Revista Brasileira de Computação Aplicada, Abril, 2021

DOI: 10.5335/rbca.v13i1.11154
Vol. 13, Nº 1, pp. 75–87
Homepage: seer.upf.br/index.php/rbca/index

ISSN 2176-6649

ARTIGO ORIGINAL

Test-Driven Development: uma revisão sistemática

Test-Driven Development: a systematic review

Gustavo Baculi Benato [1] and Plínio Roberto Souza Vilela [1]

[1]Universidade Estadual de Campinas (UNICAMP)

*g198462@dac.unicamp.br; †prvilela@unicamp.br

Recebido: 08/06/2020. **Revisado:** 04/03/2021. **Aceito:** 29/03/2021.

Resumo

Test-Driven Development (TDD) é uma prática de desenvolvimento de software que ganhou notoriedade quando Kent Beck a definiu como uma parte essencial da Extreme Programming (XP). O presente estudo analisou experimentos e conclusões de estudos, previamente publicados, em relação aos efeitos do TDD na produtividade dos desenvolvedores e na qualidade do software produzido, contrastando o TDD com o Test-Last Development (TLD). Para isto, foi conduzida uma revisão bibliográfica sistemática considerando artigos publicados entre 2003 e 2020. Ao final do processo de revisão, aproximadamente 70% dos estudos analisados, consistiram em experimentos com TDD, 11% equivaleram a revisões bibliográgicas sobre TDD e, por fim, em 18% deles, o principal tema era o TDD em sua essência, detalhando-o. A análise realizada mostra que 54% dos estudos apontaram um aumento considerável na qualidade do software, enquanto nenhum artigo apontou queda na qualidade. Em relação à produtividade, 27% dos estudos apontaram queda na produtividade e 27% foram inconclusivos. Via de regra, os estudos não apontaram melhorias significativas na produtividade quando o TDD foi utilizado. De acordo com a análise, o TDD promove maior qualidade, mesmo que alguns estudos apontem o contrário. Em relação à produtividade, o TDD é inconclusivo. Sendo assim, de acordo com os artigos analisados, não há uma posição final referente ao custo-benefício envolvido nesta prática, discutimos algumas possíveis razões para essa conclusão.

Palavras-Chave: Custo-benefício; Produtividade; Qualidade; Revisão bibliográfica sistemática; Test-Driven Development.

Abstract

Test-Driven Development (TDD) is a software development practice that became famous when Kent Beck defined it as an essential part of Extreme Programming (XP). The present study analyzed, previously published, experiments and study conclusions, related to the effects of TDD on the developers' productivity and on the quality of the software produced, contrasting TDD with Test-Last Development (TLD). Then, a systematic bibliographic review was conducted considering articles published between 2003 and 2020. At the end of the review process, approximately 70% of the studies analyzed, consisted of experiments with TDD, 11% were TDD meta-analysis and finally in 18% of them, the main theme was TDD itself. The analysis carried out shows that 54% of the studies pointed to a considerable increase in software quality, while no article pointed to a decrease in quality. Regarding productivity, 27% of studies pointed to a drop in productivity and 27% were inconclusive. However, studies did not show significant improvements in productivity when TDD was used. According to the analysis, TDD promotes higher quality, even though some studies indicate the opposite. Regarding productivity, TDD analysis is inconclusive. Therefore, according to the papers analyzed there is no final position regarding the cost-benefit involved in this practice, we discuss some of the possible reasons for this conclusion.

Keywords: Cost-benefit; Productivity; Quality; Systematic Bibliographic Review; Test-Driven Development.

1 INTRODUÇÃO

Test-Driven Development (TDD) é uma prática de desenvolvimento de software onde testes unitários automatizados são escritos de forma incremental antes mesmo do código de produção ser desenvolvido (Beck, 2003). O TDD ganhou popularidade quando foi definido por Kent Beck como uma parte essencial da *Extreme Programming* (XP) (Beck, 2010), uma metodologia ágil de desenvolvimento de software focada na aplicação de técnicas de programação, comunicação clara e trabalho em equipe.

(Beck, 2003) fornece mais detalhes sobre a prática do TDD, resumindo-a em um processo de cinco etapas:

i. Escrever um novo caso de teste;
ii. Executar todos os casos de teste e ver o novo falhar;
iii. Escrever apenas o código suficiente para fazer o novo teste passar;
iv. Executar novamente os casos de teste e ver todos passarem;
v. Refatorar o código para remover duplicações.

Essa abordagem, também chamada de *Test-First*, traz uma visão completamente oposta à abordagem tradicional (*Test-Last*), onde o código de produção é escrito de acordo com as especificações do projeto e, normalmente, somente depois de grande parte do código de produção ser escrito, os casos de teste começam a ser desenvolvidos (Hammond and Umphress, 2012). A abordagem tradicional será referenciada aqui como *Test-Last Development* - TLD.

Ao buscar entender os resultados que o TDD proporciona, o que encontra-se são análises e resultados divergentes em relação ao seu custo-benefício, sendo que em muitos casos são considerados inconclusivos, mesmo que sejam executados experimentos em ambientes diversos.

Defende-se que o uso do TDD traz melhorias na qualidade do código e na produtividade dos desenvolvedores (Tosun et al., 2018). Porém, os estudos que investigaram a efetividade do TDD não conseguiram produzir resultados conclusivos e claros (Rafique and Mišić, 2013). Sendo assim, todos os resultados possíveis - positivos, negativos e neutros - foram relatados e analisados no presente estudo.

Mesmo com alguns possíveis benefícios, deve-se considerar os custos que envolvem o uso dessa prática durante o processo de desenvolvimento de software. Entre os possíveis custos existentes, pode-se destacar a curva de aprendizagem dos desenvolvedores, dado que o modo mais usual de programar é escrever os testes unitários somente ao final do ciclo de desenvolvimento, abordagem oposta ao TDD.

Portanto, esta revisão tem o objetivo de entender o atual cenário do TDD, considerando ambientes industriais e acadêmicos, com enfoque em seu custo-benefício, ou seja, busca-se analisar a qualidade do software desenvolvido com TDD e a produtividade de equipes que fizeram uso do TDD, para, assim, obter um comparativo entre o custo-benefício do desenvolvimento de software que faz uso do TDD e do TLD.

Neste artigo, apresenta-se uma revisão bibliográfica sistemática de artigos e estudos empíricos que foram publicados de 2003 a 2020. Adotou-se o ano de 2003 pois neste ano Beck publicou seu primeiro livro sobre o TDD (Beck, 2003), portanto este ano é o marco zero da técnica. Esta revisão se limita a agregar descobertas empíricas em duas construções de resultados. A primeira é a qualidade externa do código, analisando a quantidade de defeitos apresentados em cada experimento estudado e cobertura de testes. A segunda, é a relação entre custo-benefício do TDD quando comparado com o *Test-Last Development* (TLD), analisando o tempo gasto durante o desenvolvimento de software e comparando-o à qualidade obtida.

O processo de revisão sistemática utilizado neste estudo identificou inicialmente 1177 artigos, sendo que 27 deles foram selecionados e discutidos em detalhes. Estes 27 artigos foram selecionados pois têm o TDD como seu foco de pesquisa e abordam um comparativo entre TDD e TLD, na maioria dos casos. Os demais artigos foram rejeitados porque não se encaixavam nos critérios de aceite ou satisfaziam algumas das condições de exclusão (Seção 3.4). Ao analisar cada estudo selecionado, este artigo apresenta um foco maior em entender a relação entre custo e benefício do uso do TDD, bem como a qualidade do software desenvolvido e a produtividade dos programadores. Logo, torna-se desprezível a linguagem de programação utilizadas nos experimentos dos estudos selecionados.

