UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RERISSON DANIEL COSTA SILVA MATOS

ACIDENTES NA PRÁTICA DE TDD: PERCEPÇÕES E CONSEQUÊNCIAS

CAMPINA GRANDE - PB

2022

RERISSON DANIEL COSTA SILVA MATOS

ACIDENTES NA PRÁTICA DE TDD: PERCEPÇÕES E CONSEQUÊNCIAS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador : Tiago Lima Massoni

CAMPINA GRANDE - PB

2022

RERISSON DANIEL COSTA SILVA MATOS

ACIDENTES NA PRÁTICA DE TDD: PERCEPÇÕES E CONSEQUÊNCIAS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Tiago Lima Massoni

Orientador – UASC/CEEI/UFCG

Patricia Duarte de Lima Machado

Examinador – UASC/CEEI/UFCG

Francisco Vilar Brasileiro

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 02 de Setembro de 2022.

CAMPINA GRANDE - PB

RESUMO

Apesar da existência de alguns estudos inconclusivos [8], é observável a proeminência de evidências que apontam como a técnica de Desenvolvimento Orientado a Testes (Test Driven Development, TDD) pode aumentar a qualidade de código e a confiabilidade de um sistema de software [1]. Apesar de ser uma prática relativamente elementar, um estudo [2] realizado em 2010 verificou que muitos programadores não executam todos os passos em conformidade com a proposta da técnica. Nosso objetivo neste trabalho é reproduzir parte desse estudo através de um novo questionário, utilizando alguns dos acidentes levantados em [2] acrescidos de perguntas discursivas que investigam as circunstâncias e consequências de sua existência. Como resultado, percebemos que a frequência de alguns acidentes foi similar às apresentadas em [2]. Além disso, também notamos que os desenvolvedores percebem as consequências de não executar algumas das etapas de TDD e que eles próprios possuem consciência que a prática deve ser aliada a outras técnicas de design para que o tempo investido tenha retorno compatível e a execução da técnica não seja onerosa.

Acidentes na prática de TDD: Percepções e consequências

Rerisson Daniel Matos Costa Silva Matos
Universidade Federal de Campina Grande
R. Aprígio Veloso, 882 - Universitário, Campina Grande - PB, 58428-830
rerisson.matos@ccc.ufcg.edu.br

Tiago Lima Massoni
Universidade Federal de Campina Grande
R. Aprígio Veloso, 882 - Universitário, Campina Grande - PB, 58428-830
massoni@computacao.ufcg.edu.br

RESUMO

Apesar da existência de alguns estudos inconclusivos [8], é observável a proeminência de evidências que apontam como a técnica de Desenvolvimento Orientado a Testes (Test Driven Development, TDD) pode aumentar a qualidade de código e a confiabilidade de um sistema de software [1]. Apesar de ser uma prática relativamente elementar, um estudo [2] realizado em 2010 verificou que muitos programadores não executam todos os passos em conformidade com a proposta da técnica. Nosso objetivo neste trabalho é reproduzir parte desse estudo através de um novo questionário, utilizando alguns dos acidentes levantados em [2] acrescidos de perguntas discursivas que investigam as circunstâncias e consequências de suas ocorrências. Como resultado, percebemos que a frequência de alguns acidentes foi similar às apresentadas em [2]. Além disso, também notamos que os desenvolvedores percebem as consequências de não executar algumas das etapas de TDD e que eles próprios possuem consciência de que a prática deve ser aliada a outras técnicas de design para que o retorno do tempo investido seja compatível com seu custo e a execução da técnica não seja onerosa.

Palavras-chave

Desenvolvimento orientado a testes, Técnicas de design, Avaliação de Técnicas

1. INTRODUÇÃO

TDD surge como prática essencial da metodologia Extreme Programming (XP) [3]. Como difundido por diversas metodologias ágeis, o design emerge conforme o software cresce. Assim, a técnica proporciona feedback rápido aos desenvolvedores por meio da escrita e execução constante de testes pequenos, podendo aumentar a confiança nas refatorações e a simplicidade do design do código. No contexto de design emergente, ela pode também ser entendida como ferramenta que permite ao desenvolvedor vislumbrar o comportamento do software antes mesmo de escrever o código, auxiliando no design e evolução de uma solução de software efetiva.

Destarte, além de um mecanismo rápido de validação do código desenvolvido, TDD oferece uma oportunidade para o time de desenvolvimento esclarecer as funcionalidades que o cliente precisa e entregá-las de forma mais confiável, previsível e mensurável, estabelecendo assim uma relação mutualista entre desenvolvedores, softwares e usuários. Essas características da técnica beneficiam principalmente sistemas com longo ciclo de vida, nos quais os custos de manutenção tendem a ser maiores que os custos de concepção e a presença de uma rede de segurança efetiva na prevenção de acidentes é mais relevante para a garantia da qualidade do serviço executado pelo software.

