INTERFACE TECNOLÓGICA

UTILIZAÇÃO DO TDD COMO METODOLOGIA NO DESENVOLVIMENTO DE *SOFTWARES*

TDD AS A METHODOLOGY IN SOFTWARE DEVELOPMENT

David Chaves Ferreira – davi.ch.fe@gmail.com
Faculdade de Tecnologia de Taquaritinga (Fatec) – Taquaritinga – SP – Brasil

Giuliano Scombatti Pinto – giuliano.pinto@fatec.sp.gov.br
Faculdade de Tecnologia de Taquaritinga (Fatec) – Taquaritinga – SP – Brasil

DOI: 10.31510/infa.v19i2.1536
Data de submissão: 01/09/2022
Data do aceite: 28/11/2022
Data da publicação: 20/12/2022

RESUMO

Conforme o desenvolvimento de *softwares* cresce e sua complexidade aumenta, o número de *bugs* também pode aumentar; por isso, surge a necessidade de se ter testes para validar as funcionalidades do sistema. Esses testes dão ao desenvolvedor a segurança da implementação do código novo sem o medo de criar novos *bugs*, de modo que sempre que é implementado algo novo no sistema, um teste é criado para garantir que essa nova funcionalidade está certa, também não afetando as já existentes. Além disso, os testes também servem como uma documentação para quem lê o código. Portanto, quanto mais claros os testes forem, mais fácil será fazer sua manutenção. No entanto, escrever testes não é tão simples quanto parece, pois se não for usada a metodologia correta, isso pode trazer mais complexidade do que o necessário. Este estudo tem como objetivo demonstrar como tirar maior proveito dos testes, utilizando a metodologia TDD e também como essa metodologia funciona na escrita do código. Ao final deste artigo, espera-se que o leitor tenha uma base sobre como começar a desenvolver códigos usando essa metodologia e como seu uso influencia positivamente o desenvolvimento de *softwares*.

Palavras-chave: Testes; TDD; Metodologia TDD.

ABSTRACT

As software development increases, so does its complexity. The number of bugs might also increase, so there is a need of testing it to validate the system functionality. Such tests provide the developer safety while implementing security as codes, so there will be no new bugs in the end. Furthermore, when something new is implemented in the system, a test is created to ensure that the new functionality works, not affecting the existing ones. Besides that, tests also have the role of recording data for those ones that already read codes. That said, the more tests, the easier it will be to maintain them. However, creating tests is not as simple as it seems, given that if someone does not use a proper methodology, things might get more complicated than necessary. Finally, this research aims at showing how to make the most of tests by using the

TDD methodology, as well as how it works in creating codes. In the end of this paper, we expect that our reader gets more acquainted on how he or she might start creating codes using the methodology aforementioned and how to get positive results on software development.

Keywords: Tests; TDD; TDD Methodology.

1 INTRODUÇÃO

A profissão de desenvolvedor de *softwares* cresceu muito de uns anos para cá. Hoje em dia, não basta o programador saber codificar, ele também precisa ser analítico e identificar *bugs* e falhas nos sistemas. Apenas uma pequena fração do tempo de trabalho do desenvolvedor é dedicada à escrita de códigos para funcionalidades novas, dado que ele gasta a maior parte de suas horas do dia procurando e corrigindo *bugs* enquanto cria outras coisas. Corrigir *bugs*, na verdade, é bem simples; o mais difícil é encontrá-los em tempo hábil, para que os danos causados por estes não sejam tão grandes. Mesmo assim, sempre existe a chance de que quando um é corrigido, outro é criado logo em seguida (FOWLER, 2020).

Ao começar a escrever testes para o código e a executá-los logo em seguida, a produtividade do programador referente à localização de novos *bugs* também aumenta, pois sempre que um código novo é implementado, a suíte de testes é rodada. Além disso, caso os testes da suíte falhem, então será mais fácil identificar que isso se deu por causa do novo código inserido, porque aqueles testes funcionavam antes. Desse modo, o código passa a ser autotestável, e se os testes automatizados falharem, o local do erro torna-se evidente. Assim, os testes servem como uma espécie de detector de *bugs*, e o trabalho que, antes, levava horas de depuração para encontrar *bugs*, agora é totalmente automatizado (FOWLER, 2020).

