A UTILIZAÇÃO DE TESTES AUTOMATIZADOS NO DESENVOLVIMENTO DE *SOFTWARE*

Cleber Spirlandeli[1]

Carlos Eduardo de França Roland[2]

Resumo

Desenvolver *software* é uma atividade que exige concentração e atenção a detalhes, e muitas vezes, dada a complexidade das regras a serem implementadas no código, falhas de lógica passam despercebidas. Com o objetivo de tornar o exercício da programação mais fácil e seguro, foram criadas ferramentas de automação de testes de *software* que são o objeto desta pesquisa. Buscou-se, através de pesquisa bibliográfica exploratória e estudo de caso, aprofundar o conhecimento sobre os conceitos de Desenvolvimento Guiado por Testes, e das características e utilização de uma das ferramentas disponíveis no mercado para implementar testes automatizados. Com o objetivo de se aplicar os métodos de testes, adotou-se a implementação em um caso real. Foram utilizadas as principais referências mundiais sobre os tópicos relacionados ao tema, bem como estudado o *framework* de testes Jest.JS. A partir do entendimento de como se aplicar a ferramenta buscou-se implementar rotinas de testes para se verificar e entender os resultados oferecidos pelos processos. Considera-se que a pesquisa alcançou seus objetivos como apresentado e discutido neste artigo, e que ampliou a visão dos autores sobre o uso de TDD no desenvolvimento de *software*.

Palavras-chave: Desenvolvimento Guiado por Testes. JavaScript. Jest.JS. TDD. Testes automatizados.

Abstract

Developing software is an activity that requires concentration and attention to details, and often, given the complexity of the rules to be implemented in the code, logic flaws might be unnoticed. In order to make programming easier and safer, software testing automation tools have been created that are the subject of this research. Through exploratory bibliographic research and case study, we sought to deepen the knowledge about the concepts of Test Driven Development (TDD), and the characteristics and use of one of the tools available in the market to implement automated tests. In order to apply the tests methods, the implementation was adopted in a real case. The main world references on topics related to the theme were used, as well as the Jest.JS testing framework. From understanding how to apply the tool, we tried to implement test routines to verify and understand the results offered by the processes. It is considered that the research achieved its objectives as presented and discussed in this article, and broadened the authors' view on the use of TDD in software development.

---

[1] Graduando em Análise e Desenvolvimento de Sistemas pela Fatec Dr Thomaz Novelino – Franca/SP. Endereço eletrônico: contato.spirlandeli@gmail.com
[2] Docente no curso de Análise e Desenvolvimento de Sistemas da Fatec Dr Thomaz Novelino – Franca/SP. Endereço eletrônico: carlos.roland@fatec.sp.gov.br

Keywords: *Automated tests. JavaScript. Jest.JS. Test Driven Development. TDD.*

1 Introdução

Desenvolver *software* é uma atividade fundamentalmente de raciocínio lógico para instruir uma máquina a executar processos de cálculos numéricos. É uma tarefa que exige concentração e atenção a detalhes, e muitas vezes, dada a complexidade das regras a serem implementadas no código, falhas de lógica passam despercebidas. Com o objetivo de tornar o exercício da programação mais fácil e seguro, foram criadas ferramentas de automação de testes de *software* que são o objeto desta pesquisa.

Como delimitação do escopo deste estudo, foram definidos que a plataforma de desenvolvimento usada como base seria JavaScript e as pesquisas foram direcionadas para ferramentas de testes que suportassem este ambiente. Dos *frameworks* disponíveis no mercado, optou-se por explorar o Jest.JS, e como referência para aplicação das técnicas e métodos de testes automatizados optou-se por implementá-las em duas funcionalidades de um sistema real utilizado pela empresa que um dos autores trabalha.

Como metodologia da pesquisa, foram adotadas pesquisa bibliográfica exploratória para se aprofundar o conhecimento sobre os conceitos de Desenvolvimento Guiado por Testes, e estudo de caso para uso de um *framework* de testes automatizados em um caso real.