Dada a diversidade da natureza dos softwares desenvolvidos atualmente, nem todo sistema se beneficiaria na mesma proporção do uso de TDD. Em sistemas com tempo de vida pequeno, por exemplo, a perda de produtividade causada pelo esforço cognitivo adicional empregado no desenvolvimento de testes pode não se justificar devido a complexidade do sistema não crescer muito. Além disso, é necessário que os desenvolvedores adquiram domínio de técnicas de testes de unidade e disciplina para que o uso de TDD seja efetivo, assim como a habilidade de escrever código auto testável. Além disso, aplicar TDD em um projeto que não seja novo também se mostra um desafio, dado que decisões arquiteturais prévias podem dificultar ou mesmo impedir a escrita de testes de unidade automatizados.

Em [2] Aniche realizou um levantamento dos acidentes mais comuns, em que, usando um formulário, consultou a ocorrência de acidentes durante a prática de TDD em uma amostra de 210 desenvolvedores. Essa consulta indicou que alguns dos acidentes ocorrem frequentemente na prática de 25% dos desenvolvedores. Dada a frequência acentuada desses acidentes, decidimos consultar a ocorrência de alguns deles em uma amostra menor. A escolha do tamanho da amostra se deu a fim de possibilitar o acompanhamento das situações em que os acidentes ocorreram, bem como das consequências percebidas pelos desenvolvedores.

Assim foram selecionados alguns acidentes apresentados em [2] e adicionadas duas questões discursivas sobre a situação em que o acidente ocorreu e quais foram suas consequências. Espera-se que os resultados possam dar direcionamento e destacar pontos de atenção que podem ser observados quando se está aprendendo, ensinando ou implementando TDD.

2. FUNDAMENTAÇÃO TEÓRICA

O desenvolvimento aplicando TDD é executado da seguinte forma: escreva um teste que falhe após ser executado; implemente o código que faça esse teste passar e depois refatore essa implementação. A execução técnica pode ser resumida por meio do jargão ``*Red Green Refactor*``, uma alusão às cores das barras mostradas em ambientes de desenvolvimento populares ao executar os testes.

Um exemplo de uso de TDD contemporâneo seria o uso para execução de testes de unidade em componentes React, beneficiando-se da maior velocidade e assertividade dos testes com comparação a testes manuais que seriam realizados no navegador web ou em um emulador. Um fluxo de trabalho aplicando TDD nesse contexto, para implementação de um componente rotulado para entrada do usuário, seria o seguinte: cria-se um arquivo de teste e o executa. Após esse passo, o ambiente de execução reporta um erro por notar que não existem testes implementados no arquivo. Em seguida, o desenvolvedor implementa o primeiro teste e executa a suíte novamente e o teste

falha, dessa vez por não existir código que o implementa. Após esse passo é criado o arquivo que implementa o código de produção e a suíte é executada novamente. Após notar o erro reportado devido a ausência do comportamento, o desenvolvedor implementa parte dele, por exemplo, verificar que o rótulo da entrada aparece. Após verificar que o teste passou, o desenvolvedor volta ao código de produção e avalia se é possível e necessário refatorá-lo e age de acordo com seu julgamento, prosseguindo à repetição do ciclo escrevendo um novo teste e observando sua falha.

Como podemos notar no exemplo anterior, apesar do princípio da técnica ser simples e de execução relativamente mecânica, o ambiente de desenvolvimento de software possui diversas nuances que podem comprometer a execução efetiva da técnica, principalmente relativas à granularidade dos testes e como testar em situações atípicas.

2.1 TDD e Qualidade de software

Em [1], são apresentados estudos que nos permitem afirmar com relativa segurança que TDD pode aumentar a qualidade interna e externa do software, reduzindo a quantidade de defeitos que são percebidos no ambiente de produção e aumentando a produtividade a médio-longo prazo. Naturalmente isso aumenta a auto-estima da equipe por reduzir a necessidade de hotfixes e diminuir a insegurança durante refatorações necessárias à extensão e modificação do código [7].

Além disso, testar um código que não existe é uma mudança de paradigma e vai contra a abordagens tradicionais de desenvolvimento de software. Isso gera ceticismo em iniciantes e não iniciados na prática. Em um estudo qualitativo [5] realizado em 2011, todos os participantes afirmaram inicialmente não acreditar nos benefícios do uso de TDD, mas que com o tempo e experiência eles tendem a se tornar mais evidentes. Também é relatado em alguns estudos que há uma perda de produtividade nas fases iniciais do desenvolvimento, mas que geralmente é compensada no longo prazo [6].

Em [2] é apresentada uma análise que relaciona alguns acidentes na prática de TDD com o tempo de prática e experiência dos respondentes.