A maioria dos estudos selecionados apontam um aumento na qualidade do código e testes unitários quando o TDD foi utilizado. Porém, são inconclusivos, no que diz respeito à produtividade dos programadores durante o processo de desenvolvimento de software.

As seções restantes deste artigo estão organizadas da seguinte forma: A Seção 2 está focada na descrição e contextualização da prática do TDD; A Seção 3 apresenta uma descrição detalhada do método de pesquisa e dos estudos selecionados para esta revisão; A Seção 4 apresenta os resultados das análises dos artigos selecionados; A Seção 5 apresenta a discussão dos resultados; A Seção 6 relata as ameaças à validade desta revisão sistemática; Por fim, a Seção 7 apresenta a conclusão e possibilidades para estudos futuros.

2 Contexto

O *Test-Driven Development* é uma das práticas ágeis mais controversas em termos de impacto na qualidade do software e na produtividade do programador. Após mais de uma década de pesquisa, ainda não há um parecer final sobre sua eficácia. O TDD assegurava maior qualidade e produtividade, juntamente com um design limpo. Além disto, basicamente bastava seguir uma regra: não escreva código sem um teste que falhe (Karac and Turhan, 2018). Kent (Beck, 2003) define o TDD como um conjunto de técnicas que incentivam o desenvolvimento de projetos simples e o desenvolvimento de um conjunto de testes. Além disso, Beck define o TDD como uma prática de desenvolvimento e design, e não apenas de teste de software.

No ciclo do TDD, o teste e o código implementado normalmente estão relacionados a uma pequena unidade de software, seja um método ou uma função. Desta forma, os testes escritos neste ciclo são testes unitários. O TDD também é conhecido pelo ciclo de refatoração verde-vermelho (Beck, 2003), que resume-se nas seguintes eta-

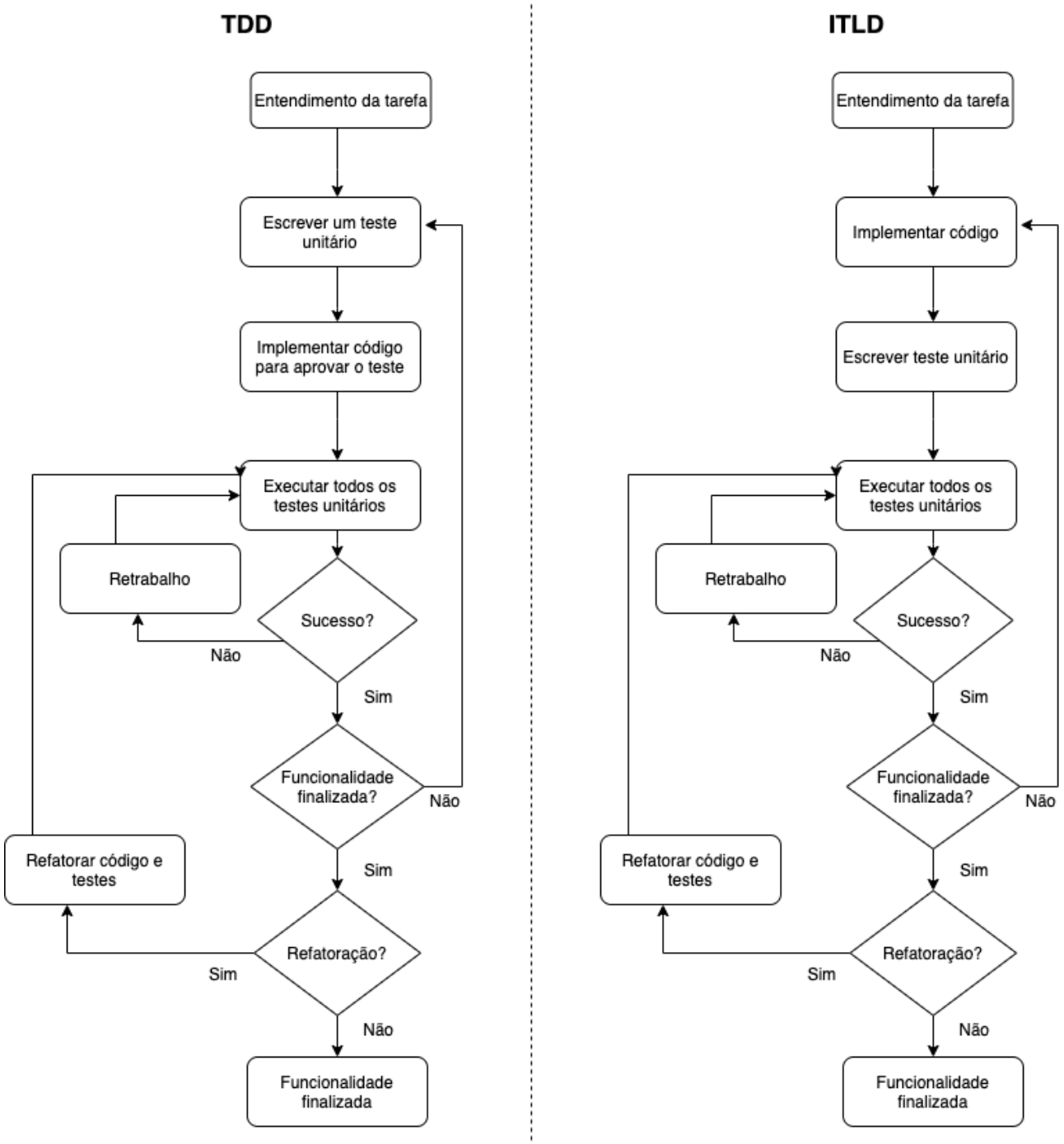

Figura 1: À esquerda, diagrama representando um ciclo *Test-Driven Development* (TDD) e, à direita, um ciclo *Iterative Test-Last Development* (ITLD)

pas:

i. Projete e implemente um teste unitário;
ii. Execute todos os testes e verifique a falha do novo teste adicionado no passo 1 (vermelho);
iii. Desenvolva um novo código que seja suficiente para satisfazer o novo teste;
iv. Execute todos os testes, repita o passo *iii*, se necessário, até que todos os testes tenham sido aprovados (verde);
v. Refatore o código e/ou o teste unitário para otimizar sua estrutura;
vi. Execute todos os testes após a refatoração para assim garantir que todos os testes passaram, ou seja, que todos os testes estão aprovados.

Seguindo a prática do ciclo de refatoração de verde-vermelho, antes de desenvolver uma nova funcionalidade, o programador deve implementar um teste unitário e, apenas após a falha na execução do teste, o código da funcionalidade deve ser desenvolvido. Ao final do ciclo (verde-vermelho), o desenvolvedor deve refatorar o código e os testes antes de iniciar um novo desenvolvimento. Essa refatoração é de suma importância para melhorar as estruturas internas do software e garantir sempre um código limpo e coeso.

A Fig. 1 ilustra as diferenças entre o *Test-Driven Development* (TDD) e o *Iterative Test-Last Development* (ITLD). A notória diferença entre as duas práticas é a inserção da escrita de testes unitários logo no início do ciclo.

