço dos participantes.

Fonte: Autoria própria.

Participante	Demanda Mental	Demanda Física	Demanda Temporal	Desempenho	Esforço	Frustração
Participante 01	70	40	60	80	60	20
Participante 02	80	30	10	60	60	10
Participante 03	90	10	80	60	80	40
Participante 04	80	50	50	100	70	10
Participante 05	100	10	10	20	60	100
Participante 06	80	60	50	80	70	60
Mediana	80	35	50	70	65	30
Total	500	200	260	400	400	240

400. Verificamos que para os participantes a demanda temporal foi aceitável, com um valor total de 260. Avaliando a frustração dos participantes, não houve frustração desnecessária, resultando no valor de 240, valor considerado aceitável. Observamos que a demanda física dos participantes foi baixa, resultando no valor de 200. Concluímos que o esforço percebido foi aceitável, embora exista um esforço excessivo na demanda mental, desempenho e esforço. Cabe avaliar, em estudos futuros, a adequação das tarefas ou aperfeiçoamento das ferramentas para reduzir a demanda mental, desempenho e esforço necessário.

4.3. Ameaças a Validade

Considerando as ameaças a validade dos resultados baseou-se no *checklist* apresentado por Wohlin et al. (2000), que consiste de quatro ameaças: interna, externa, constructos e conclusão.

- **Validade Interna:** Considera-se que determinado aspecto investigado não possua riscos providos a partir de outros aspectos não considerados (WOHLIN et al., 2000). Com a utilização do modelo GQM elencou-se medidas fortemente ligadas e disponíveis para o estudo, na qual, tais medidas são necessárias para responder a questão de pesquisa deste estudo. De outro lado, a amostra da população foi cautelosamente escolhida, considerando o tempo de experiência, função exercida e conhecimentos sobre as técnicas e práticas utilizadas pelo estudo, além de todo o cuidado para o treinamento da mesma, evitando assim que ocorram efeitos indesejados. Contou com uma população formada por desenvolvedores e arquitetos de software, de uma organização de tamanho médio, que não possuem prática com TDD e critérios de teste.

Foi necessário descrever os papéis existentes dentro da organização, assimilando-os com funções conhecidas, conforme descrito na Seção 3.1, possibilitando a generalização em organizações que empregam métodos ágeis no desenvolvimento de software. Além dos participantes, assimilamos o Bowling Game, comumente utilizado em estudo com TDD, com características da indústria de software. De acordo com a organização existe a possibilidade de resultados melhores para TDD em conjunto com critérios de teste, não pela utilização de critérios de teste, mas pela melhor aplicação do TDD e mais conhecimento do software em desenvolvimento. No entanto, dada a natureza do desenvolvimento com Scrum, preservando o aspecto de comunhão do time, optou-se por esse design ao invés de designar, aleatoriamente, o tratamento para aplicação em cada atividade.

- **Validade Externa:** Destaca-se as preocupações que podem limitar a capacidade de generalizar as descobertas do estudo e quais descobertas são importantes para pessoas interessadas fora do estudo de caso investigado (WOHLIN et al., 2000). Estudos semelhantes foram conduzidos anteriormente na academia (Pachulski Camara; Graciotto Silva, 2016) e indústria (FUCCI et al., 2017; CAUSEVIC et al., 2013a; CAUSEVIC et al., 2013b; SANTOS et al., 2018; TOSUN et al., 2018) de software, além de estudos que interferem no fluxo do processo do TDD (Pachulski Camara; Graciotto Silva, 2016; CAUSEVIC et al., 2013a; SHELTON et al., 2012). Para que o estudo possibilitasse a generalização, seria necessário considerar mais participantes, de empresas diferentes e com software domínios mais diversificados, o que não foi possível neste momento durante a realização do estudo. Em trabalhos futuros, recomenda-se a adoção de exemplos utilizados pela indústria de software.
- **Validade dos Constructos:** Consiste em garantir que as medidas operacionais do estudo realmente representem o que está sendo investigado e possam responder a questão de pesquisa (WOHLIN et al., 2000). Utilizou-se a abordagem GQM para a definição do estudo experimental, uma vez que é possível definir objetivos, criar questões que atendam os objetivos definidos, além de ter medidas objetivas que respondam as questões levantadas. Dadas as medidas é possível mensurar os objetivos de forma quantificável, possibilitando ter respostas palpáveis para conclusão do estudo experimental.
- **Validade das Conclusões:** Busca-se obter resultados semelhantes, evitando que outro estudo executado de forma semelhante produza resultados diferentes (WOHLIN et al., 2000). Para que tais ameaças a validade sejam minimizadas, detalhamos todos os aspectos fundamentais do estudo. Descrevemos na Seção 3.1 o fluxo do processo na qual o estudo será aplicado, caracterizando os papéis envolvidos. Demonstrou-se a proposta de alteração no fluxo do TDD apresentado na Seção 3.2. Descrevemos o procedimento metodológico de forma detalhada na Seção 3.3, elencando os participantes do estudo, expondo as características do ambiente e da aplicação a ser utilizada pelos participantes