2.2 Acidentes na prática de TDD

Conforme levantado por [2], podemos destacar alguns dos acidentes mais recorrentes: esquecer de refatorar o código, implementar testes muito complexos e refatorar outra parte do código enquanto desenvolvemos um teste. Diferentemente de [2], onde os acidentes foram chamados de erros, neste trabalho foram chamados de acidentes devido a muitos deles ocorrerem devido a deficiências na consolidação de bons hábitos de desenvolvimento que muitas vezes não se encontram no espaço de consciência de times com pouco tempo de prática.

Apesar da técnica ser concisa e direta, os desenvolvedores iniciantes no uso de TDD podem enfrentar algumas dificuldades [4]. Como apontado por [2], alguns acidentes são frequentemente cometidos por cerca de 25% dos programadores. Essas dificuldades podem frustrar — a ponto de comprometer seu rendimento ou capacidade de resolução de problemas — o desenvolvedor iniciante na prática de TDD, principalmente pela diminuição da velocidade de entrega de código [5]. Além disso, é importante que bons princípios de OO e design sejam aplicados em conjunto para que TDD seja eficiente.

Por isso se faz necessário o desenvolvimento de instrumentos que possibilitem um entendimento de como a prática é percebida por aqueles que a executam, assim como de que situações favorecem a ocorrência dos acidentes e o custo das adaptações necessárias frente às consequências.

Dada a diversidade de domínios onde TDD pode ser empregado, há aspectos específicos que podem ser ignorados ao analisar apenas métricas quantitativas. Em [2], por exemplo, é analisada a frequência e a correlação entre tempo de experiência e a ocorrência de acidentes na prática de TDD. Apesar de serem bons indicativos de como TDD ocorre *in vivo*, ao restringirmos a percepção às características qualitativas, omitimos a análise da importância e o significado da prática no contexto em que ocorreu.

Gerosa et al. usaram formulários tanto em [2] quanto em [5] como instrumento para coleta de dados. O formulário abstrai especificidades da implementação, como *framework* usado para implementação dos testes, linguagem de programação e domínio da aplicação, permitindo uma compreensão da execução da técnica que é menos dependente da implementação, porém mais enviesada pela experiência individual da desenvolvedora e mais abundante em contexto. A amostra analisada em [2] foi obtida através da divulgação em listas online de discussão, enquanto [5] foi aplicado a alguns participantes de uma conferência de métodos ágeis.

3. METODOLOGIA

Pelo fato dos acidentes na prática de TDD serem fenômenos contemporâneos influenciados por diversas variáveis do ambiente de implementação e também fatores humanos subjetivos, o estudo qualitativo é eficiente na caracterização do ambiente e na avaliação das trocas proporcionadas pelo uso da técnica [5]. Além disso, normalmente TDD é associada a outras técnicas ágeis, como princípios de design, *shift left quality* e evolução iterativa. Concomitantemente, a análise quantitativa é relevante para uma caracterização mais comparável de variáveis como frequência de ocorrência de eventos, tempo de experiência e prática com TDD. Assim, neste trabalho optou-se pelo uso de um formulário com perguntas mistas para caracterização do tempo de experiência, frequência de ocorrência dos acidentes e perguntas abertas a fim de observarmos a situação e consequências de falhas na aplicação da técnica pelos respondentes.

O trabalho de pesquisa que nos inspirou [2] apresenta um levantamento sobre os possíveis acidentes na execução de TDD. Considerando o propósito de avaliar mais profundamente o contexto de ocorrência e viabilizar a análise das respostas discursivas, foram selecionados 4 dos 9 acidentes apresentados no formulário. Em [2] as perguntas sobre a ocorrência dos acidentes eram apresentadas em inglês, por isso foi necessário traduzi-las e adaptar sua redação.

O critério de escolha foi definido observando o impacto na confiança dos desenvolvedores, seja por dificultar a aplicação de TDD ou a consolidação da técnica como hábito de boa higiene de desenvolvimento. Na Tabela 1 são apresentados os acidentes selecionados, a tradução das perguntas e impactos considerados na seleção.

Tabela 1. Questões selecionadas, nova redação e impactos

Pergunta original	Tradução	Impactos
*How often do you forget to watch the test fail? *Sometimes we write a failing test, and then write the code that make the test pass before checking that we really got the red bar.*	*Quão frequentemente você lembra de executar o teste e observar sua falha antes de implementar o código? *Às vezes implementamos um teste e esquecemos de executá-lo antes de implementar o*	*Ocorrência de resultados falsos positivos na execução dos testes; baixo fundamento para confiança nos testes; Testes ineficazes.*