Ao fazer uso de uma determinada prática no processo de desenvolvimento de software almeja-se conquistar resultados positivos. No caso do TDD, a intenção é obter mais qualidade e uma maior produtividade da equipe de desenvolvimento. Entretanto, o TDD é uma prática controversa e que divide opiniões, em muitos estudos ela apresenta-se com um resultado de qualidade e produtividade inversamente proporcionais, ou seja, se uma destas variáveis aumenta, a outra diminui. Exemplo em (Bhat and Nagappan, 2006), onde o número de defeitos encontrados foi menor com o TDD (maior qualidade), porém o número de linhas de código escritas foi inferior e o tempo de desenvolvimento com base na quantidade de pessoas por mês foi superior (menor produtividade) [1]. O TDD, também, pode ser observado em experimentos que apresentam resultados neutros e negativos, onde o número de defeitos e complexidade ciclomática[2] se mantém os mesmos (sem alterações na qualidade) e a produtividade diminui (Vu et al., 2009), logo, neste caso, não seria positivo aplicar uma prática assim, com tantas incertezas. A presente revisão tem como objetivo analisar estes diferentes resultados do TDD e entender se é válida a relação de custo-benefício.

3 Metodologia

O método de pesquisa utilizado neste estudo foi uma revisão bibliográfica sistemática (RBS), seguindo as orientações contidas em (Conforto et al., 2011). Uma revisão sistemática é um estudo empírico no qual uma hipótese ou pergunta de pesquisa é abordada para agrupar indicadores de estudos primários por meio de um processo sistemático de pesquisa e extração de dados.

3.1 Processo de pesquisa

O processo de pesquisa da revisão bibliográfica sistemática teve seu início a partir da definição do protocolo de pesquisa, que define o núcleo da revisão. Toda a pesquisa foi estruturada e documentada no software Parsifal (Parsifal, 2019), uma ferramenta online projetada para auxiliar os pesquisadores a realizar revisões sistemáticas da literatura no contexto da engenharia de software.

A primeira tarefa foi elaborar um título para a revisão, juntamente com uma breve descrição. A segunda tarefa consistiu na elaboração do protocolo de pesquisa, onde foi descrito o objetivo, o PICO (Mamédio da Costa Santos et al., 2007) (em detalhes na Seção 3.2), a *string* de busca (em detalhes na Seção 3.3) e os repositórios e bibliotecas digitais que foram utilizados e os critérios de inclusão e exclusão.

Durante a segunda tarefa, também foram elaboradas as perguntas de pesquisa, as quais exercem papel fundamental em uma revisão sistemática, visto que as mesmas guiam todo o processo. Abaixo estão as perguntas de pesquisa do presente estudo:

P1 - Os custos envolvidos na aplicação do *Test-Driven Development* compensam seus benefícios?
P2 - Quando o *Test-Driven Development* for aplicado, a quantidade de defeitos encontrados no software nas fases mais avançadas do desenvolvimento será menor?
P3 - Quando o *Test-Driven Development* for aplicado, muitos problemas serão detectados e removidos ainda nas fases iniciais, muito provavelmente, pelos próprios desenvolvedores?

Logo após a elaboração das perguntas de pesquisa, os estudos primários foram filtrados em repositórios online, através da estrutura de identificação e seleção definida na Seção 3.2. Após a busca nos repositórios digitais, os artigos foram selecionados e classificados conforme apresentam as Seções 3.3 a 3.5.

3.2 Identificação de estudos primários

Em uma revisão sistemática, é essencial elencar as palavras chaves de pesquisa, as quais exercem um papel fundamental no estudo. Deste modo, esta revisão adotou uma linha amplamente utilizada na medicina para identificar a eficácia de um tratamento, conhecida como PICO (*Population, Intervention, Comparison, Outcome*) (Mamédio da Costa Santos et al., 2007). É válido ressaltar que a utilização do PICO faz parte do Parsifal. Esta metodologia consiste na definição de quatro itens que estão descritos na Tabela 1.

Nesta revisão, o PICO ficou definido da seguinte forma:

- **População**: desenvolvedores e/ou engenheiros de software, não havendo restrições referente à linguagem de programação;
- **Intervenção**: o uso da prática *Test-Driven Development*
- **Comparação**: é estabelecida entre o *Test-Driven Development* (TDD) comparado com a prática *Test-Last Development* (TLD);
- **Resultado**: análise do custo-benefício, qualidade e produtividade do software desenvolvido com TDD.

3.3 Processo de seleção dos estudos

Primeiramente, para estruturar a seleção dos estudos, foi criada a *string* de busca com base no PICO da revisão. A *string* de busca consiste em um texto com condicionais e palavras-chaves, que é utilizada para filtrar os estudos. Abaixo, estão as *strings* de busca utilizadas nesta revisão

[1] As métricas citadas foram consideradas pelos autores do artigo analisado
[2] Métrica utilizada para a medição da complexidade do código fonte de um software, que é calculada a partir de um fluxograma do código fonte. (Suleman Sarwar et al., 2013)

Acrônimo	Definição	Descrição
P	*Population* (População)	Grupo de pessoas envolvidas
I	*Intervention* (Intervenção)	O que está sendo analisado
C	*Comparision* (Comparação)	O que está sendo comparado
O	*Outcome* (Resultado)	Resultado esperado

Tabela 1: Descrição da metodologia PICO (Mamédio da Costa Santos et al., 2007) (*What is a PICOC?*, 2019), utilizada, nesta revisão, para listar os termos fundamentais e auxiliar na criação do processo de pesquisa.

sistemática. A primeira foi utilizada no repositório Springer e a segunda nos demais repositórios.

("software development"OR "software engineering"OR
"agile software development") AND ("test-driven development"or "test driven development") AND ("error analysis"OR "software quality"OR "software test"OR "software testing"OR "unit testing"OR "unit tests"OR "failure analysis"OR "defect"OR "error")

("software development"OR "software engineering") AND ("test-driven development"OR "test driven development") AND ("quality"OR "productivity") AND ("company"OR "industry"OR "college"OR "university") AND ("case") AND ("software quality"OR "software test"OR "unit testing"OR "unit tests"OR "unit test")

Com a *string* já definida, deve-se elencar os repositórios *online* que seriam utilizados para buscar estudos. Elegeu-se quatro repositórios:

- *ACM Digital Library* (*ACM Digital Library*, 2021);
- *IEEE Xplore Digital Library* (*IEEE Xplore Digital Library*, 2021);
- *ScienceDirect* (*ScienceDirect*, 2021).
- *Springer* (*Springer*, 2021).

Dando sequência ao processo de busca por estudos, aplicou-se a *string* de busca em cada um dos repositórios e foi feito o *download* do resultado total em formato BibTeX [3]. Cada arquivo, então, foi importado no Parsifal (Parsifal, 2019), para que, assim, ficasse documentado todos os artigos filtrados e facilitasse a sistematização desta revisão. Ao todo foram filtrados 1177 estudos, que dividiram-se da seguinte forma:

- *IEEE Xplore Digital Library*: 100 estudos.
- *ScienceDirect*: 100 estudos.
- *ACM Digital Library*: 81 estudos.
- *Springer*: 896 estudos.

Devido a quantidade de artigos encontrados no repositório *Springer*, houve a necessidade da *string* de busca ser diferente e, após a importação no Parsifal (Parsifal, 2019), ocorreu também um filtro manual para chegar a número de artigos mais enxutos para a presente revisão.

[3] Ferramenta de formatação usada em documentos LaTeX criada por Oren Patashnik e Leslie Lamport em 1985.

3.4 Critérios de inclusão e exclusão

Foram estabelecidos cinco critérios para inclusão dos artigos encontrados durante as pesquisas feitas nas bibliotecas digitais. Portanto, para um estudo ser incluído na revisão sistemática ele deve satisfazer todas as cinco condições listadas abaixo e não infringir qualquer um dos quatro critérios de exclusão apresentados em seguida.