e na Seção 3.3.4 narrou-se a execução do estudo empírico. Buscou-se garantir o máximo de detalhes possibilitando que outros autores venham a replicar este estudo sem gerar resultados diferentes. Devido a quantidade reduzida de participantes não foi possível a realização de testes estatísticos de hipótese quanto às perguntas de pesquisa.

4.4. Considerações finais

Obtemos dois resultados principais neste capítulo: a ferramenta Butterfly e a avaliação da abordagem do TDD em conjunto com critérios de teste funcional. Considerando o estudo realizado com a ferramenta Butterfly, foi possível avaliá-la e analisar a corretude da mesma por meio de um estudo experimental. Com o apoio da ferramenta Butterfly, foi possível coletar a métrica M1 definida no modelo GQM na Seção 3.3, métrica que é referente ao percentual de utilização do TDD. Além do estudo experimental com a ferramenta Butterfly, realizamos outro estudo experimental relatado neste capítulo. O estudo experimental foi realizado com a finalidade de avaliar a estratégia proposta. Onde a estratégia propôs a utilização do TDD em conjunto com critérios de teste. Utilizamos critérios de testes funcionais de particionamento em classes de equivalência e análise do valor limite. Além disso, por meio do instrumento NASA-TLX, analisamos o esforço percebido pelos desenvolvedores durante a utilização do TDD e TDD em conjunto com critérios de teste.

Capítulo 5

Conclusões

Com este estudo foi possível avaliar o desenvolvimento de software juntamente com técnicas como TDD e critérios de teste funcional. Visando analisar os efeitos de critérios de teste durante o processo de desenvolvimento de software utilizando TDD. Por meio de um estudo empírico definido utilizando o modelo GQM foi possível estabelecer metas objetivas, além de questões que levam a atingir as metas definidas. As questões foram respondidas por medidas cautelosamente definidas. Dessa forma, respondemos a questão de pesquisa *Existem efeitos de melhoria no desenvolvimento de software durante a adoção do Desenvolvimento Orientado por Teste (TDD) e critérios de teste?* De acordo com as métricas estabelecidas, constatamos efeitos de melhoria no desenvolvimento de software. Em relação a conformidade do processo do TDD tivemos redução no percentual da utilização do TDD porém tivemos aumento do percentual da cobertura dos requisitos de teste. Em relação a qualidade do software com a utilização do TDD tivemos a redução do número de *code smells* e defeitos porém também foi reduzido o percentual de cobertura de linhas e cobertura de mutação. Sugerimos avaliações em cenários diferentes ou uso de aplicações desenvolvidas no dia-a-dia dos participantes, mais próximo dos requisitos típicos de empresas, ao invés de aplicações comumente utilizadas por estudos empíricos do TDD, como o *Bowling Game*.

Desenvolvemos a ferramenta Butterfly, ferramenta que tem como objetivo avaliar a conformidade do ciclo de desenvolvimento iterativo de software, por meio da utilização de heurísticas classificando em categorias no momento em que o desenvolvedor está utilizando. A ferramenta age de forma transparente e não afeta na produtividade dos desenvolvedores que a usam, além de disponibilizar um relatório em tempo real das categorias classificadas. As categorias disponíveis para classificação da ferramenta são: *Test Addition*, *Test-first*, *Test-last*, *Refactoring*, *Test Driven Development* e *Unknown*. A ferramenta possui código fonte aberto e está disponível no Github, seja para novas contribuições quanto para utilização na academia ou indústria de software.