	código que fará ele passar	
*How often do you forget to apply a refactoring after the green bar? *Sometimes we start writing a new test immediately after we get the green bar, skipping the refactoring step.*	*Quão frequentemente você aplica uma refatoração depois que o teste passa? *Às vezes começamos a escrever um novo teste imediatamente após o anterior passar, pulando assim a etapa de refactoring.*	*Menor verificação da implementação; Maior presença de smells, código morto; arquitetura mais rígida.*
*How often do you need to write a complex test scenario? *Sometimes we need to write a big setUp and mock lots of objects in order to have a valid scenario to do our test.*	*Quão frequentemente você tem que escrever cenários de teste complexos? * Às vezes é necessário escrever um método setUp muito grande ou usar muitos test doubles (mocks, spies, etc.) para ter um cenário de teste válido.*	*Acoplamento dos testes a test doubles; exaustão cognitiva para concepção e manutenção; forte dependência entre o código de verificação e a implementação.*
*How often do you apply a refactoring on a piece of code while working on another test? *Sometimes we are working on a test and we pass through a piece of code that needs to be refactored and then we we refactor it at that moment, before finishing the test*	*Quão frequentemente você aplica um refactor em uma parte do código enquanto trabalha em outro teste? *Às vezes enquanto trabalhamos em um teste passamos por código que precisa ser refatorado e o refatoramos naquele momento, antes de terminar o teste no qual estávamos trabalhando.*	*Maior sobrecarga cognitiva; menor integridade conceitual na execução e entrega;*

Como exemplo do primeiro acidente da tabela, podemos citar uma situação na qual o desenvolvedor ainda não tem nenhum código e tem certeza que o teste falharia por não encontrar o código da implementação. No segundo acidente, vale mencionar como exemplo uma situação onde um desenvolvedor faz *upload* do código logo após consertar um bug e ver que o código escrito para verificá-lo passar, sem avaliar se o código ou o teste poderiam ser refatorados. Sobre testes complexos, observamos sua ocorrência quando um código tem dependências externas com comportamento, como bibliotecas auxiliares de aplicações que dependem de recursos de rede ou de disco. E sobre o último acidente selecionado, uma situação de ocorrência seria quando a desenvolvedora abre um arquivo que pode ser útil para investigação da causa da falha do teste atual, mas acaba notando algum código que poderia ser refatorado mas que não vai fazer o teste atual passar e efetua a refatoração, desviando sua atenção de seu objetivo primário que era fazer o teste passar.

Além da mudança na redação, os itens para seleção de frequência de ocorrência de cada acidente também foram modificados para maior facilidade na leitura e resposta. Enquanto em [2] é usada uma escala numerada, neste trabalho foi usada uma escala verbal, variando de "Nunca" à "Sempre", apresentada na Figura 1.

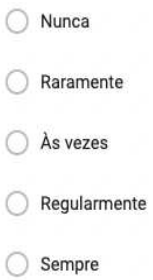

Figura 1. Escala verbal usada para aferição da frequência de ocorrência dos acidentes

Além das perguntas sobre frequência, foram adicionados dois itens de resposta aberta nos quais os respondentes puderam discorrer sobre uma situação onde o acidente ocorreu e quais as consequências percebidas.

Após a elaboração do formulário foi realizada uma etapa de validação com três desenvolvedores, com o objetivo de garantir que as perguntas estavam sendo compreendidas pelos respondentes. Um dos desenvolvedores sugeriu que fossem adicionados exemplos para maior clareza do que era necessário para responder às perguntas abertas. Assim foram inseridos exemplos das situações e de possíveis consequências da ocorrência de cada acidente na descrição das questões.

Em seguida o formulário foi consolidado e enviado individualmente para ex-alunos do curso de ciência da computação da UFCG (Universidade Federal de Campina Grande) que atualmente trabalham em diversas empresas que desenvolvem software para setores variados.

Trata-se de uma amostra não-probabilística, recrutada através de conveniência. A seleção dessas pessoas se deu levando em consideração a diversidade da experiência e a possibilidade de acompanhamento mais próximo para melhor compreensão das respostas pelo pesquisador. Além do trabalho em empresas diversas, também foi considerada a formação anterior ao curso de graduação. Das 10 pessoas convidadas, 4 possuíam além da graduação em ciência da computação, formação técnica de nível médio em tecnologia, tendo assim acesso precoce a projetos de pesquisa e desenvolvimento nos laboratórios da UFCG.

Para análise das respostas objetivas, observamos a frequência de ocorrência de cada acidente na amostra. As situações e consequências de cada acidente será realizada a partir da leitura e síntese das características da situação e de suas consequências na execução da técnica pelos respondentes. Os resultados da síntese foram revisados e discutidos com outro pesquisador.

4. RESULTADOS

Das dez pessoas que receberam o convite para responder o questionário, oito responderam e duas justificaram considerar que não tinham experiência suficiente com TDD para responder. Na Figura 2 pode-se notar que a amostra se constituiu de desenvolvedores com menos de 10 anos de experiência, tendo 50% dos respondentes trabalhado entre 2 e 4 anos desenvolvendo software.