Critérios de Inclusão:

i. Os artigos devem estar totalmente disponíveis para leitura e *download*;
ii. Os artigos devem ter sido publicados entre 2003 e 2020, sendo artigos científicos, trabalhos acadêmicos ou anais de conferências;
iii. Os artigos devem ter como principal tema a aplicação do *Test-Driven Development* no processo de desenvolvimento de software;
iv. Os trabalhos devem apresentar resultados sobre a aplicação do TDD no processo de desenvolvimento de software, de forma direta ou indireta;
v. Os artigos devem abordar resultados de experimentos ou abordar uma análise sobre o TDD.

Na Tabela 2, pode-se observar a quantidade de artigos que foram aceitos em cada um dos repositórios, quantos estavam duplicados e quantos foram rejeitados.

Além dos critérios de inclusão, foram definidos os critérios de exclusão. Dessa forma, artigos foram excluídos se atendessem qualquer critério de exclusão abaixo.

Critérios de Exclusão - referenciados na Tabela 3:

i. Artigos que não estão pautados unicamente na prática do TDD, analisando qualidade e/ou produtividade;
ii. Artigos que não contenham experimentos práticos de TDD;
iii. Trabalhos que não foram publicados em formato acadêmico;
iv. Artigos que não foram escritos em inglês;

Na Tabela 3, pode-se observar a quantidade de artigos que foram rejeitados em cada um dos repositórios, de acordo com cada critério de exclusão.

Esses critérios são de suma importância para manter o foco durante a leitura dos artigos e selecionar estudos de forma coesa.

Repositório	Aceitos	Duplicados	Rejeitados
ACM Digital Library	9	12	60
IEEE Xplore Digital Library	8	4	88
ScienceDirect	2	0	98
Springer	8	1	887

Tabela 2: Quantidade de artigos aceitos, duplicados e rejeitados, em cada um dos repositórios.

Repositório	Critério I	Critério II	Critério III	Critério IV
ACM Digital Library	52	8	0	0
IEEE Xplore Digital Library	86	2	0	0
ScienceDirect	93	1	4	0
Springer	800	87	0	0

Tabela 3: Quantidade de artigos excluídos de acordo com os critérios de exclusão para cada repositório.

3.5 Extração de dados e categorias

A fim de sistematizar e padronizar a extração dos estudos, criou-se duas categorias principais: "assunto relevante" e "assunto não relevante".

Dentro da categoria "assunto relevante", existem quatro subcategorias.

i. Experimento sobre TDD: aqui se encaixam artigos que relatam experimentos conduzidos e analisados pelo(s) próprio(s) autor(es);
ii. Experimento de terceiros e experimento próprio: engloba artigos que analisam experimentos de outros autores e, também, descrevem experimento(s) do(s) próprio(s) autor(es);
iii. Revisão Bibliográfica Sistemática sobre TDD: consiste em estudos que são revisões sistemáticas, gerando grande embasamento para o presente estudo;
iv. *Test-Driven Development*: estudos que são descritivos e detalham a prática do TDD durante o processo de desenvolvimento de software.

Cada estudo selecionado foi classificado com uma das 4 categorias de aceite possíveis, detalhadas acima. Essa classificação exerce papel fundamental pois no momento em que o estudo for analisado em detalhes já será de conhecimento a categoria em que ele se enquadra, se é um experimento do próprio autor, uma RBS, um descritivo sobre TDD ou um experimento de terceiros com experimento do próprio autor e, assim, cria-se um padrão de análise e armazenamento dos estudos.

4 Análise e resultados

Ao todo foram selecionados e analisados 27 estudos nesta revisão bibliográfica sistemática. Os estudos selecionados foram publicados entre 2003 e 2020 e apresentam: um comparativo entre as práticas de TDD e TLD (seja analisando experimentos de terceiros com um experimento próprio ou somente executando um experimento de própria autoria); a prática do TDD em detalhes; ou, uma RBS sobre o TDD.

A Tabela 4 lista todos os estudos selecionados para esta revisão. Ela apresenta, também, o ano de publicação, o ambiente em que os experimentos foram feitos, o perfil dos participantes e a categoria de cada artigo (apresentada na Seção 3.5).

Em relação ao ambiente em que o estudo foi conduzido, apresentam-se três possibilidades. O ambiente "Industrial" representa 37,07% dos estudos selecionados, já o ambiente "Acadêmico" representa 33,33%, o ambiente "Misto", que engloba profissionais e estudantes, corresponde a 11,11% e, por fim, os estudos que não apresentam ambiente totalizam 18,52% do total. Os estudos que não apresentam ambiente tem essa classificação pois se encaixam na categoria "Test-Driven Development" (apresentada na Seção 3.5).

Os perfis dos participantes foram divididos em quatro categorias: estudante, profissional, misto e não categorizado. As proporções dos perfis seguem os mesmo números dos ambientes citados acima, sendo que o ambiente "Industrial" equivale ao perfil "Profissional", o ambiente "Acadêmico" representa o perfil "Estudante", o ambiente "Misto" corresponde ao perfil "Misto" e, por fim, os estudos que não foram classificados com nenhum ambiente equivalem ao perfil que não foi categorizado.

Seguindo a análise dos resultados, para cada estudo foram analisados dois pontos principais, que serviram como base para responder às perguntas de pesquisa: a qualidade do software e a produtividade dos desenvolvedores. A qualidade do software está diretamente ligada à quantidade de defeitos em um sistema e, também, à complexidade ciclomática do código desenvolvido. Já a produtividade é a relação direta entre o que foi desenvolvido e o tempo que foi gasto para realizar a tarefa.

A fim de tornar a visualização dos resultados dos estudos mais legíveis, as Tabelas 5 e 6 apresentam os artigos que foram categorizados como "Experimento sobre TDD", "Experimento de terceiros e experimento próprio" ou "Revisão Bibliográfica Sistemática sobre TDD" e suas respectivas conclusões referente às duas variáveis que norteiam esta revisão: qualidade e produtividade. Para isso, nessas tabelas, existem quatro classificações para os resultados obtidos por cada estudo. São elas:

Artigo	Ano	Ambiente	Participantes	Categoria
(Bhat and Nagappan, 2006)	2006	Industrial	Profissionais	Experimento sobre TDD
(Canfora et al., 2006)	2006	Industrial	Profissionais	Experimento sobre TDD
(Crispin, 2006)	2006	-	-	TDD
(Dogša and Batič, 2011)	2011	Industrial	Profissioanais	Experimento sobre TDD
(Fucci, 2014)	2014	Acadêmico	Estudantes	Experimento sobre TDD
(Fucci et al., 2014b)	2014	Acadêmico	Estudantes	Experimento sobre TDD
(Fucci et al., 2014a)	2014	Acadêmico	Estudantes	Experimento de terceiros e próprio
(George and Williams, 2003)	2003	Industrial	Profissionais	Experimento sobre TDD
(Gupta and Jalote, 2007)	2007	Acadêmico	Estudantes	Experimento sobre TDD
(Hammond and Umphress, 2012)	2012	-	-	TDD
(Janzen and Saiedian, 2005)	2005	-	-	TDD
(Karac and Turhan, 2018)	2018	-	-	TDD
(Karamat and Jamil, 2006)	2006	Acadêmico	Estudantes	Experimento sobre TDD
(Latorre, 2014)	2014	Industrial	Profissionais	Experimento sobre TDD
(Madeyski and Szała, 2007)	2007	Acadêmico	Estudantes	Experimento sobre TDD
(Mäkinen and Münch, 2014)	2014	Misto	Misto	RBS sobre TDD
(Maximilien and Williams, 2003)	2003	Industrial	Profissionais	Experimento sobre TDD
(Nagappan et al., 2008)	2008	Industrial	Profissionais	Experimento sobre TDD
(Pančur and Ciglarič, 2011)	2011	Acadêmico	Estudantes	Experimento de terceiros e próprio
(Rafique and Mišić, 2013)	2013	Misto	Misto	RBS sobre TDD
(Romano et al., 2017)	2017	Industrial	Profissionais	Experimento sobre TDD
(Sangwan and LaPlante, 2006)	2006	-	-	TDD
(Siniaalto and Abrahamsson, 2008)	2007	Misto	Misto	RBS sobre TDD
(Tosun et al., 2017)	2017	Industrial	Profissionais	Experimento sobre TDD
(Tosun et al., 2018)	2018	Industrial	Profissionais	Experimento sobre TDD
(Vu et al., 2009)	2009	Acadêmico	Estudantes	Experimento sobre TDD
(Xu et al., 2009)	2009	Acadêmico	Estudantes	Experimento sobre TDD