Realizamos a execução de um estudo experimental em um contexto industrial, provendo a união da academia e indústria de software. Os participantes eram profissionais da indústria de software com cargos de desenvolvedores ou arquitetos de software, que possuem curso superior completo em TI. Foram utilizadas atividades que simulam funcionalidades reais. Os efeitos foram avaliados também após a fase de desenvolvimento, ampliando assim dados que não são frequentemente contemplados em outros estudos. Avaliamos também os participantes quanto ao esforço despendido para realização do estudo, onde pudemos constatar que a atividade desenvolvida é semelhante a atividades desenvolvidas no dia-a-dia. Propomos para a academia e indústria um novo paradigma da utilização critérios de teste em conjunto com o TDD, além da ferramenta *Butterfly*, responsável por mensurar a conformidade do processo do TDD, apoiando profissionais e pesquisadores no acompanhamento da adoção do TDD.

Referências

BASILI, Victor R. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*. College Park, MD,EUA, 1992. 24 p. Disponível em: <http://www.cs.umd.edu/~basili/publications/technical/T78.pdf>.

BASILI, Victor R.; CALDIERA, Gianluigi; ROMBACH, H. Dieter. The Goal Question Metric approach. In: *Encyclopedia of Software Engineering*. EUA: Wiley, 1994. p. 10. ISBN 9780471028956.

BECK, Kent. *eXtreme Programming Explained: Embrace Change*. 1. ed. EUA: Addison-Wesley, 1999. 224 p. (XP Series). ISBN 978-0201616415.

BECK, Kent. *Test-Driven Development: By Example*. 1. ed. EUA: Addison-Wesley Professional, 2002. 240 p. ISBN 978-0321146533.

BECK, Kent; BEEDLE, Mike; BENNEKUM, Arie van; COCKBURN, Alistair; CUNNINGHAM, Ward; FOWLER, Martin; GRENNING, James; HIGHSMITH, Jim; HUNT, Andrew; JEFFRIES, Ron; KERN, Jon; MARICK, Brian; MARTIN, Robert C.; MELLOR, Steve; SCHWABER, Ken; SUTHERLAND, Jeff; THOMAS, Dave. *Manifesto for Agile Software Development*. 2001. Página Web. Disponível em: <http://agilemanifesto.org/>.

BECKER, Karin; PEDROSO, Bruno de Souza Costa; PIMENTA, Marcelo Soares; JACOBI, Ricardo Pezzuol. Besouro: A framework for exploring compliance rules in automatic TDD behavior assessment. *Information and Software Technology*, Elsevier, Amsterdam, Países Baixos, v. 57, p. 494–508, jan. 2015. ISSN 0950-5849.

BRIAND, L.; BIANCULLI, D.; NEJATI, S.; PASTORE, F.; SABETZADEH, M. The case for context-driven software engineering research: Generalizability is overrated. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, EUA, v. 34, n. 5, p. 72–75, set. 2017. ISSN 0740-7459.

CAUSEVIC, A.; PUNNEKKAT, S.; SUNDMARK, D. TDD^{HQ}: Achieving higher quality testing in test driven development. In: *39th Euromicro Conference Series on Software Engineering and Advanced Applications*. Washington, DC, EUA: IEEE, 2013. p. 33–36. ISBN 978-0-7695-5091-6.

CAUSEVIC, Adnan; SHUKLA, Rakesh; PUNNEKKAT, Sasikumar; SUNDMARK, Daniel. Effects of negative testing on TDD: An industrial experiment. In: BAUMEISTER, Hubert; WEBER, Barbara (Ed.). *14th International Conference on Agile Software Development*. Berlin, Alemanha: Springer, 2013. (Lecture Notes in Business Information Processing, v. 149), p. 91–105. ISBN 978-3-642-38313-7.