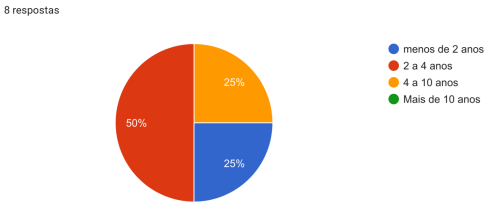

Figura 2. Tempo de trabalho com desenvolvimento de software

No aspecto de tempo de experiência com TDD, a maior parte respondeu que tinha praticado por menos de 2 anos, com apenas um respondente indicando que praticava há mais tempo, conforme apresentado na Figura 3.

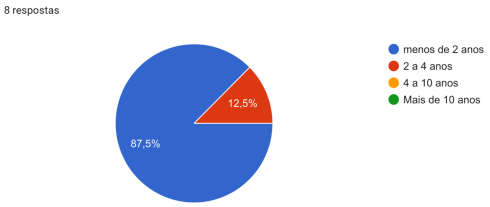

Figura 3. Tempo experiência com TDD

Quando perguntados sobre o contexto de execução, a maior parte indicou a prática na indústria e em projetos acadêmicos, como exposto na Figura 4. Nessa pergunta foi possível assinalar mais de uma alternativa.

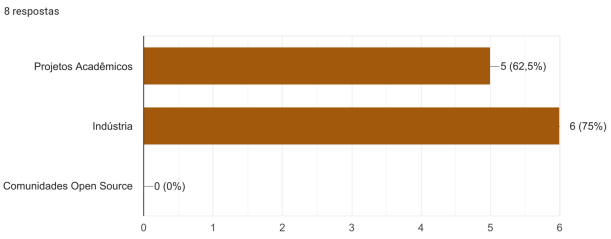

Figura 4. Contexto da prática de TDD

4.1 Quão frequentemente você lembra de executar o teste e observar sua falha antes de implementar o código?

O segundo passo da aplicação da TDD é observar o teste falhar após escrevê-lo. Essa etapa é importante pois indica que o que fará o teste passar é o código que será implementado em seguida e não um acaso na escrita do teste. Caso não seja executada, é possível que o teste passe sem verificar realmente o código, resultando assim em falsos positivos. Em [2] é apontado que 24% encontram esse acidente regularmente ou frequentemente, pois muitas vezes parece óbvio que o teste vai falhar, dado que o código de produção que ele testa ainda não foi implementado.

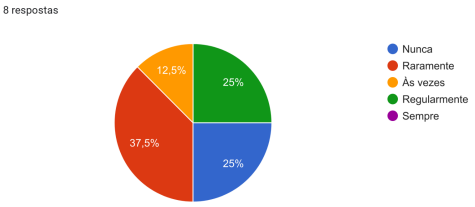

Figura 5. Quão frequentemente os desenvolvedores lembram de observar a falha de um novo teste

Esse acidente se mostra bastante frequente na experiência de implementação dos respondentes, com apenas 25% executando essa etapa regularmente e ninguém a executando sempre, conforme a Figura 5.

Dentre as situações de ocorrência do acidente elencadas apresentaram-se principalmente questões relativas ao ambiente de desenvolvimento e processo de software. Duas respostas indicaram que o acidente ocorreu quando não havia necessidade de apontamento das falhas do teste para que o trabalho fosse considerado como realizado.

Outras três apontaram que o grau de complexidade de execução é relevante para ocorrência do acidente, indicando a ocorrência do acidente quando o ajuste ou função era simples e quando a falha do teste era supostamente conhecida pelo desenvolvedor.

Em relação às consequências observadas, foi apontado *upload* de código defeituoso verificado como correto pelo teste falso positivo no sistema de versões, atrasando o ciclo de feedback e desenvolvimento e comprometendo a confiança na suíte de testes, sugerindo assim que caso a falha do teste tivesse sido verificada localmente o custo seria menor.

Outra resposta mencionou como consequências *"Não perceber erro na implementação do teste, pouco entendimento do fluxo funcional, implementar requisito da forma errada, retrabalho para melhorar teste e implementação do requisito"*, indicando que esquecer de observar a falha do teste pode aumentar o custo de manutenções futuras.

Podemos notar que as consequências apontadas pelos respondentes são compatíveis com os impactos considerados na Tabela 1. Além disso, a frequência de ocorrência na amostra indica que é relevante a adoção de estratégias que previnam a ocorrência e mitiguem as consequências desse acidente.