Tabela 4: Estudos analisados nesta revisão bibliográfica sistemática com seus respectivos dados: ano, ambiente, participantes e categoria

- **Aumento**: o estudo aponta que houve aumento da variável;
- **Sem Aumento**: o estudo não aponta ganho e nem perda na variável;
- **Diminuição**: o estudo aponta que houve perda na variável;
- **Inconclusivo**: o estudo não tem um parecer final sobre ganho ou perda na variável;

4.1 Qualidade

Em relação à variável qualidade, nenhum estudo apontou que o TDD diminua a qualidade do software quando comparado com o desenvolvimento TLD.

Os dados obtidos com esta revisão encontram-se na Tabela 5. Entre os estudos analisados, 54,55% apontaram um aumento na qualidade (seja ela interna ou externa); 27,27% não apontaram nenhuma melhoria na qualidade; e 18,18% são inconclusivos em relação à variável qualidade. Com o uso do TDD, a preocupação com a escrita de testes se torna maior, visto que é requerido a criação de testes unitários que desenvolvem gradualmente pequenas partes da funcionalidade até que um recurso seja totalmente implementado (Vu et al., 2009). A maioria dos estudos selecionados, apontam que o TDD aumenta a qualidade de modo geral, conforme previa Kent Beck quando apresentou a metodologia XP, que faz uso do TDD (Beck, 2010).

Pode-se observar, que em (Tosun et al., 2018) os resultados do experimento indicam que o TDD tem um efeito positivo na cobertura de código em *branches*, aumentando, assim, a qualidade. Também, em (George and Williams, 2003) os autores mostram que o desenvolvimento de software com TDD, aparentemente, mesmo considerando as limitações do estudo, produz código com qualidade superior quando comparado com o código desenvolvido com uma prática mais tradicional, por exemplo, o TLD, utilizando um conjunto de casos de teste de caixa preta.

4.2 Produtividade

Analisando os estudos selecionados, levando em consideração a variável produtividade, pode-se notar que o TDD, de modo geral, não apresenta aumento na produtividade dos desenvolvedores de software, assim como é retratado no experimento de (Vu et al., 2009), onde o grupo *Test-Last*, segundo os autores do experimento, parece ser mais produtivo que o *Test-First* (TDD).

Na Tabela 6, estão listados os estudos que analisam a produtividade do TDD comparando-o com o TLD. Na coluna "Aumento" estão os estudos que apontam melhoria na produtividade do desenvolvedor quando o TDD foi utilizado. Na coluna "Sem Aumento" constam os estudos que não apresentam ganhos ou perdas de produtividade. Já a coluna "Diminuição" engloba os estudos que apresentam

Qualidade

Aumento	Sem Aumento	Diminuição	Inconclusivo
(Bhat and Nagappan, 2006)	(Fucci, 2014)		(Canfora et al., 2006)
(Dogša and Batič, 2011)	(Fucci et al., 2014a)		(Fucci et al., 2014b)
(George and Williams, 2003)	(Latorre, 2014)		(Karamat and Jamil, 2006)
(Gupta and Jalote, 2007)	(Madeyski and Szała, 2007)		(Romano et al., 2017)
(Mäkinen and Münch, 2014)	(Pančur and Ciglarič, 2011)		
(Maximilien and Williams, 2003)	(Vu et al., 2009)		
(Nagappan et al., 2008)			
(Rafique and Mišić, 2013)			
(Siniaalto and Abrahamsson, 2008)			
(Tosun et al., 2017)			
(Tosun et al., 2018)			
(Xu et al., 2009)			

Tabela 5: Classificação dos artigos selecionados analisando a qualidade do software, comparando TDD com TLD

Produtividade

Aumento	Sem Aumento	Diminuição	Inconclusivo
(Gupta and Jalote, 2007)	(Fucci, 2014)	(Bhat and Nagappan, 2006)	(Dogša and Batič, 2011)
(Madeyski and Szała, 2007)	(Fucci et al., 2014a)	(Canfora et al., 2006)	(Fucci et al., 2014b)
	(Latorre, 2014)	(George and Williams, 2003)	(Karamat and Jamil, 2006)
	(Nagappan et al., 2008)	(Vu et al., 2009)	(Mäkinen and Münch, 2014)
	(Pančur and Ciglarič, 2011)		(Maximilien and Williams, 2003)
	(Siniaalto and Abrahamsson, 2008)		(Rafique and Mišić, 2013)
	(Tosun et al., 2017)		(Romano et al., 2017)
	(Xu et al., 2009)		(Tosun et al., 2018)

Tabela 6: Classificação dos artigos selecionados analisando a produtividade dos desenvolvedores de software, comparando TDD com TLD.

uma queda na produtividade dos desenvolvedores quando os mesmos fizeram uso do TDD. Por fim, a coluna "Inconclusivo" apresentam os estudos que não conseguiram ter um parecer final sobre o uso da prática.

Nesta revisão, somente dois estudos apontaram que quando o TDD foi aplicado a produtividade aumenta, conforme observa-se em (Gupta and Jalote, 2007) e (Madeyski and Szała, 2007), representando, assim, 9,09% do estudos selecionados. Enquanto que 36,36% foram classificados como "Sem Aumento". Seguindo a análise, 27,27% dos estudos apontaram que quando o TDD foi utilizado a produtividade diminui e, por fim, 27,27% dos estudos são inconclusivos.

5 Discussão

Os estudos que foram selecionados (e não foram categorizados como "TDD") comparam o uso de TDD com TLD e mostram uma evidente divisão em dois grandes grupos quanto ao ambiente do experimento: acadêmico e industrial. Como visto na Tabela 4, a participação dos estudos nessa categoria é equilibrada: o ambiente industrial representa 45,45%, o acadêmico 40,91% e, por fim, o menor grupo, que é o ambiente misto, representa 13,64%.

Cada estudo selecionado fez uso de métricas e indicadores diferentes entre si para extrair informações. Logo, isso dificultou a análise e comparação dos resultados. Sendo assim, esta revisão considerou os efeitos constatados nos estudos analisados, independentemente das métricas e linguagem de programação utilizadas.

Todavia, pode-se observar que a cobertura do código fonte com testes unitários foi a métrica mais utilizada como indicador para determinar a qualidade do software.

A cobertura do código indica o percentual do código fonte que foi executado pelo menos uma vez durante os testes. Logo, um código de produção está totalmente coberto por testes unitários quando sua cobertura atinge 100%. Entretanto, essa cobertura não garante que os testes unitários estejam implementados da melhor forma e nem que o sistema está livre de defeitos ou que não apresentará falhas durante a sua utilização.