Foram utilizadas as principais referências mundiais sobre os tópicos relacionados ao tema e a partir do entendimento de como se aplicar Jest.JS, foram usadas duas funcionalidades de um sistema real para implementação de rotinas de testes para execução e obtenção de *feedback* dos resultados.

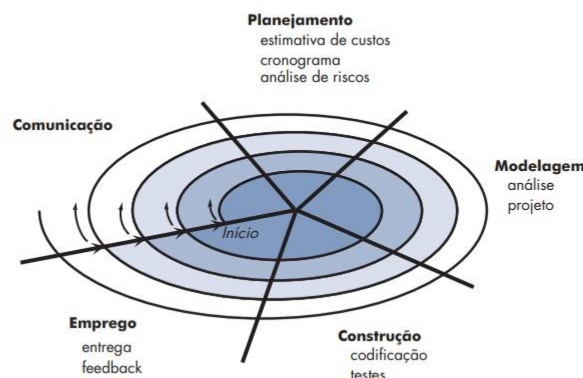
Este artigo é estruturado nesta Introdução, seguida da seção que apresenta o embasamento teórico sobre os processos do ciclo de vida de desenvolvimento de *software*, de testes automatizados e desenvolvimento guiado por testes, e as vantagens e desvantagens de seu uso, para então apresentar as características de implementação de TDD, seguido da Seção 4 que descreve e comenta a aplicação de Jest.JS no caso real de testes em duas funcionalidades do sistema escolhido. Por fim são tecidas considerações sobre a execução do projeto e relacionadas as referências bibliográficas utilizadas na pesquisa.

2 Embasamento teórico

Para se desenvolver *software* é necessário cumprir etapas desde sua concepção até que o projeto seja concluído. Áreas do conhecimento da Ciência da Computação, como a Engenharia de Software, propõem métodos e padrões para se alcançar melhores resultados de esforços de programação.

Pressman (2011) afirma que para melhores resultados no desenvolvimento de *software*, é necessário se cumprir etapas em ciclos crescentes de desenvolvimento, como em uma espiral (Figura 1), de comunicação, planejamento, modelagem, construção e entrega.

Figura 1 – Ciclo de vida de um projeto



Fonte: Pressman (2011, p. 65).

Este artigo tem o propósito de destacar, do ciclo de vida de um projeto de *software*, a etapa de Construção mais especificamente nas atividades de Testes.

2.1 Testes Automatizados de *Software*

Teste automatizado é realizado por um programa que testa outro programa para localizar eventuais erros. Ao criar um programa somente para testes, é possível executar os testes constantemente, várias vezes, de forma rápida e com baixo custo. Uma das formas de se desenvolver *software* dirigido por testes é com a implementação de trechos de código, chamados de Testes Unitários ou Testes Funcionais, nos quais se testa a menor parte passível de verificação de um programa. Por exemplo se a programação for orientada a objetos testam-se os métodos criados nas classes de objetos. Com a utilização de testes unitários as mensagens de retorno (*feedback*) indicam se algo parou de funcionar a partir de alterações realizadas numa funcionalidade, gerando algum erro ocorrido em algum trecho do sistema.

 ISSN: 2595-5942

Portanto, a execução regular dos testes permite que se descubram imediatamente os erros e suas origens durante a implementação do código, reduzindo os custos de correção das falhas quando forem detectadas em usos futuros (ANICHE, 2014).