Aplicando a metodologia TDD na escrita de testes, é possível criar um *software* muito mais assertivo e com menos falhas e *bugs*. É claro que o *software* não ficará totalmente sem *bugs*; entretanto, os *bugs* e as falhas serão reduzidos significativamente, porque seu uso influencia positivamente o código gerado. De modo geral, o resultado é que se torna mais simples testá-lo, sendo também uma ferramenta muito útil para aprender como se escrever um código testável (GPIRESS, 2020).

Um dos principais benefícios que a metodologia TDD tem em comparação com outras é que, ao usá-la, o programador começará a implementação do teste primeiro; depois, partirá para o código. Isso dá ao programador a liberdade de pensar na implementação do código antes

de desenvolvê-lo, de modo que ele entenderá mais consistentemente os requisitos técnicos do código e a melhor forma de implementá-lo enquanto escreve o teste (DIGITE, 2022).

As seções deste artigo abordarão desde o surgimento do TDD, passando por sua evolução, regras, técnicas para otimizar seu uso no dia a dia, problemas causados ao sistema pela falta de atenção em relação a um teste e a forma por meio da qual o TDD age na escrita de códigos limpos. Portanto, o objetivo deste trabalho é demonstrar, de maneira simples, os benefícios que a metodologia TDD traz para o desenvolvimento de *softwares* na identificação e correção de *bugs* e na clareza do código gerado.

A metodologia utilizada foi a pesquisa bibliográfica, que foi realizada em livros e artigos científicos a fim de entender os procedimentos relatados por diversos especialistas na área de desenvolvimento em relação à escrita de códigos e testes usando a metodologia TDD.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão apresentados, com base em referências bibliográficas, os conceitos teóricos abordados neste artigo. Esses conceitos farão com que o leitor entenda e reflita melhor sobre o tema.

2.1 O que é TDD?

O Desenvolvimento Orientado a Testes (TDD, na sigla em inglês), é uma metodologia para escrever o teste antes do código principal, para que, depois que ele estiver funcionando, seja melhorado (DIGITE, 2022).

A princípio, parece que o TDD foi criado para escrever testes. Entretanto, seu maior objetivo é outro. Utilizar essa metodologia significa escrever códigos limpos, testáveis e refatoráveis continuamente (DIGITE, 2022).

Além disso, o TDD traz outros benefícios, como:

- Documentação: Os testes servirão como uma espécie de documentação, as especificações nunca estarão desatualizadas (DIGITE, 2022).
- Código limpo: O código escrito ficará fácil de entender e de ser modificado (DIGITE, 2022).
- Segurança: Aplicando o teste, o desenvolvedor criará uma rede de segurança para realizar mudanças no código sem medo (DIGITE, 2022).

O TDD comumente é usado para tarefas difíceis, nas quais o programador não sabe por onde começar sua implementação. Usando uma analogia, imagine que é preciso puxar um balde com água de um poço; se o poço não for muito fundo, será fácil puxá-lo com uma manivela simples, porém se o poço for muito fundo, será mais difícil puxar o balde por conta do cansaço físico. Logo, será necessária uma manivela com catraca para evitar que o balde desça enquanto a pessoa o puxa. O TDD, nesse cenário, é como se fossem os dentes dessa catraca, cada teste criado com TDD dá a segurança de o programador estar mais próximo da funcionalidade real. Assim que termina de criar todos os testes, a funcionalidade estará pronta, e, nesse caso, o programador terá uma catraca com dentes (BECK, 2010).

2.2 A evolução do TDD

O TDD não é tão atual assim, aliás, seu surgimento se deu apenas por volta dos anos 1990, sendo ainda muito usado no nosso cotidiano. Surgiu com outras metodologias que fazem parte do manifesto ágil. O objetivo do TDD, na época, era de criar códigos que podiam ser checados e validados de maneira simples (GRUPO MULTI, 2019).

Ao longo dos anos, o TDD se tornou um dos campos de estudo mais essenciais no ramo da programação (MARTIN, 2009).