CAUSEVIC, A.; SUNDMARK, D.; PUNNEKKAT, S. Factors limiting industrial adoption of test driven development: A systematic review. In: *4th IEEE International Conference on Software Testing, Verification and Validation*. EUA: IEEE, 2011. p. 337–346. ISBN 978-1-61284-174-8. ISSN 2159-4848.

CRONBACH, Lee J. Coefficient alpha and the internal structure of tests. v. 16, n. 3, p. 297–334, set. 1951.

DELAMARO, Márcio Eduardo; MALDONADO, José Carlos; JINO, Mario. *Introdução ao Teste de Software*. 2. ed. Rio de Janeiro, RJ,Brasil: Elsevier, 2016. 448 p. (Campus). ISBN 978-85-352-8352-5.

DIEBOLD, Philipp; DAHLEM, Marc. Agile practices in practice: A mapping study. In: *18th International Conference on Evaluation and Assessment in Software Engineering*. New York, NY, EUA: ACM, 2014. p. 30:1–30:10. ISBN 978-1-4503-2476-2.

Digital.ai. *14th Annual State of Agile Report*. 2020. Página Web. Disponível em: <https://stateofagile.com/>.

DINGSøYR, Torgeir; NERUR, Sridhar; BALIJEPALLY, VenuGopal; MOE, Nils Brede. A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software*, Elsevier, Países Baixos, v. 85, n. 6, p. 1213 – 1221, jun. 2012. ISSN 0164-1212.

DOLEZEL, Michal; BUCHALCEVOVA, Alena; MENCIK, Michal. The state of agile software development in the Czech republic: Preliminary findings indicate the dominance of "abridged" scrum. In: BASL, Petr Doucekand Josef; TJOA, A Min; RAFFAI, Maria; PAVLICEK, Antonin; DETTER, Katrin (Ed.). *13th IFIP WG 8.9 International Conference on Research and Practical Issues of Enterprise Information Systems*. [S.l.: s.n.], 2019. v. 375, p. 43–54. ISBN 978-3-030-37631-4.

DYBA, T.; DINGSOYR, T. What do we know about agile software development? *IEEE Software*, v. 26, n. 5, p. 6–9, Sep 2009. ISSN 0740-7459.

ELBANNA, A.; SARKER, S. The risks of agile software development: Learning from adopters. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, EUA, v. 33, n. 5, p. 72–79, set.–out. 2016. ISSN 0740-7459.

FUCCI, D.; ERDOGMUS, H.; TURHAN, B.; OIVO, M.; JURISTO, N. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *Transactions on Software Engineering*, IEEE Computer Society, EUA, v. 43, n. 7, p. 597–614, jul. 2017. ISSN 0098-5589.

GAROUSI, Vahid; FELDERER, Michael; KUHRMANN, Marco; HERKILOGLU, Kadir. What industry wants from academia in software testing?: Hearing practitioners' opinions. In: *21st International Conference on Evaluation and Assessment in Software Engineering (EASE 2017)*. New York, NY, EUA: ACM, 2017. p. 65–69. ISBN 978-1-4503-4804-1.

GEORGE, Boby; WILLIAMS, Laurie. A structured experiment of test-driven development. *Information and Software Technology*, Elsevier, Amsterdam, Países Baixos, v. 46, n. 5, p. 337–342, abr. 2004. ISSN 0950-5849.

GHAFARI, Mohammad; GROSS, Timm; FUCCI, Davide; FELDERER, Michael. Why research on test-driven development is inconclusive? In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. New York, NY, USA: Association for Computing Machinery, 2020. (ESEM '20). ISBN 9781450375801. Disponível em: <https://doi.org/10.1145/3382494.3410687>.

HAMMOND, Susan; UMPHRESS, David. Test driven development: The state of the practice. In: SMITH, Randy K.; VRBSKY, Susan V. (Ed.). *50th ACM Annual Southeast Regional Conference*. New York, NY,EUA: ACM, 2012. p. 158–163. ISBN 978-1-4503-1203-5.

HODA, Rashina; SALLEH, Norsaremah; GRUNDY, John. The rise and evolution of agile software development. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, EUA, v. 35, n. 5, p. 58–63, set.—out. 2018. ISSN 0740-7459.

ISO; IEC; IEEE. *ISO/IEC/IEEE 24765:2010 – Systems and software engineering – Vocabulary*. 2010. 418 p. Padrão.