4.2 Quão frequentemente você aplica uma refatoração depois que o teste passa?

A terceira etapa da técnica consiste em refatorar o código e o teste. Por ser aplicada implementando o teste e o código mais simples possíveis, não é raro que ambos possuam baixa qualidade nas iterações iniciais do desenvolvimento. A fim de melhorar a legibilidade e identificar oportunidades de reuso e simplificação do código e do teste, é de suma importância que ambos sejam analisados e refatorados caso necessário. [2] Sugere que este erro pode ser comum devido a um aspecto psicológico da programação. Assim, ao perceber que um teste passou, a pessoa sente satisfação pela aprovação e passa imediatamente a escrever o teste seguinte, esquecendo de analisar o que foi escrito e refatorar caso necessário.

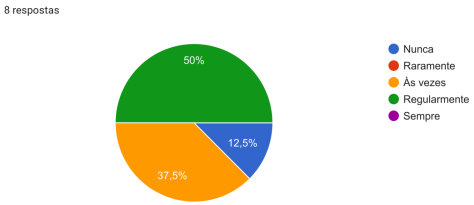

Figura 6: Quão frequentemente desenvolvedores aplicam uma refatoração após o teste passar

No caso desse acidente, metade da amostra apontou executá-la regularmente e pouco mais de 12% indicou nunca executá-la, como expresso na Figura 6.

Em relação às situações nas quais essa etapa mostrou-se inexistente foram apontadas implementações de casos de uso que envolviam manutenção da implementação de validações em três respostas. Um desses casos aponta que o acidente ocorreu *"após implementar um update em uma entidade da aplicação [...]"*, alterando atributos de um modelo. A citação dessas situações sugere que o código que implementa esse tipo de validação pode ser mais coeso e ter uma forma comum estabelecida, deixando pouca margem para refatorações após correções ou incrementos da implementação.

Outras duas indicaram um projeto no qual a revisão considerava apenas o aspecto funcional do software, devido a outras tarefas pendentes e que a refatoração era postergada *ad infinitum*.

As consequências citaram aspectos relacionados à baixa qualidade de código e dificuldade de manutenção, apontando maior custo de refatoração do código após ser promovido a produção. Outras consequências mencionadas foram a inviabilização de reuso e baixo desempenho do código. No geral as consequências foram compatíveis com os impactos apontados no critério de seleção da Tabela 1.

4.3 Quão frequentemente você tem que escrever cenários de teste complexos?

Dada a complexidade de alguns requisitos de software, muitas vezes é difícil decompor os requisitos em casos de teste simples. Ao aplicar TDD, essa dificuldade acaba influenciando a complexidade dos casos de teste e comprometendo a velocidade do ciclo de iteração de desenvolvimento. Uma aplicação efetiva de TDD busca reduzir o tempo desses ciclos. Assim, caso a desenvolvedora se pegue tendo que escrever cenários de teste complexos com frequência é um indicativo de que TDD pode não ser adequado ao domínio ou pode ser necessário mais treinamento e refinamento das habilidades de teste e de arquitetura do time.

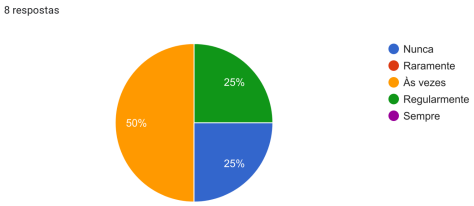

Figura 7. Quão frequentemente desenvolvedores têm que escrever casos de testes complexos

75% das desenvolvedoras indicaram que esse acidente ocorria regularmente ou às vezes, como mostrado na Figura 7. As situações de ocorrência desse acidente apontaram principalmente casos onde o código tinha dependências de bibliotecas externas, sendo esse aspecto citado em 4 das 6 situações elencadas. As 2 restantes citaram excesso de mocks e dificuldade de teste devido a complexidade inata da solução que estava sendo desenvolvida.

Dentre as consequências respondidas, duas citaram dificuldade de entendimento e da escrita de testes sem consumir APIs externas, ressaltando também dificuldades para criar os mocks necessários. Uma delas menciona que o acidente ocorreu "*Ao utilizar bibliotecas de terceiros para formulários e validações, sempre acabo recaindo em muitos mocks e spies para verificar validações corretas e submissão de dados"*, ressaltando a importância do conhecimento de estratégias de testes. Essas situações poderiam se beneficiar da geração automática de *test doubles*, por exemplo.

Outras duas indicaram que a escrita de testes complexos leva a baixa confiança nos incrementos seguintes do código, muito esforço para aumento de cobertura e o sentimento de que reescrever código e testes seria mais simples. Sobre essa consequência, uma das respostas aponta que a escrita de testes complexos acaba consumindo *"muito tempo debugando/estudando fluxo principal para cobrir novo incremento feito, terrível para manter código e testes - por vezes reimplementar parecia mais adequado que incrementar"*.