Por volta dos anos 2000, surgiu uma outra metodologia, chamada de BDD (Desenvolvimento Orientado por Comportamento), que foi apresentada por Dan North. Essa metodologia surgiu como uma evolução da metodologia TDD. Basicamente, o BDD tem como objetivo aproximar as pessoas não técnicas das funcionalidades técnicas do sistema, de modo que qualquer pessoa possa entender isso. Isto é, o processo é feito por meio de cenários de testes chamados de *features*, que são montados antes de o teste ser escrito. Esses cenários dizem o que a funcionalidade precisa ter para ser executada, o que ela fará em seguida e qual é o resultado esperado (CARVALHO, 2019).

As *features* são regidas por três palavras (**dado**, **quando**, **então**). Não é uma obrigação usar esse modelo, mas Dan North recomenda o uso de um padrão comum (CARVALHO, 2019). A seguir, apresentamos um exemplo de uma *feature* de cadastro de usuários.

Dado que o administrador do sistema precisa cadastrar um novo funcionário, **quando** ele informar o nome e a data de nascimento deste, **então** o funcionário deve ser salvo no banco de dados (CARVALHO, 2019).

Cada *feature* do BDD pode ser traduzida em uma classe de teste, e cada passo descrito na *feature* pode ser traduzido como uma ação no teste (CARVALHO, 2019). Desse modo, pode-se otimizar a aplicabilidade dessa metodologia com a metodologia TDD.

2.3 Contribuição do TDD no Manifesto Ágil

O TDD também é uma prática ágil, pois oferece um valor muito grande para o projeto, oferece um *software* funcional acima de documentação abrangente. Além disso, os pioneiros do TDD, Kent Beck, Martin Fowler e Ron Jeffries estavam entre as pessoas que escreveram o manifesto ágil (DIGITE, 2022).

2.4 Ciclos do TDD

A metodologia TDD requer passar por 3 ciclos repetidas vezes (DIGITE, 2022).
São eles:

- **Vermelho**

Nesta etapa é escrito um teste com o cenário necessário para testar a funcionalidade, porém é esperado que o teste não funcione, ou nem mesmo compile (BECK, 2010).

- **Verde**

Nesta etapa são criados os recursos, fazendo de tudo para que o teste funcione rapidamente (BECK, 2010).

- **Refatorar**

Nesta etapa deve-se consertar as duplicações de código e objetos *fakes* criados para que o teste funcione rapidamente (BECK, 2010).

Somente esses ciclos não bastam; para escrever código usando a metodologia TDD, é preciso também mesclar esses ciclos com outras estratégias, que Robert Cecil Martin definiu como leis do TDD em seu livro *Código Limpo* (DIGITE, 2022).

Lei nº 1

Não se deve escrever o código de produção antes de escrever um teste unitário com falhas (MARTIN, 2009).

Lei nº 2

Não se deve escrever mais testes unitários do que um falhando (MARTIN, 2009).

Lei nº 3

Não se deve escrever mais códigos de produção do que o necessário para que o teste falhando funcione (MARTIN, 2009).

O objetivo dessas leis é manter os testes concentrados em cada fase para evitar que o programador fique preso nos ciclos do TDD (DIGITE, 2022).

Seguindo essas leis, o desenvolvimento dirigido por testes se torna mais produtivo, pois o código de teste é escrito apenas poucos segundos antes do código de produção. Desse modo, podemos ter dezenas de testes criados diariamente e centenas a cada mês (MARTIN, 2009).

2.5 Problemas que testes sujos trazem para o *software*

Antigamente, os testes não eram tão valorizados quanto hoje em dia: os desenvolvedores não ligavam para a qualidade nem para sua manutenção; os testes tinham problemas como variáveis mal definidas, as funções de testes não eram curtas e descritivas, o teste não era algo bem pensado, se a barrinha verde de teste funcionasse, era isso que importava (MARTIN, 2009).

Testes mal escritos, como os descritos no parágrafo anterior, são o mesmo que teste nenhum. O grande problema desses testes é a manutenção. Não há a menor dúvida de que os testes, ao longo do tempo, vão mudar de acordo com a evolução do próprio código de produção. O problema é que conforme o código de produção evoluía, os testes também precisavam ser alterados, e o que funcionava antes passou a não funcionar mais, e o programador tinha que voltar a mexer no teste para que ele voltasse a funcionar. Por isso, perdia-se mais tempo tentando fazer os testes funcionar do que escrevendo testes novos (MARTIN, 2009).