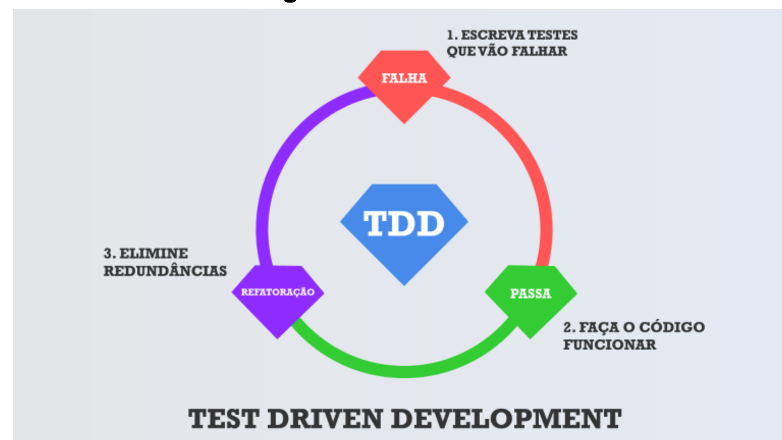
2.2 Desenvolvimento Guiado por Testes

Observando as tendências do mercado, diversas metodologias ágeis como XP recomendam utilizar as boas práticas do Test-Driven Development (TDD) (em tradução livre: Desenvolvimento Guiado por Testes) para o desenvolvimento de *software*, sugerindo que o desenvolvedor escreva testes automatizados antes de implementar a funcionalidade, constantemente e ao longo de todo o desenvolvimento (ANICHE, 2012).

Percebe-se que conforme o projeto aumenta, ganhando novas funcionalidades, o teste automatizado também se propaga, cobrindo maior área do sistema, garantindo que todo o código funcione como definido após cada alteração (MYKLEBUST *et al.*, 2010).

O processo de desenvolvimento de testes, assim como o ciclo de vida de desenvolvimento de *software*, é composto por um conjunto de atividades que regem o método. Inicia-se com a escrita dos testes para que falhem, em seguida escreve-se o código de tal forma a passar pelo teste, e então se revê o código, eliminando redundâncias num processo chamado de refatoração de código. Na Figura 2, pode-se observar as etapas do ciclo de desenvolvimento utilizando TDD.

Figura 2 – Ciclo do TDD



Fonte: 4Linux (2017)

ISSN: 2595-5942

Os autores dos referenciais citados neste documento argumentam que o uso correto do TDD pode ajudar a obter produtividade e produzir *software* de qualidade superior.

Dentre os motivos relacionados aos benefícios, Gomes (2013, p. 72) declara que:

> O uso de TDD assegura que todo código adicionado ao repositório seja testado e, além disso, dá ênfase em se escrever apenas o código necessário para que os testes passem, evitando que o desenvolvedor escreva mais do que é necessário, contribuindo para um código mais simples, enxuto e também mais fácil de dar manutenção.

Segundo Aniche (2014), o desenvolvimento guiado por testes proporciona benefícios tais como: foco no teste e não na implementação, garante que toda funcionalidade será testada, simplicidade, melhoria na elaboração da classe e código limpo, como detalhado a seguir.

- Foco no teste e não na implementação: iniciando o código pelo teste, o desenvolvedor constrói uma funcionalidade sem redundância, objetiva e clara, de acordo com a regra de negócio.
- O código nasce testado: se o desenvolvedor pratica o ciclo corretamente, garante que, para cada funcionalidade, antes seja criado um teste e então todo código escrito possui ao menos um teste.
- Simplicidade**:** o desenvolvedor cria testes validando apenas o problema apresentado, com isso, constrói a classe objetiva, eliminando trechos desnecessários e torna o código mais legível.
- Melhor reflexão sobre o *design* da classe: “[...] quando o desenvolvedor consegue observar com atenção o código do teste, seu *design* de classes pode crescer muito em qualidade” (ANICHE, 2014, p. 24). Na criação da classe, o desenvolvedor é induzido a analisar melhor as regras de negócios, abstrair o problema mais profundamente e também elaborar a solução de maneira simples.
- Maior qualidade do código: segundo Fowler (2019), desenvolver *software* de qualidade necessita de revisões regulares e refatoração. À medida que os desenvolvedores adquirem maior conhecimento e domínio das regras de negócios, o desenvolvimento guiado por testes proporciona realizar aprimoramentos de código por meio da refatoração, mas garantindo que as funcionalidades continuem funcionando corretamente.

ISSN: 2595-5942

Em vista disso, seguindo o ciclo do TDD corretamente, o *software* obtém o benefício de maior qualidade pois o desenvolvedor realiza inúmeras refatorações nos códigos dos testes e do sistema.