Também foi mencionado que a dificuldade na escrita de testes complexos faz com que os cenários de testes sejam menos fiéis às circunstâncias dos cenários de produção, sugerindo menor eficácia desses testes. Sobre esse aspecto, uma das consequências cita que *"geralmente são testes mais difíceis de escrever e que às vezes não replicam o cenário exatamente como devia, mas nos contentamos com a barrinha verde sem averiguar se o teste é realmente útil"*. Outra aponta *"problemas na criação de novos testes mais condizentes com a realidade"*.

4.4 Quão frequentemente você aplica um refactoring em uma parte do código enquanto trabalha em outro teste?

Outro ponto comum de equívoco durante a prática de TDD levantado em [2] é que quando estamos trabalhando para fazer um teste passar é comum que tenhamos que ler código legado que evoca o impulso de refatorá-lo imediatamente, mesmo que não seja relacionado ao teste que está falhando.

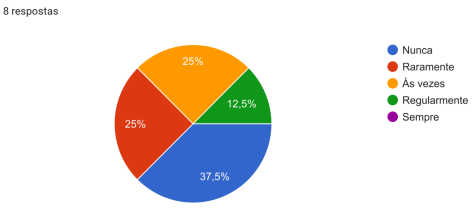

Figura 8. Quão frequentemente desenvolvedores refatoram código não relacionado ao teste no qual trabalham

37,5% dos desenvolvedores apontaram nunca refatorar código não relacionado ao teste que estavam implementando, como podemos observar na figura 8. Outros 37,5% responderam que esse acidente ocorreu às vezes ou regularmente e os demais responderam que ele ocorria raramente em sua prática.

As situações levantadas nas respostas indicaram ocorrência desse acidente em implementações nas quais o código a ser refatorado era mais simples, diminuindo assim o risco de consequências severas à sua ocorrência. Algumas respostas, por exemplo,

citaram situações de correções de tipagem e simplificação de condicionais.

Apesar disso, uma das respostas apontou um caso de necessidade de mudança arquitetural em que pair programming com o desenvolvedor do código legado se mostrou útil para melhor compreensão do propósito do legado.

As consequências apontadas incluíram quebrar testes e só perceber ao rodar toda a suíte, atrasando assim o ciclo de feedback; deixar de lado o que estava fazendo antes; perder o foco impactando o escopo do trabalho atual. Uma das respostas mencionou como consequência positiva deixar o código melhor do que encontrou e mais fácil de manter no futuro.

5. DISCUSSÃO

O resultado da frequência dos acidentes na amostra é compatível com os resultados que [2] apresenta, apesar da diferença no tamanho das amostras. Analisando de forma macro as correlações apontadas em [2] entre a frequência de ocorrência dos acidentes faz sentido quando observamos que quase 90% dos respondentes praticaram TDD por menos de 2 anos.

Apesar da amostra selecionada em [2] ser quase vinte vezes maior, comparar a frequência de ocorrência dos acidentes pode ajudar a compreender melhor a experiência da amostra do trabalho atual.

Em relação a esquecer de observar a falha do teste, [2] mostra que 24% dos desenvolvedores afirmaram ter visto esse acidente regularmente ou frequentemente. Já neste trabalho foi observado que 50% indicaram encontrar esse acidente raramente ou às vezes, o que é compatível com [2] quando consideramos o recorte dos desenvolvedores com até quatro anos de experiência. De forma geral as consequências e situações elencadas sobre esse acidente concordam com os impactos considerados em sua escolha, indicando que é relevante a educação sobre essa etapa para consolidação e uso eficiente da técnica.

Sobre aplicar uma refatoração depois que o teste passa, [2] indica que no geral os desenvolvedores esquecem quase regularmente de aplicar um refactor após observar a falha do teste. Os resultados apresentados neste trabalho são contraditórios quando consideramos o apontado por [2], pois 50% indicaram regularmente aplicar uma refatoração após ver o resultado positivo do teste. As causas para essa contradição podem estar relacionadas à particularidade da experiência e tamanho da amostra, já que como foi apontado as situações onde esse acidente ocorria envolviam códigos mais padronizados que expunham pouca margem ou vantagens em aplicar um refactor.

A escrita de casos de teste complexos é apontada em [2] como sendo encontrada regularmente pelos desenvolvedores Esse resultado é compatível com o apresentado pela amostra consultada, com 75% respondendo ter que escrever testes complexos sempre ou às vezes. A dificuldade relativa à *test doubles* e técnicas complementares de design exposta nas situações e ocorrências desse acidente é compatível com o apontado em [5], que mostra que são necessários conhecimentos complementares de Orientação a Objetos e design para que TDD seja efetivo.