Quanto pior e mais confuso o teste era, mais difícil era mudá-lo, e quando os gerentes de projeto questionavam a demora para a entrega da funcionalidade, os programadores culpavam os testes. Portanto, os testes eram vistos como um problema em constante crescimento (MARTIN, 2009).

2.6 Auxílio do TDD na escrita de código limpo

Tal problema relatado na seção anterior seria facilmente resolvido se tivesse sido utilizada uma metodologia capaz de proporcionar testes e códigos limpos, de fácil compreensão. A metodologia TDD é uma delas, sendo que o foco do TDD é a escrita de código limpo e que funcionam (BECK, 2010). O TDD dá ao programador a chance de melhorar o código a cada modificação. Se o programador apenas escrever o código de produção do início, nunca terá tempo para melhorá-lo. Além disso, um dos passos do TDD é a própria refatoração (BECK, 2010).

2.7 Princípio F.I.R.S.T.

Testes limpos seguem cinco regras que formam a sigla em inglês F.I.R.S.T., (*Fast, Independent, Repeatable, Self-validating, Timely*) (MARTIN, 2009).
A seguir, cada item será detalhado.

- **Fast**:

Rápido. Os testes devem ser rápidos, pois assim o programador não terá problemas em executá-los com frequência. Se eles forem lentos, o programador ficará relutante em executar a suíte de testes e, assim, não encontrará com rapidez os *bugs* para consertá-los com celeridade (MARTIN, 2009).

- **Independent**:

Independente. Os testes devem ser independentes uns dos outros e executáveis em qualquer ordem. Caso um teste dependa do outro, será mais difícil para o programador identificar o que ocasionou a falha e, assim, ele levará mais tempo para corrigi-la. Além disso, se existir dependência entre testes, caso um deles falhe, então isso ocasionará um efeito dominó, e todos os testes falharão (MARTIN, 2009).

- **Repeatable:**

Repetitividade. Os testes devem ser repetitivos, de forma que se possa executá-los em diferentes ambientes, como o ambiente de produção e QA. Além disso, os testes não podem depender de fatores externos, como indisponibilidade de rede, pois isso tornará sua correção mais difícil, e o programador sempre terá uma desculpa se os testes falharem (MARTIN, 2009).

- **Self-validating**

Autovalidação. Os testes devem ser autovalidáveis, de forma que o programador não deve ter que fazer comparações manualmente para saber se o teste falhou ou não, pois, desse modo, pode ser muito difícil executar todos os testes (MARTIN, 2009).

- **Timely**

Pontualidade. Os testes precisam ser escritos imediatamente antes do código de produção. Quando o programador deixa os testes por último, eles tendem a ser mais difíceis de criar, pois o programador pode ter criado o código de produção de uma maneira que não pode ser testado ou, ainda, o programador pode achar que já terminou a implementação e que não precisa de testes (MARTIN, 2009). Além disso, os testes podem ficar sujos e mal escritos e, como visto anteriormente, a escrita dos testes antes do código de produção auxilia muito na clareza do código.

O padrão FIRST é muito utilizado na escrita de testes, e seus objetivos se assemelham muito aos que o TDD oferece, principalmente o último, que diz que os testes precisam ser pontuais e escritos antes do código de produção. Portanto, esse padrão é usado com a metodologia TDD.

3 PROCEDIMENTOS METODOLÓGICOS

Para o desenvolvimento do presente trabalho, a metodologia utilizada foi a pesquisa bibliográfica em livros, artigos e *sites* especializados em Tecnologia da Informação (TI) e em desenvolvimento de *softwares*. Inicialmente, foram realizadas buscas para identificar os materiais mais relevantes sobre testes e as melhores práticas para a implementação desses testes. Em seguida, visou-se demonstrar as fases e ciclos para um desenvolvimento mais produtivo utilizando a metodologia TDD e os seus benefícios na escrita de código limpo. Posteriormente,

foram demonstrados os resultados que essa metodologia ofereceu na escrita de uma funcionalidade nova no sistema. Por fim, foi redigida a conclusão, reforçando os benefícios dessa técnica na identificação e correção de *bugs*.

4 RESULTADOS E DISCUSSÃO

A seguir, será demonstrado um exemplo prático de como é criada uma funcionalidade no sistema utilizando a metodologia TDD, por meio da qual serão notados os benefícios citados neste artigo.