Conforme o TDD é utilizado de forma correta, pode trazer outros benefícios para o projeto. Dentre eles, destacam-se: reduzir custo de manutenção, melhorar a compreensão dos requisitos, obter *feedback* constante, e aumentar a produtividade. Em relação à redução de custo de manutenção, de acordo com Devmedia (2013), muitas empresas se desinteressam por implementar testes automatizados, e nestes casos os erros são encontrados tardiamente elevando os custos do produto. Diante disso, o TDD relacionado às metodologias ágeis pode reduzir os custos de desenvolvimento e manutenção. Destaca-se ainda que, além de reduzir custos, também garante maior confiabilidade e qualidade do *software*. Sob o aspecto do *feedback* constante, Aniche (2012) comenta que programadores que usam TDD corretamente escrevem um pequeno teste e um pequeno código e, em seguida obtêm *feedbacks* imediatos diversas vezes até que a funcionalidade seja concluída. Logo, ao receber *feedback* constantemente, o programador pode corrigir problemas e fazer melhorias no código, reduzindo assim o custo de manutenção, pois é mais fácil corrigir um problema que acabou de escrever do que em alguns dias.

Do mesmo modo, em relação ao *feedback*, Gomes (2013, p. 126) declara que:

> Desenvolvedores precisam receber feedback sobre as alterações que fizeram no software, precisam receber feedback a cada alteração para que saibam se os testes de regressão continuam passando, por isso utilizam integração contínua e TDD. Além disso, precisam saber se o código que escrevem está legível do ponto de vista de seus pares.

Além dos benefícios apresentados outros estudos, como de George e Williams (2003) e de Janzen (2006), analisaram a efetiva melhoria em projetos de *software* utilizando TDD. Os estudos mostram que o código de programadores que usam TDD passa em 50% mais testes de caixa-preta do que códigos sem a prática; reduz a complexidade do algoritmo e o tempo gasto para encontrar erros; há redução de 40 a 50% de defeitos, e um impacto mínimo na produtividade.

Como comentado por Aniche (2014, p. 142) estudos realizados demonstram que:

> [...] 87.5% dos programadores acreditam que TDD facilitou o entendimento dos requisitos e 95.8% acreditam que TDD reduziu o tempo gasto com debug. 78% também acreditam que TDD aumentou a produtividade da equipe. Entretanto, apenas 50% acreditam que TDD ajuda a diminuir o tempo de desenvolvimento. Sobre qualidade, 92% acreditam que TDD ajuda a manter

um código de maior qualidade e 79% acreditam que ele promove um design mais simples.

Percebe-se que, em estudo de caso realizado em produtos na Microsoft e IBM, houve redução de 40% a 90% no número de defeitos. Além disso, foram notados aumento considerável na qualidade do código e maior facilidade na resolução de futuras manutenções, mas também se constatou aumento no tempo inicial do projeto de 15% a 35% (NAGAPPAN, 2008).

Complementando os benefícios apresentados, por outro lado também são consideradas algumas desvantagens no uso do método TDD para desenvolvimento do projeto.

O criador do *framework* Ruby on Rails, David Heinemeier Hansson, é um dos principais apoiadores do TDD como o melhor método de desenvolvimento. Entretanto em abril/2014, ele fez uma declaração importante e inesperada pela comunidade, apresentando dessa vez uma visão contraria e declarando que discordava do uso de TDD (HANSSON, 2014). O fórum iMasters, referência para a comunidade de tecnologia com foco no compartilhamento de ideias e tendências na área, traduziu a declaração de Hansson e destacou alguns pontos importantes do por que não seria vantajoso utilizar o TDD. De acordo com iMasters (2014, *online*), Hansson declara que:

- Escrever testes com antecedência, antes do código, faz com que demore maior tempo para finalizar o projeto, atrasando seu lançamento pois o tempo gasto para gerar testes automatizados poderia ser utilizado para criar outras funcionalidades do sistema.
- Os projetos vão mudar e os testes antigos serão ultrapassados. Quando o objetivo de uma funcionalidade é alterado, o teste antigo passa a ser inutilizado pois ao menos uma parte do código não será mais utilizada.
- Os testes não são funcionalidades do sistema, são usados para evitar possíveis erros no futuro sobre o funcionamento adequado dos recursos. O desenvolvimento de testes extensivos leva mais tempo e se torna uma tarefa desagradável e, com o tempo, os desenvolvedores desistem de desenvolver testes.
- O TDD não precisa ser utilizado em todas as funcionalidades e funções, deve ser utilizado de maneira inteligente.