A respeito de aplicar uma refatoração de código não relacionado ao teste em que se estava trabalhando no momento, [2] apurou que quase 40% dos desenvolvedores encontram esse acidente regularmente. Esse resultado é compatível com os 37,5% que o encontram às vezes ou regularmente, apurados da amostra deste trabalho. Apesar disso, as situações apresentadas são compatíveis com os critérios selecionados previamente para seleção, mostrando assim que apesar de ser um acidente regular na prática os desenvolvedores enxergam suas consequências e podem tomar ações para mitigá-las.

A percepção dos desenvolvedores das consequências foi compatível com a de praticantes iniciantes que responderam [5], concordando que a prática de TDD não pode ser considerada uma técnica que isoladamente resolve todos os problemas de software, pois estabelece uma relação dialética com outras variáveis do processo de desenvolvimento. Além disso, também é sugerido pelas consequências apontadas que a escrita de testes e a execução pragmática da técnica podem ser onerosas, especialmente em ambientes de produção com prazos apertados e alta demanda por funcionalidades.

As situações de ocorrência e frequência dos acidentes podem consideradas indicativos da adequabilidade e efetividade da prática de TDD no contexto específico de um projeto, orientando a adoção de práticas complementares e educação da equipe ou mesmo como sintomas de que TDD não é adequado ao projeto.

6. CONCLUSÃO

Este trabalho reproduz a consulta de alguns dos acidentes levantados por [2], aprofundando sua compreensão com exemplos concretos da ocorrência desses acidentes na prática de ex-alunos do curso de ciência da computação da UFCG. A adição das perguntas para consulta das situações de ocorrência e das consequências dos acidentes possibilitou uma investigação da percepção da amostra acerca desses eventos na prática de TDD.

Mesmo dez anos depois da enquete realizada por [2], alguns acidentes tiveram ocorrência similar. Apesar da técnica ser relativamente simples, pudemos notar que ainda existe espaço para a educação de desenvolvedores em situações mais elaboradas da aplicação de TDD. A análise discursiva nos permitiu vislumbrar situações concretas onde técnicas complementares como orientação a objetos e princípios de design poderiam evitar ou mitigar causas e consequências dos acidentes.

Além disso, notamos que o questionário mostrou-se eficiente para a maior compreensão dos resultados apresentados por [2] e [5], contextualizados às particularidades de seleção restrita à conveniência e acompanhamento mais próximo da experiência da amostra.

Como trabalhos futuros, os demais acidentes levantados por [2] e o instrumento podem ser usados para a coleta de dados em contextos mais específicos de desenvolvimento de software, como times que tenham decidido adotar TDD e desejem avaliar a qualidade e efetividade da prática no contexto de desenvolvimento de uma equipe. Variações do questionário também podem ser úteis em investigações futuras da carga cognitiva imposta pelo uso de TDD e da relação entre o custo de usar TDD e do valor promovido pela técnica.

É importante também ressaltar que a validade dos resultados apresentados é restrita à experiência da amostra e ao momento no qual o questionário foi respondido.

7. REFERÊNCIAS

[1] Bissi, W.; Neto A.; Emer M.; “The effects of test driven development on internal quality, external quality and productivity: A systematic review”. Information and Software Technology, Volume 74, 2016, p. 45-54

[2] Aniche, F.; Gerosa M., "Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers". Third International Conference on Software Testing, Verification, and Validation Workshops, Paris, 2010, p. 469-478

[3] Beck, K. “Extreme Programming Explained, Second Edition: Embrace Change”. Boston, Massachusetts, USA, Addison-Wesley, 2004.

[4] Choma J.; Guerra E. M.; Da Silva, T. “Developers’ Initial Perceptions on TDD Practice: A Thematic Analysis with Distinct Domains and Languages”. In: Garbajosa J., Wang X., Aguiar A. (eds) Agile Processes in Software Engineering and Extreme Programming. XP 2018. Lecture Notes in Business Information Processing, vol 314. Springer, Cham.

[5] Aniche M. F.; Gerosa M. A, Ferreira, T. M. "What concerns beginner test-driven development practitioners: a qualitative analysis of opinions in an agile conference". 2nd Brazilian Workshop on Agile Methods. 2011.

[6] Buchan, J., Li, L., & MacDonell, S.G. (2011) Causal Factors, Benefits and Challenges of Test-Driven Development: Practitioner Perceptions, in Proceedings of the 18th Asia-Pacific Software Engineering Conference (APSEC 2011). Hochiminh City, Vietnam, IEEE Computer Society Press, pp.405-413. doi: 10.1109/APSEC.2011.44

[7] Jeffries, Ron & Melnik, Grigori. (2007). Guest Editors' Introduction: TDD--The Art of Fearless Programming. Software, IEEE. 24. 24 - 30. 10.1109/MS.2007.75.

[8] Ghafari, Mohammad & Gross, Timm & Fucci, Davide & Felderer, Michael. (2020). Why Research on Test-Driven Development is Inconclusive?. 1-10. 10.1145/3382494.3410687.