Para esse exemplo, será criada uma funcionalidade de cadastro de usuário. O sistema é um jogo *on-line* de RPG que precisa ter usuário cadastrado para ser jogado.

A linguagem de desenvolvimento usada para essa demonstração é a Kotlin, com o *framework* de testes JUnit 5.

4.1 *Feature* da funcionalidade

Em virtude de o usuário precisar ter um cadastro no sistema para jogar, quando informar seu nome e data de nascimento, então o sistema deverá cadastrá-lo.

Quadro 1 – Classe de teste da funcionalidade

```
import dto.PlayerDTO
import model.Player
import org.junit.jupiter.api.BeforeEach
import org.junit.jupiter.api.Test
import repository.CreatePlayerRepository
import service.CreatePlayerService
import service.CreatePlayerServiceImpl
import kotlin.test.assertNotNull

class MockCreatePlayerRepository : CreatePlayerRepository {
    override fun save(player: PlayerDTO): Player {
        return Player(1, player.name, player.birthDate)
    }
}

class CreatePlayerServiceTest {

    lateinit var createPlayerService: CreatePlayerService

    @BeforeEach
    fun setup() {
        createPlayerService = CreatePlayerServiceImpl(MockCreatePlayerRepository())
    }

    @Test
    fun createPlayer() {
        val player: PlayerDTO = PlayerDTO("Jogador 1", "2010-01-01")
        val createdPlayer: Player = createPlayerService.create(player)
        assertNotNull(createdPlayer.id);
    }
}
```

Fonte: De autoria própria.

O Quadro 1 mostra uma classe que testa quando um usuário é cadastrado no sistema com sucesso.

Ignorando a maior parte dessa classe e focando o mais importante, que é o teste (método anotado, com @Test em cima), esse teste tem apenas 3 linhas. A primeira linha trata da criação do objeto que simula um jogador, cujo nome é jogador 1; a segunda linha trata do método de salvamento do jogador no sistema – para isso, utilizamos o método *create* da classe responsável por salvar o jogador, que, nesse caso, é a classe CreatePlayerServiceImpl; e a terceira e última linha valida a criação do jogador com sucesso se ele tiver um identificador válido.

A classe de teste mostrada anteriormente, sozinha, não funciona, pois as demais classes ainda não foram criadas. Por isso, esse teste está no ciclo vermelho do TDD, no qual deve ser criado um teste com falhas. Entretanto, ele já diz muita coisa. Só de olhar para ele, o programador já entende o que esperar dessa funcionalidade e o que é preciso para executá-la, ele já sabe que para criar um usuário, é preciso informar seu nome e data de nascimento e já sabe como devem se chamar a classe e o método. Com isso, já se poupou o tempo que o programador levaria para procurar em outros arquivos do projeto uma documentação de como a funcionalidade funciona, isto é, como se fosse uma documentação para ele; e quem, posteriormente, der manutenção nesse código, seguirá o restante das classes.

Quadro 2 – Classe PlayerDTO

```
package dto
class PlayerDTO (val name: String, val birthDate: String) {
}
```

Fonte: De autoria própria.

O Quadro 2 mostra a classe responsável por armazenar o nome e a data de nascimento que o usuário informou.

Quadro 3 – Classe Player

```
package model
class Player (val id: Int, val name: String, val birthDate: String) {
}
```

Fonte: De autoria própria.

O Quadro 3 mostra a classe responsável por armazenar o nome, a data de nascimento e o ID do usuário já cadastrado.

Quadro 4 – Classe CreatePlayerService

```
package service

import dto.PlayerDTO
import model.Player

interface CreatePlayerService {
    fun create(player: PlayerDTO): Player
}
```

Fonte: De autoria própria.

O Quadro 4 mostra uma interface com o método responsável por cadastrar o usuário, sendo que esse método, mais tarde, será implementado pela classe responsável pela lógica de cadastro do usuário no sistema.

Quadro 5 – Classe CreatePlayerServiceImpl

```
package service

import dto.PlayerDTO
import model.Player
import repository.CreatePlayerRepository

class CreatePlayerServiceImpl(
    private val repository: CreatePlayerRepository
) : CreatePlayerService {

    override fun create(player: PlayerDTO): Player {
        val savePlayer: Player = repository.save(player)
        return Player(savePlayer.id, savePlayer.name, savePlayer.birthDate)
    }
}
```

Fonte: De autoria própria.