ISSN: 2595-5942

Segundo iMasters (2014), Hansson declara que se percebe “que o desperdício de tempo foi muito maior do que se você tivesse escrito apenas o código que fez o que deveria ter feito sem os testes”.

Para Aniche (2014), nenhuma prática deve ser usada ou descartada integralmente. Todo o contexto precisa ser analisado e não necessariamente em todos os casos precisa ser praticado o TDD. É aceitável não ser usado em alguns pontos do projeto, mas não a completa eliminação de testes. Entende-se que o projeto não precisa ter 100% de cobertura de testes, mas sugere-se que a grande maioria do código tenha.

Eventualmente, trechos de código simples e de fácil entendimento não precisam ser testados, mas a equipe de desenvolvimento deve sempre desejar que todo o código seja testado, visto que, à medida que o programador adquire experiência usando a prática do TDD, seus testes são mais significativos, pois ele sabiamente decidirá se o trecho de código que está desenvolvendo precisa ser testado.

3 Ferramentas e métodos de desenvolvimento

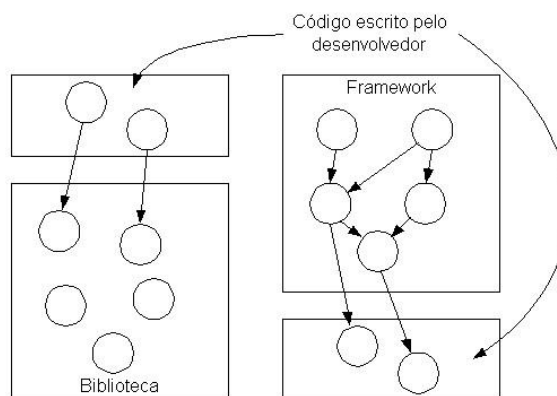
Nesta seção são apresentados os métodos e as ferramentas de desenvolvimento estudados na pesquisa sobre uso de TDD na implementação de *software*.

3.1 *Frameworks*

Frameworks auxiliam no desenvolvimento rápido e seguro de *software*. Assim, *frameworks* têm como objetivo a reusabilidade na resolução de problemas recorrentes, permitindo que o desenvolvedor foque em solucionar o problema proposto sem reescrever *software*. No contexto da reusabilidade são utilizadas bibliotecas e o *framework* gerencia o acesso ao conjunto delas tornando sua utilização mais simples (TABLELESS, 2019).

Na figura 3, pode-se observar o que é um f*ramework* e uma Biblioteca.

Figura 3 – *Framework* e Biblioteca



Fonte: Sauvé (2019)

3.2 *Back-end* e *Front-end*

Em aplicações *web*, o programador *back-end* desenvolve *software* que roda no servidor, implementando a lógica de programação responsável pelas regras de negócio e não tem contato direto com o desenvolvimento da interface visual do sistema, chamado de *front-end* (TREINAWEB, 2017).

Existem diversas tecnologias de programação que atuam no *back-end.* Dentre as tecnologias existentes uma delas é o Node.JS, que é uma plataforma de desenvolvimento que permite o uso de JavaScript no lado do servidor. Ou seja, uma tecnologia baseada no JavaScript V8 do Google que é implementada no navegador Chrome, facilitando a criação de aplicações rápidas e escaláveis (NODEBR, 2016).

3.3 *Frameworks* para Testes Automatizados

São disponibilizados no mercado de desenvolvimento de *software* uma gama de ferramentas para implementação de testes automatizados. Como exemplo de *frameworks* de testes para programação JavaScript encontram-se Mocha, Mochify, Sinon.JS e Jest.JS, com algumas pequenas diferenças entre eles.