JANZEN, D.; SAIEDIAN, H. Test-driven development concepts, taxonomy, and future direction. *Computer*, IEEE Computer Society, EUA, v. 38, n. 9, p. 43–50, set. 2005. ISSN 0018-9162.

JURISTO, Natalia; MORENO, Ana M. *Basics of Software Engineering Experimentation*. [S.l.]: Kluwer Academic Publishers, 2001. 395 p. ISBN 9781441950116.

KOU, Hongbing; JOHNSON, Philip M.; ERDOGMUS, Hakan. Operational definition and automated inference ofătest-driven development with Zorro. *Annals of Software Engineering*, Kluwer, Red Bank, NJ, EUA, v. 17, n. 1, p. 57–85, mar. 2010. ISSN 1022-7091.

MALDONADO, J. C. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese (Doutorado) — DCA/FEEC/UNICAMP, Campinas, SP, Brazil, jul. 1991.

MATOS, Altieres De; RODER, Larissa Bonifácio; NICOLODI, Luciane Baldo; Ré, Reginaldo; SILVA, Marco Aurélio Graciotto. Estudo preliminar sobre os impactos do desenvolvimento orientado a testes no processo de revisão de código. In: *Escola Regional de Engenharia de Software (2018)*. Dois Vizinhos, PR, Brasil: UTFPR, 2018. p. 10.

MATOS, Altieres de; Ré, Reginaldo; SILVA, Marco Aurélio Graciotto. A tool to measure TDD compliance: a case study with professionals. In: TONIN, Graziela Simone; ESTáCIO, Bernardo; GOLDMAN, Alfredo; GUERRA, Eduardo (Ed.). *9th Brazilian Workshop on Agile Methods (WBMA'2018)*. [S.l.: s.n.], 2018. v. 981, p. 1–15. ISBN 978-3-030-14309-1.

MYERS, Glenford J.; BADGETT, Tom; SANDLER, Corey. *The Art of Software Testing*. 3. ed. Hoboken, NJ,EUA: John Wiley & Sons, 2013. 240 p. Revised and updated by Tom Badgett and Todd M. Thomas, with Corey Sandler. ISBN 978-1-118-03196-4.

NAZIR, Noshiba; HASTEER, Nitasha; BANSAL, Abhay. A survey on agile practices in the Indian IT industry. In: *2016 6th International Conference - Cloud System and Big Data Engineering (Confluence)*. [S.l.]: IEEE, 2016. p. 635–640. ISBN 978-1-4673-8204-5.

OFFUTT, A. Jefferson; LEE, Ammei; ROTHERMEL, Gregg; UNTCH, Roland H.; ZAPF, Christian. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, v. 5, n. 2, p. 99–118, abr. 1996.

Pachulski Camara, Bruno Henrique; Graciotto Silva, Marco Aurélio. A strategy to combine test-driven development and test criteria to improve learning of programming skills. In: *47th ACM Technical Symposium on Computing Science Education*. New York, NY, EUA: ACM, 2016. p. 443–448. ISBN 978-1-4503-3685-7.

RAULAMO-JURVANEN, Päivi; MäNTYLä, Mika; GAROUSI, Vahid. Choosing the right test automation tool: A grey literature review of practitioner sources. In: *21st International Conference on Evaluation and Assessment in Software Engineering (EASE 2017)*. New York, NY, EUA: ACM, 2017. p. 21–30. ISBN 978-1-4503-4804-1.

RODRíGUEZ, Pilar; MARKKULA, Jouni; OIVO, Markku; TURULA, Kimmo. Survey on agile and lean usage in Finnish software industry. In: *6th International Symposium on Empirical Software Engineering and Measurement*. New York, NY,EUA: ACM, 2012. p. 139–148. ISBN 978-1-4503-1056-7.

SANTOS, Adrian; SPISAK, Jaroslav; OIVO, Markku; JURISTO, Natalia. *Improving Development Practices through Experimentation: an Industrial TDD Case*. 2018. ArXiv. Disponível em: <https://arxiv.org/abs/1809.01828>.