O Quadro 5 mostra a classe com a lógica responsável por salvar o usuário no sistema.

Quadro 6 – Classe CreatePlayerRepository

```
package repository

import dto.PlayerDTO
import model.Player

interface CreatePlayerRepository {
   fun save(player: PlayerDTO): Player
}
```

Fonte: De autoria própria.

Por último, o Quadro 6 mostra uma interface com o método responsável por salvar o usuário no banco de dados.

Depois de incluir todas as classes e interfaces, o teste funciona. Essa foi a fase verde do TDD, restando apenas a fase de refatoração. Porém, como esse código é bem simples, não existem muitas refatorações, pois como essa funcionalidade foi escrita com a metodologia TDD, isso também garantiu que foi implementada da melhor forma possível.

Agora que temos o teste para criar um usuário no sistema, caso algum dia um programador mexa na classe responsável pela lógica de criação do usuário, o teste é rodado. Se ele parar de funcionar, então ficará claro que algo foi inserido pelo programador. Desse modo, a chance de adicionar *bugs* no código é bem reduzida.

A utilização da metodologia TDD traz muitos benefícios, como vimos neste artigo e nesse exemplo, mas ela também tem suas desvantagens, por exemplo, o tempo perdido na escrita de testes, pois muitos desenvolvedores relatam que, ao usá-la, torna-se mais demorado criar funcionalidades novas. Porém, as vantagens superam as desvantagens, pois, como visto, grande parte do tempo de trabalho do programador é gasta procurando *bugs*. Com o uso da metodologia TDD e uma esteira grande de testes, esse tempo investido na criação de testes compensa o tempo que seria perdido procurando *bugs*.

5 CONCLUSÃO

O objetivo deste trabalho foi apresentar para o leitor uma metodologia de desenvolvimento de *softwares* capaz de criar uma rotina de testes que não só é benéfica, mas também essencial para o *software*, pois ajuda na identificação e correção de *bugs* e, sem ela, o

programador perderia várias horas depurando códigos atrás de *bugs*, o que certamente causaria ainda mais demora na entrega das atividades e funcionalidades novas do sistema.

A utilização da metodologia TDD na escrita de testes não é a única existente, pois, como visto neste trabalho, seu uso influencia a forma como o código é implementado, propondo testes e códigos limpos. Tais benefícios justificaram seu uso, pois um teste malcuidado prejudica a entrega e qualidade do *software.*

Ressaltamos que essa metodologia não é só usada para testes, também é usada para gerar documentação e segurança para quem a usa.

Concluímos que o desenvolvimento dirigido a testes tende a crescer e a evoluir ainda mais no desenvolvimento de *softwares* por trazer grandes benefícios na identificação e correção de *bugs*, documentação, códigos limpos e segurança nos *softwares*.

REFERÊNCIAS

BECK, Kent. **TDD: Desenvolvimento Guiado por Testes.** [S. 1.], Bookman, 2010.

CARVALHO, André. **BDD: Desenvolvimento orientado a comportamento.** 2019 Disponível em: https://inside.contabilizei.com.br/bdd-desenvolvimento-orientado-a-comportamento-62e71f2eabe9. Acesso em: 12 set. 2022.

FOWLER, Martin. **Refatoração: Aperfeiçoando o Design de Códigos Existentes.** [*S. l.*] Novatec, 2020.

GPIRESS. **Quando usar TDD**, 2020. Disponível em: < https://dev.to/gpiress/quando-usar-tdd-1c18>. Acesso em: 29 set. 2021.

MARTIN, Robert C. **Código Limpo: Habilidades Práticas do Agile Software.** [*S. l.*]: Alta Books, 2009.

O QUE É TEST DRIVEN DEVELOPMENT (TDD). **Digite**, 2022. Disponível em: https://www.digite.com/pt-br/agile/desenvolvimento-orientado-a-testes-tdd/. Acesso em: 05 set. 2022.

TDD: Fundamentos do Desenvolvimento Orientado a Testes. **Grupo Multi**, 2019. Disponível em: https://www.grupomult.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/ . Acesso em: 07 set. 2022.