O *framework* Jest.JS é um projeto desenvolvido pelo Facebook que recebe atualizações contínuas e possui excelente documentação. É o mais utilizado no mercado tendo, entre os dias 30/10/2019 e 05/11/2019, atingido a marca de 5.025.404 *downloads* semanais segundo NPM (2019), sendo nesta data usado em quase 1.500.000 de repositórios públicos no Github (GITHUB, 2019). Em vista disso grandes empresas como Facebook, Twitter, Jornal New York Times, Spotify, AirBnB e Instagram utilizam o Jest.JS para testes automatizados (JESTJS, 2019).

4 Resultados e Discussão

Neste estudo foram implementados testes automatizados utilizando o *framework* Jest.JS bem como a aplicação de boas práticas de programação que são apresentadas nesta seção do artigo.

Usou-se como base para estudo de caso o módulo de gerenciamento de usuários e de acesso ao sistema da empresa onde um dos autores trabalha.

Na funcionalidade Usuários é possível se criar um novo registro, listar os dados de cadastros existentes e excluir um usuário cadastrado. Para gerenciar Acessos ao sistema, a função valida as credenciais do Usuário (identificação e senha), gera uma chave criptografada e valida uma chave de acesso já existente. Neste contexto operacional foram implementados os testes automatizados que são apresentados e comentados.

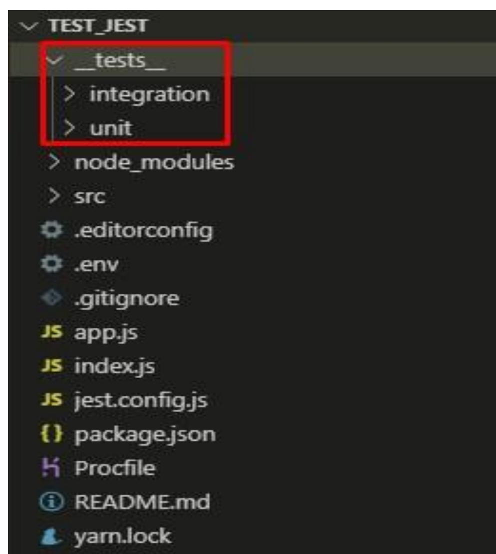
Para organizar a inserção de testes automatizados em JavaScript é necessário se criar uma pasta para separar os trechos de testes dos de desenvolvimento da lógica do sistema. Para isso, recomenda-se criar na raiz do projeto, uma pasta chamada __tests__. Os dois sublinhados antes e depois do termo *tests* servem para diferenciar a pasta com rotinas de testes das pastas das rotinas do sistema e deixá-la sempre em destaque na primeira posição da hierarquia de pastas do projeto.

Em seguida deve-se criar, nesta pasta de testes, outras duas para separar códigos de funcionalidades do sistema de códigos unitários, nomeadas integration e unit como mostrado na Figura 4.

Para validar todas as lógicas que abrangem a funcionalidade do usuário, é necessário, dentro da pasta integration, se criar o arquivo users.test.js para inserção das descrições dos parâmetros de teste como apresentado na Figura 5. Os métodos utilizados para a criação das rotinas de testes são comentados após a apresentação das figuras.

ISSN: 2595-5942

Figura 4 – Hierarquia de pastas



Fonte: o autor

Repare que, para os testes na funcionalidade Usuários, foram instanciados os objetos request e app, e utilizados os métodos describe e it, com os seguintes objetivos:

- request: herda atributos e métodos da biblioteca supertest e tem como objetivo realizar chamadas para executar processos para receber informações do sistema (.get), adicionar (.post), alterar (.put) e apagar (.delete).
- app: herda a classe de início que geralmente contém as principais configurações de como a aplicação deverá funcionar.
- describe: uma breve descrição sobre qual tópico o trecho de teste opera.
- it: recebe uma descrição curta, semelhante ao describe e, é responsável por conter a implementação do teste de alguma funcionalidade do sistema. Recomenda-se como boa prática que para se criar as descrições dos testes, o termo it deve ser entendido como parte da frase como por exemplo *it should create a new user*.