A maior parte dos estudos presentes nesta revisão (54,55%), Fig. 2, indicam uma melhoria na qualidade do software quando o TDD foi utilizado, sendo que nenhum estudo apontou uma diminuição na qualidade. As métricas que mais foram utilizadas nos experimentos analisados, como forma de mensurar a qualidade, foram a complexidade ciclomática do código desenvolvido e a porcentagem da cobertura de código com testes unitários.

Deve-se ressaltar que os estudos (Vu et al., 2009), (Fucci, 2014), (Fucci et al., 2014a) e (Pančur and Ciglarič, 2011) não apontaram melhora na qualidade quando o TDD foi aplicado. Nestes estudos, os desenvolvedores que fizeram uso de TLD produziram código com qualidade semelhante ao código dos que aplicaram TDD. O que

Figura 2: Variação da Qualidade do Software

Figura 3: Variação da Produtividade da Equipe

pode-se notar aqui é que todos os estudos que não apontam aumento na qualidade foram desenvolvidos em ambiente acadêmico, onde os desenvolvedores eram estudantes universitários ou ex-universitários recém formados, com pouca experiência com programação. O TDD foi um conceito relativamente novo para muitos dos estudantes que estavam acostumados com a abordagem tradicional (Vu et al., 2009). Neste mesmo estudo (Vu et al., 2009), o grupo *Test-Last* pareceu ser mais produtivo, segundo os autores, que o equivalente no *Test-First* (TDD) em termos de horas por recurso implementado e total de recursos concluídos. Todas as três equipes, participantes do experimento, escreveram sobre a mesma quantidade de linhas de código de produção, mas o grupo que utilizou o *Test-Last* superou os grupos que utilizaram o *Test-First* com mais de sete vezes a quantidade de código de teste. Embora os estudos acadêmicos apresentem, de forma majoritária, uma estabilidade na qualidade quando o TDD é aplicado, existem também estudos que mostram que, mesmo em ambiente acadêmico, o TDD garanta mais qualidade (Xu et al., 2009).

Com relação à produtividade, a métrica mais utilizada, nos estudos analisados, foi o tempo gasto para desenvolvedor cada nova funcionalidade do software. Além desta, alguns artigos, por exemplo (Maximilien and Williams, 2003), utilizam a métrica Kilo Lines of Code (KLOC), que indica quantas mil linhas de código um desenvolvedor ou uma equipe foi capaz de produzir. Considerando todos os estudos selecionados, é notório que apenas um estudo (Gupta and Jalote, 2007) apontou aumento da produtividade dos desenvolvedores quando o TDD foi aplicado. Neste estudo, de Gupta e Jalote, os resultados sugeriram que a abordagem TDD fosse mais eficiente, pois exigia menos esforço de desenvolvimento e também permitia pequenas melhorias na produtividade do desenvolvedor. Do lado oposto desta variável, cerca de 27,27% dos estudos apontaram que o TDD diminui a produtividade, como em (Vu et al., 2009) onde o grupo *Test-Last* pareceu ser mais produtivo que o equivalente no *Test-First* em termos de horas por *feature* implementado e total de *features* concluídas. Isso ocorre pois existem evidências estatísticas de que o TDD requer mais tempo que o TLD, devido a sua iteratividade (Canfora et al., 2006).

A avaliação do aumento da produtividade foi menos enfática nos estudos analisados, chegando a apenas 9% - Fig. 3, mas devemos observar que a melhoria da produtividade deve ser considerada a médio prazo, considerando a influência nas atividades de manutenção do código, além do número de linhas produzidas em um determinado período de tempo. A avaliação da produtividade, vista apenas pelo tempo total de desenvolvimento de uma nova funcionalidade e escrita dos testes para ela, pode não refletir adequadamente os benefícios alcançados em um desenvolvimento em ambiente real. Se o TDD levar à escrita de um número maior de testes, isso pode refletir benefícios na fase de manutenção do código. Assim sendo, no futuro, o desenvolvedor que vier a dar manutenção no código pode não ser o mesmo que escreveu o código original. Havendo mais testes unitários escritos aumenta-se a confiança de que não haverá risco de regressão ao se modificar o código existente.

5.1 Comparativo entre os ambientes dos experimentos analisados

Conforme já foi abordado no presente estudo, divide-se, de forma quase que igualitária os ambientes em que foram desenvolvidos os experimentos dos artigos selecionados. Considerando somente os estudos onde ocorrem experimentos com TDD, 45,45% deles foram feitos em ambiente industrial e 40,91% em ambiente acadêmico. Contrastando os dois ambientes, pode-se notar certa heterogeneidade em seus respectivos resultados.

No ambiente industrial, 70% dos estudos apontaram aumento na qualidade do software desenvolvido quando o TDD foi utilizado, conforme pode ser visto em (Bhat and Nagappan, 2006), 10% evidenciam neutralidade, não apresentam aumenta e nem diminuição (Latorre, 2014),

enquanto que os 20% restantes foram inconclusivos. Seguindo a análise, no ambiente acadêmico, somente 22% dos estudos indicaram aumento na qualidade (Xu et al., 2009), enquanto que 56% não apontaram nenhum aumento e 22% foram inconclusivos. Neste ponto, é interessante notar que há uma discrepância em relação ao aumento da qualidade nos dois ambientes. Pode-se atribuir que mais estudos industriais apontaram aumento na qualidade visto que os profissionais que participaram dos experimentos já possuíam experiência prévia com programação e que no, ambiente acadêmico, os desenvolvedores participantes eram graduandos ou recém-formados com pouca experiência prática.

Seguindo para analisar a produtividade, tem-se que no ambiente industrial 40% retrataram uma diminuição nesta variável, conforme nota-se em (Dogša and Batič, 2011), 30% dos estudos são inconclusivos, em conformidade com (Canfora et al., 2006) e (Tosun et al., 2018), respectivamente. E , por fim, 30% são neutros, ou seja, não apresentam melhora ou piora significativa conforme foi analisado em (Latorre, 2014), (Nagappan et al., 2008) e (Tosun et al., 2017). Já no ambiente acadêmico, é relevante ressaltar que 22% dos estudos relataram um aumento na produtividade. Além disso, 14% apontou para uma diminuição, sendo que os demais estudos dividem-se entre neutralidade ou inconclusão. Fica evidente que no ambiente industrial, os resultados se comportam apenas de duas maneiras, tornando a análise mais fácil. Do outro lado, no ambiente acadêmico, existem diferentes tipos de resultados em relação à produtividade, podendo-se atribuir estas divergências à falta de experiência dos estudantes e/ou recém-formados que participaram dos experimentos e, também à curva de aprendizado do TDD, que está atrelada à mudança de mentalidade dos desenvolvedores, as quais impactam negativamente a produtividade. Mesmo com essa variabilidade nos resultados acadêmicos, a maioria dos experimentos apontaram diminuição ou inconclusão sobre a produtividade.

Sendo assim, evidencia-se que o TDD não traz aumento significativo na produtividade em empresas ou universidades, de acordo com os estudos selecionados.

5.2 Hipóteses de pesquisa

Após ter discutido em detalhes, as duas variáveis guias desta revisão, qualidade e produtividade, e ter analisado os diferentes ambientes nos quais foram realizados experimentos, aqui são apresentadas as hipóteses que motivaram o presente estudo e suas respectivas conclusões após a revisão ter sido desenvolvida.

P1: Os custos envolvidos na aplicação do *Test-Driven Development* compensam seus benefícios?