SCHWABER, Ken. SCRUM development process. In: SUTHERLAND, Jeff (Ed.). *Business Object Design and Implementation Workshop – Addendum to the Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1995)*. New York, NY, EUA: ACM, 1995. p. 1–23. ISBN 0897917219. Disponível em: <http://www.jeffsutherland.org/oopsla/schwapub.pdf>.

SCHWABER, Ken; SUTHERLAND, Jeff. *The Scrum Guide – The Definitive Guide to Scrum: The Rules of the Game*. 2017. 19 p. Página Web. Disponível em: <https://www.scrumguides.org/>.

SHELTON, William; LI, Nan; AMMANN, Paul; OFFUTT, Jeff. Adding criteria-based tests to test driven development. In: *5th International Conference on Software Testing, Verification and Validation*. New York, NY, EUA: IEEE, 2012. p. 878–886. ISBN 978-1-4577-1906-6.

SPINELLIS, D. State-of-the-art software testing. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, EUA, v. 34, n. 5, p. 4–6, set. 2017. ISSN 0740-7459.

TOSUN, Ayse; AHMED, Muzamil; TURHAN, Burak; JURISTO, Natalia. On the effectiveness of unit tests in test-driven development. In: *Proceedings of the 2018 International Conference on Software and System Process*. New York, NY, EUA: ACM, 2018. p. 113–122. ISBN 978-1-4503-6459-1.

VALLON, Raoul; ESTáCIO, Bernardo José da Silva; PRIKLADNICKI, Rafael; GRECHENIG, Thomas. Systematic literature review on agile practices in global software development. v. 96, p. 161–180, abr. 2018.

VERSIONONE. *12th annual state of agile^{TM} report*. 2018. Página Web. Disponível em: <https://stateofagile.versionone.com/>.

VINCENZI, Auri; DELAMARO, Márcio; HöHN, Erika; MALDONADO, José Carlos. Functional, control and data flow, and mutation testing: Theory and practice. In: BORBA, Paulo; CAVALCANTI, Ana; SAMPAIO, Augusto; WOODCOOK, Jim (Ed.). *Testing Techniques in Software Engineering*. Berlin,Alemanha: Springer, 2010, (Lecture Notes in Computer Science, v. 6153). cap. 2, p. 18–58. ISBN 978-3-642-14334-2.

WOHLIN, Claes; RUNESON, Per; HöST, Martin; OHLSSON, Magnus C.; REGNELL, Björn; WESSLéN, Anders. *Experimentation in Software Engineering: An Introduction*. 1.

ed. Sweden: Kluwer Academic Publishers, 2000. 204 p. (The Kluwer International Series in Software Engineering).

Apêndices

Apêndice A

Questionário NASA-TLX

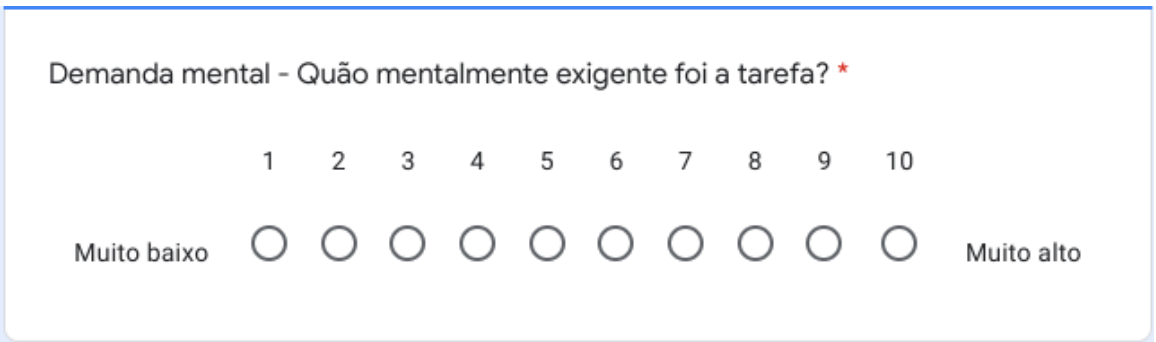

Figura A.1. Questionário NASA-TLX: Questão sobre demanda mental.
Fonte: Adaptado de https://humansystems.arc.nasa.gov/groups/TLX/downloads/TLXScale.pdf.