 ISSN: 2595-5942

Figura 5 – Testes para a funcionalidade Usuários

```
1     // IMPORTAR BIBLIOTECAS E/OU CLASSES EXTERNAS
2     const request = require('supertest');
3     const app = require('../../app');
4
5     // BREVE DESCRIÇÃO DA FUNCIONALIDADE A TESTAR
6     describe('1.0 - Should tests functionalities with the users', () => {
7         // CRIAR VÁRIAVEL PARA ARMAZENAR O ID DA RESPOSTA
8         let _id = null;
9
10        // BREVE DESCRIÇÃO DO TESTE DE FUNCIONALIDADE E EXECUÇÃO DO PRIMEIRO TESTE
11 >      it('1.1 - Should create a new user', async () => {···
34        });
35        // BREVE DESCRIÇÃO DO TESTE DE FUNCIONALIDADE E EXECUÇÃO DO SEGUNDO TESTE
36 >      it('1.2 - Should get all users in database', async () => {···
49        });
50        // BREVE DESCRIÇÃO DO TESTE DE FUNCIONALIDADE E EXECUÇÃO DO TERCEIRO TESTE
51 >      it('1.3 - Should delete the user that was created in 1.1', async () => {···
56        });
57
58    });
```

Fonte: o autor

Define-se então, no corpo dos métodos it, a lógica dos testes para cada funcionalidade. Assim, considerando a função de criação de um novo usuário, é preciso realizar uma chamada (request) da aplicação (app) passando os atributos necessários para que o registro seja criado, conforme mostrado na Figura 6.

Para se criar um teste deve-se adotar duas linhas de raciocínio. Primeiro deve-se criar o trecho de código que chama a funcionalidade do sistema, aguardar o retorno dela e então validar o que a aplicação devolveu.

Observe pelo exemplo mostrado que, para criar um novo usuário, foi chamada a funcionalidade da aplicação (request(aap)) aguardando o retorno (response = await) para então ser realizada a validação dos atributos recebidos (expect(response.status.toBe(201); e demais atributos nas linhas de código seguintes).

Figura 6 – Testar funcionalidade de criar um novo usuário

```
10    // BREVE DESCRIÇÃO SOBRE O TESTE DA FUNCIONALIDADE E EXECUTAR O PROMEIRO TESTE
11    it('1.1 - Should create a new user', async () => {
12        // REALIZAR CHAMADA A APLICAÇÃO
13        const response = await request(app)
14            .post('/users/register')
15            .set('Content-Type', 'application/json')
16            .send({
17                name: "Test Jest",
18                email: "test001@jest.com.br",
19                password: "1234567!"
20            });
21
22        // ALTERAR VALOR DA VARIAVEL PARA O VALOR RECEBIDO DA APLICAÇÃO
23        _id = response.body._id;
24
25        // INICIAR VALIDAÇÃO COM O JEST
26        expect(response.status).toBe(201);
27        expect(response.body).toHaveProperty("active");
28        expect(response.body).toHaveProperty("_id");
29        expect(response.body).toHaveProperty("name");
30        expect(response.body).toHaveProperty("email");
31        expect(response.body).toHaveProperty("password");
32        expect(response.body).toHaveProperty("createdAt");
33        expect(response.body).toHaveProperty("updatedAt");
34    });
```

Fonte: o autor

Os elementos que são usados para construção da lógica dos testes automatizados são:

- método: responsável por informar o tipo de chamada à aplicação. Os principais métodos são: get, post, put e delete;
- url: é o endereço dos recursos disponíveis na rede, responsável por informar em qual endereço a aplicação está hospedada;
- set: é o cabeçalho HTTP, parte importante da solicitação e resposta da aplicação. Conforme exemplo da Figura 6, a aplicação está sendo informada que receberá atributos (‘Content-Type’) e deverá retornar dados no formato JSON (‘application/json’);
- send: responsável por informar os dados a serem passados para a aplicação, que no exemplo são do usuário a ser adicionado na aplicação (name, email, password);
- async: declara que é um método assíncrono. Quando um método assíncrono é executado ele retorna uma promisse, que deverá retornar um valor;

- await: uma função assíncrona pode conter a expressão await que pausa a execução do método até que a promisse passada seja executada e retorne um valor.