R1: Os estudos que foram filtrados nessa revisão apontam diferentes resultados sobre o uso do *Test-Driven Development* no desenvolvimento de software quando comparados com o *Test-Last Development*. Em (George and Williams, 2003), os desenvolvedores que fizeram uso de TDD produziram código de maior qualidade. Neste mesmo estudo, os desenvolvedores do grupo TDD superaram os padrões da indústria nos três tipos de cobertura de código (métodos, *branches* e *statement*). No entanto, demoraram mais tempo, em média, para concluir o trabalho.

Deste modo, com base na proporção dos estudos selecionados e em estudos como o citado acima, não se pode afirmar ou negar que o custos envolvidos na aplicação do *Test-Driven Development* compensam seus benefícios embora alguns estudos, como (Fucci et al., 2014b), argumentem que o esforço da implementação do TDD no fluxo de trabalho dos desenvolvedores não gera um retorno viável do investimento, pelo menos a curto prazo.

P2: Quando o *Test-Driven Development* for aplicado, a quantidade de defeitos encontrados no software nas fases mais avançadas do desenvolvimento será menor?

R2: A quantidade de defeitos encontrados nas fases mais avançadas do processo de desenvolvimento de software é menor quando o TDD é aplicado. Isso pode ser visto e analisado em (Maximilien and Williams, 2003) onde, com uso do TDD, o teste unitário realmente acontece. Com o TDD, o teste unitário se torna parte integrante do desenvolvimento do código. Como resultado, foi alcançada uma melhoria de 50% na taxa de defeitos de testes funcionais do sistema.

Também, descobriu-se que os casos de teste escritos para uma tarefa TDD têm uma capacidade de detecção de defeitos mais alta do que os casos de teste escritos para uma tarefa TLD. (Tosun et al., 2018). Sendo assim, menos defeitos chegam às fases finais do desenvolvimento, onde ocorrem testes funcionais.

P3: Quando o *Test-Driven Development* for aplicado, muitos problemas serão detectados e removidos ainda nas fases iniciais, muito provavelmente, pelos próprios desenvolvedores?

R3: Em relação a detecção e remoção de defeitos ainda nas fases iniciais, pode-se concluir que isso realmente ocorre devido a análise feita na Pergunta 2, onde os defeitos nas fases tardias são menores quando o TDD é utilizado. Logo, pode-se concluir que estes defeitos foram tratados pelos próprios desenvolvedores antes mesmo de chegar na fase de testes funcionais, pois com o TDD, o teste unitário se torna parte do desenvolvimento, e não da fase de teste. A iteratividade do TDD alinhada com refatorações constantes promovem essa detecção e remoção precoce dos defeitos.

6 Ameaças à validade

As principais ameaças à validade desta revisão são:

- A variabilidade dos seres humanos em relação às suas habilidades e motivação;
- Possuir um espaço amostral reduzido. Mesmo assim, todos os processos de uma revisão sistemática foram devidamente executados;
- Nos experimentos dos estudos selecionados e analisados, os desenvolvedores que usaram TDD podem ter tido um ganho de motivação, a fim de produzirem um código com maior qualidade, já que neste caso, o TDD representa uma nova abordagem que visa almejar um código entregável com qualidade superior.
- O TDD pode ter sido um conceito relativamente novo para muitos dos estudantes que estavam familiarizados somente com a abordagem tradicional (TLD). Isso pode, de certa forma, explicar o porque todos os estudos que não apontaram melhora na qualidade foram realizados

em ambiente acadêmico;
- Esta revisão não conta com estudos divididos de forma igualitária no que se refere ao ambiente de desenvolvimento e perfil dos participantes dos experimentos analisados. Entretanto, esta divergência não impacta os resultados finais deste estudo, visto que estas categorizações foram somente utilizadas a fim de documentar e analisar os estudos selecionados.

7 Conclusão

Esta revisão bibliográfica sistemática foi realizada com o objetivo de identificar, selecionar e analisar estudos empíricos que abordam o uso da prática TDD no desenvolvimento de software, sempre tendo como foco avaliar os efeitos originados na qualidade do software desenvolvido e na produtividade dos desenvolvedores.

Sendo assim, todos os estudos selecionados estão mapeados na Tabela 4, e os resultados da análise são apresentados na Seção 4, enquanto a Seção 5 apresenta uma discussão comparando os resultados obtidos e como eles respondem as hipóteses do autor.

Os estudos selecionados foram filtrados a partir de bibliotecas online, onde foram selecionados artigos científicos, trabalhos acadêmicos e anais de conferências.

É importante ressaltar que os resultados da análise desses artigos mostraram que, em quase metade deles, não houve um resultado claro sobre a relação entre custo e benefício do TDD, sendo assim, seus resultados foram inconclusivos. Apenas 4,5% dos estudos apontaram o uso do TDD como positivo de modo geral, ou seja, neste estudo a qualidade aumentou e a produtividade também, não havendo necessidade de empenhar mais tempo durante o desenvolvimento do experimento. Enquanto isso 9%, representam os estudos que descrevem o TDD como uma prática negativa, ou seja, não aumenta a qualidade e eleva o tempo gasto durante o desenvolvimento do software.

Esta revisão buscou entender a relação de custo-benefício do TDD, mas não chegou a ser conclusiva, sendo assim, este estudo não pode afirmar ou negar que o uso da prática realmente traz melhorias compensando seus custos. Todavia, pode-se afirmar que o TDD traz mais qualidade para o software e, principalmente, para os testes unitários que são desenvolvidos pelo próprio programador.

Já em relação a quantidade de defeitos encontrados nas fases mais avançadas do processo de desenvolvimento de software pode-se afirmar que ela é menor quando o TDD é aplicado, por conta do desenvolvimento dos testes unitários. Dando sequência nesta análise, esta revisão, também, buscou compreender se os defeitos que não foram encontrados nas fases mais avançadas do desenvolvimento de software teriam sido tratados pelos próprios desenvolvedores durante a codificação. Ao final deste estudo, pode-se considerar que esta hipótese é verdadeira, devido aos testes unitários e refatoração que ocorrem de forma iterativa no TDD.

Por fim, o presente estudo elenca as oportunidades para futuros estudos, a fim de contribuir ainda mais com a temática aqui abordada.

- Validar com um experimento prático a hipótese de que o TDD traz um ganho na relação custo-benefício;
- Investigar como o TDD se comporta quando é utilizado em sistemas legados, visto que a maioria dos estudos focam no desenvolvimento de novos sistemas;
- Realizar experimentos usando projetos reais em ambientes corporativos, considerando um período maior de desenvolvimento.

As oportunidades apresentadas evidenciam que ainda existem algumas lacunas que os pesquisadores de engenharia de software precisam investigar para elucidar e fornecer novas evidências sobre os efeitos produzidos pelo TDD.

8 Agradecimentos

Ao professor Julio Cesar de Lemos por ter auxiliado os autores na utilização do ambiente Parsifal.

Referências

ACM Digital Library (2021). Disponível em https://dl.acm.org/.

Beck, K. (2003). *Test-Driven Development: By Example*, Addison-Wesley.

Beck, K. (2010). *Extreme Programming Explained: Embrace Change*, Addison-Wesley.

Bhat, T. and Nagappan, N. (2006). Evaluating the efficacy of test-driven development: Industrial case studies, *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, Association for Computing Machinery, New York, NY, USA, p. 356–363. https://doi.org/10.1145/1159733.1159787.