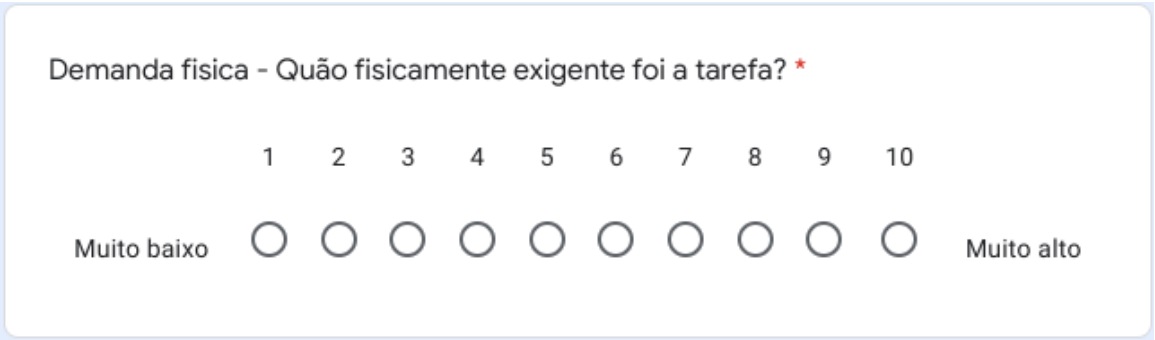

Figura A.2. Questionário NASA-TLX: Questão sobre demanda física.
Fonte: Adaptado de https://humansystems.arc.nasa.gov/groups/TLX/downloads/TLXScale.pdf.

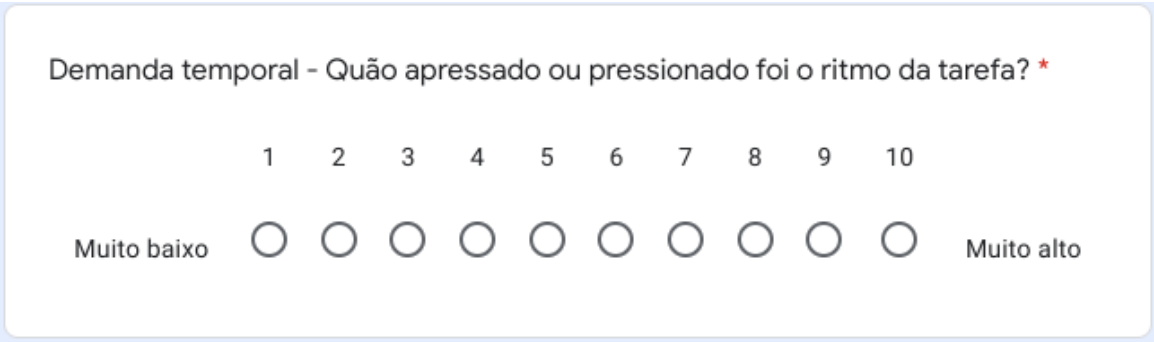

Figura A.3. Questionário NASA-TLX: Questão sobre demanda temporal.
Fonte: Adaptado de https://humansystems.arc.nasa.gov/groups/TLX/downloads/TLXScale.pdf.