Canfora, G., Cimitile, A., Garcia, F., Piattini, M. and Visaggio, C. A. (2006). Evaluating advantages of test driven development: A controlled experiment with professionals, *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, Association for Computing Machinery, New York, NY, USA, p. 364–371. https://doi.org/10.1145/1159733.1159788.

Conforto, E. C., Amaral, D. C. and da Silva, S. L. (2011). Roteiro para revisão bibliográfica sistemática: aplicação no desenvolvimento de produtos e gerenciamento de projetos, *VIII Congresso Brasileiro de Gestão de Desenvolvimento de Produto* . "https://edisciplinas.usp.br/pluginfile.php/2205710/mod_resource/content/1/Roteiro%20para%20revis%C3%A3o%20bibliogr%C3%A1fica%20sistem%C3%A1tica.pdf.

Crispin, L. (2006). Driving software quality: How test-driven development impacts software quality, *IEEE Software* **23**(6): 70–71. https://doi.org/10.1109/MS.2006.157.

Dogša, T. and Batič, D. (2011). The effectiveness of test-driven development: an industrial case study, *Software Quality Journal* (19): 643 – 661. https://doi.org/10.1007/s10664-016-9490-0.

Fucci, D. (2014). Understanding the dynamics of test-driven development, *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, Association for Computing Machinery, New York, NY, USA, p. 690–693. https://doi.org/10.1145/2591062.2591086.

Fucci, D., Turhan, B. and Oivo, M. (2014a). Conformance factor in test-driven development: Initial results from an enhanced replication, *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2601248.2601272.

Fucci, D., Turhan, B. and Oivo, M. (2014b). Impact of process conformance on the effects of test-driven development, *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2652524.2652526.

George, B. and Williams, L. (2003). An initial investigation of test driven development in industry, *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, Association for Computing Machinery, New York, NY, USA, p. 1135–1139. https://doi.org/10.1145/952532.952753.

Gupta, A. and Jalote, P. (2007). An experimental evaluation of the effectiveness and efficiency of the test driven development, *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pp. 285–294. https://doi.org/10.1109/ESEM.2007.41.

Hammond, S. and Umphress, D. (2012). Test driven development: The state of the practice, *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE '12, Association for Computing Machinery, New York, NY, USA, p. 158–163. https://doi.org/10.1145/2184512.2184550.

IEEE Xplore Digital Library (2021). Disponível em https://ieeexplore.ieee.org/Xplore/home.jsp.

Janzen, D. and Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction, *Computer* **38**(9): 43–50. https://doi.org/10.1109/MC.2005.314.

Karac, I. and Turhan, B. (2018). What do we (really) know about test-driven development?, *IEEE Software* **35**(4): 81–85.
URL: *https://ieeexplore.ieee.org/document/5604358*

Karamat, T. and Jamil, A. N. (2006). Reducing test cost and improving documentation in tdd (test driven development), *Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06)*, pp. 73–76. https://doi.org/10.1109/SNPD-SAWN.2006.59.

Latorre, R. (2014). A successful application of a test-driven development strategy in the industrial environment, *Empirical Software Engineering* (19): 753 – 773. https://doi.org/10.1007/s10664-013-9281-9.

Madeyski, L. and Szała, Ł. (2007). The impact of test-driven development on software development productivity - an empirical study, *in* P. Abrahamsson, N. Baddoo, T. Margaria and R. Messnarz (eds), *Software Process Improvement*, Springer Berlin Heidelberg, pp. 200–211. https://doi.org/10.1007/978-3-540-75381-0_18.

Mäkinen, S. and Münch, J. (2014). Effects of test-driven development: A comparative analysis of empirical studies, *in* D. Winkler, S. Biffl and J. Bergsmann (eds), *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering*, Springer International Publishing, Cham, pp. 155–169. https://doi.org/10.1007/978-3-319-03602-1_10.

Mamédio da Costa Santos, C., Andrucioli de Mattos Pimenta, C. and Roberto Cuce Nobre, M. (2007). The pico strategy for the research question construction and evidence search, *Revista Latino-Americana de Enfermagem*, *note=http://dx.doi.org/10.1590/S0104-11692007000300023* .

Maximilien, E. M. and Williams, L. (2003). Assessing test-driven development at ibm, *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 564–569. https://doi.org/10.1109/ICSE.2003.1201238.

Nagappan, N., Maximilien, E. M., Bhat, T. and Williams, L. (2008). Realizing quality improvement through test driven development: results and experiences of four industrial teams, *Empirical Software Engineering* (13): 289 – 302. https://doi.org/10.1007/s10664-008-9062-z.

Pančur, M. and Ciglarič, M. (2011). Impact of test-driven development on productivity, code and tests: A controlled experiment, *Information and Software Technology* **53**(6): 557 – 573. https://doi.org/10.1016/j.infsof.2011.02.002.

Parsifal (2019). Parsifal. Disponível em https://parsif.al/.

Rafique, Y. and Mišić, V. B. (2013). The effects of test-driven development on external quality and productivity: A meta-analysis, *IEEE Transactions on Software Engineering* **39**(6): 835–856. https://doi.org/10.1109/TSE.2012.28.

Romano, S., Fucci, D., Scanniello, G., Turhan, B. and Juristo, N. (2017). Findings from a multi-method study on test-driven development, *Information and Software Technology* **89**: 64 – 77. https://doi.org/10.1016/j.infsof.2017.03.010.

Sangwan, R. S. and LaPlante, P. A. (2006). Test-driven development in large projects, *IT Professional* **8**(5): 25–29. https://doi.org/10.1109/MITP.2006.122.

ScienceDirect (2021). Disponível em https://www.sciencedirect.com/.

Siniaalto, M. and Abrahamsson, P. (2008). Does test-driven development improve the program code? alarming results from a comparative case study, *in* B. Meyer, J. R. Nawrocki and B. Walter (eds), *Balancing Agility and Formalism in Software Engineering*, Springer Berlin Heidelberg, pp. 143–156. https://doi.org/10.1007/978-3-540-85279-7_12.

Springer (2021). Disponível em https://link.springer.com/.

Suleman Sarwar, M. M., Shahzad, S. and Ahmad, I. (2013). Cyclomatic complexity: The nesting problem, *Eighth International Conference on Digital Information Management (ICDIM 2013)*, pp. 274–279. https://doi.org/10.1109/ICDIM.2013.6693981.

Tosun, A., Ahmed, M., Turhan, B. and Juristo, N. (2018). On the effectiveness of unit tests in test-driven development, *Proceedings of the 2018 International Conference on Software and System Process*, ICSSP '18, Association for Computing Machinery, New York, NY, USA, p. 113–122. https://doi.org/10.1145/3202710.3203153.

Tosun, A., Dieste, O., Fucci, D., Vegas, S., Turhan, B., Santos, H. E. A., Oivo, M., Toro, K., Jarvinen, J. and Juristo, N. (2017). An industry experiment on the effects of test-driven development on external quality and productivity, *Empirical Software Engineering* (22): 2763 – 2805. https://doi.org/10.1007/s10664-016-9490-0.

Vu, J. H., Frojd, N., Shenkel-Therolf, C. and Janzen, D. S. (2009). Evaluating test-driven development in an industry-sponsored capstone project, *2009 Sixth International Conference on Information Technology: New Generations*, pp. 229–234.

What is a PICOC? (2019). Disponível em https://www.cebma.org/faq/what-is-a-picoc/.

Xu, S., Li, T. and Ishii, N. (2009). *Evaluation of Test-Driven Development: An Academic Case Study*, Springer Berlin Heidelberg, pp. 229–238. https://doi.org/10.1007/978-3-642-05441-9_20.