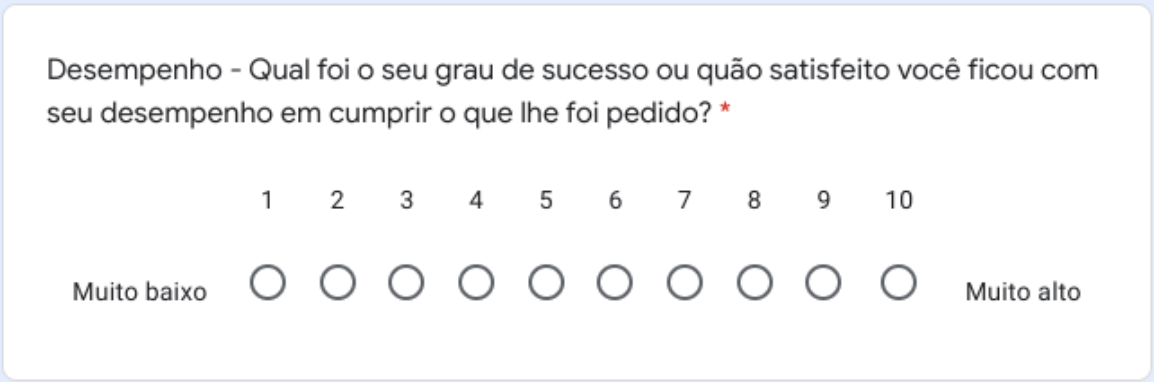

Figura A.4. Questionário NASA-TLX: Questão sobre desempenho.
Fonte: Adaptado de https://humansystems.arc.nasa.gov/groups/TLX/downloads/TLXScale.pdf.

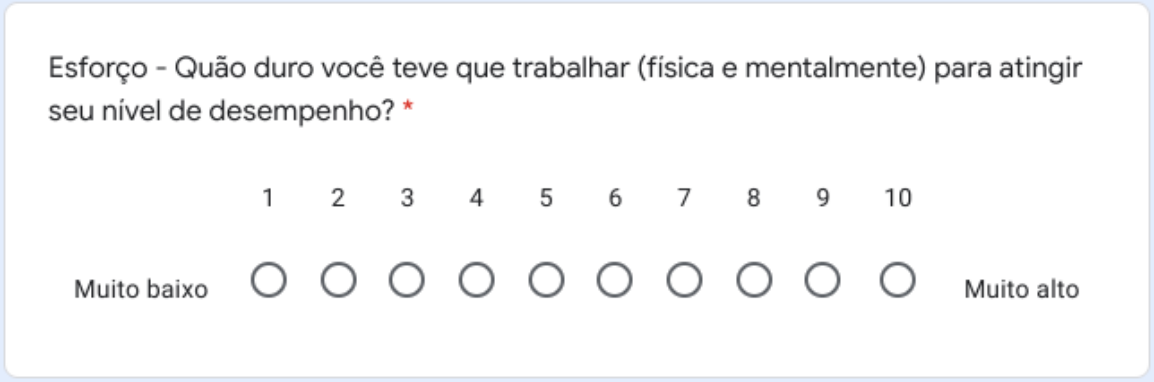

Figura A.5. Questionário NASA-TLX: Questão sobre esforço.
Fonte: Adaptado de https://humansystems.arc.nasa.gov/groups/TLX/downloads/TLXScale.pdf.

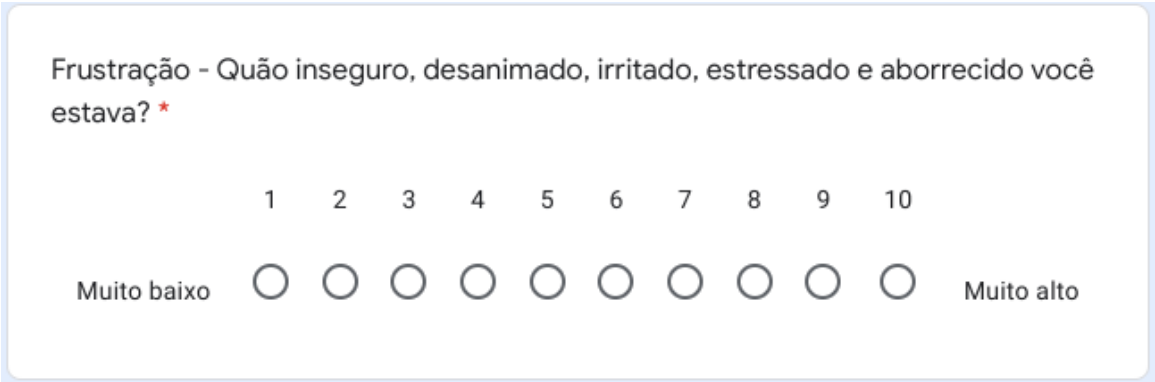

Figura A.6. Questionário NASA-TLX: Questão sobre frustração.

Fonte: Adaptado de https://humansystems.arc.nasa.gov/groups/TLX/downloads/TLXScale.pdf.