O passo seguinte é receber e validar a resposta da aplicação. Para isso, é necessário utilizar o método expect com elementos:

- variável: responsável por armazenar os atributos que a aplicação retorna. No exemplo, os elementos retornados são armazenados na variável chamada response.
- expect: é um método do Jest, responsável por fazer comparações com o retorno da aplicação. Basicamente, no trecho de código apresentado na Figura 6, são verificados se o retorno da aplicação recebeu os atributos active, _id, name, email, password, createdAt e updatedAt. Além de validar os campos recebidos, também é validado o status que deve ser igual a 201. O Jest.JS oferece um conjunto de tipos de validação utilizando o expect, que é apresentado na sua documentação.

A Figura 7 mostra testes implementados para validar as funcionalidades Listar (get) e Excluir (delete) Usuários da aplicação

Ainda na pasta integration, é necessário ter o arquivo token.test.js que valida o acesso às funcionalidades da aplicação com as credenciais dos testes. Pode-se observar a estrutura do código desenvolvido na Figura 8 e o detalhamento de cada teste nas Figuras 9 a 11.

Figura 7 – Funcionalidades 1.2 (Listar) e 1.3 (Excluir) Usuário

```
35    // BREVE DESCRIÇÃO DO TESTE DE FUNCIONALIDADE E EXECUÇÃO DO SEGUNDO TESTE
36    it('1.2 - Should get all users in database', async () => {
37        const response = await request(app)
38            .get('/users');
39
40        expect(response.status).toBe(200);
41        expect(response.body.users.length).not.toBe(0);
42        expect(response.body.users[0]).toHaveProperty("active");
43        expect(response.body.users[0]).toHaveProperty("_id");
44        expect(response.body.users[0]).toHaveProperty("name");
45        expect(response.body.users[0]).toHaveProperty("email");
46        expect(response.body.users[0]).toHaveProperty("password");
47        expect(response.body.users[0]).toHaveProperty("createdAt");
48        expect(response.body.users[0]).toHaveProperty("updatedAt");
49    });
50
51    // BREVE DESCRIÇÃO DO TESTE DE FUNCIONALIDADE E EXECUÇÃO DO TERCEIRO TESTE
52    it('1.3 - Should delete the user that was created in 1.1', async () => {
53        const response = await request(app)
54            .delete(`/users/${_id}`);
55
56        expect(response.status).toBe(200);
57    });
```

Fonte: o autor

Figura 8 – Testes de funcionalidade de credenciais do sistema

```
1
2   const request = require('supertest');
3   const app = require('../../app');
4
5   describe('1.0 - Should generate token with valid use', () => {
6
7       let _id = null;
8       let token = null;
9 >     it('1.1 - Should create a new user', async () => {···
28      });
29
30 >    it('1.2 - Should active the user that was created in 1.1', async () => {···
43      });
44
45 >    it('1.3 - Should user to log in using valid credentials', async () => {···
58      });
59
60 >    it('1.4 - Should decode the token generated in 1.3', async () => {···
72      });
73
74 >    it('1.5 - Should delete the user that was created in 1.1', async () => {···
79      });
80
81 >    it('1.6 - Should does not send id to delete and return error 400', async () => {···
87      });
88
89  });
```

Fonte: o autor

Figura 9 – Testes de funcionalidade de credenciais do sistema, itens 1.1 e 1.2

```
9       it('1.1 - Should create a new user', async () => {
10          const response = await request(app)
11              .post('/users/register')
12              .set('Content-Type', 'application/json')
13              .send({
14                  name: "Test Jest",
15                  email: "test@jest.com",
16                  password: "1234567!"
17              });
18
19          _id = response.body._id;
20          expect(response.status).toBe(201);
21          expect(response.body).toHaveProperty("active");
22          expect(response.body).toHaveProperty("_id");
23          expect(response.body).toHaveProperty("name");
24          expect(response.body).toHaveProperty("email");
25          expect(response.body).toHaveProperty("password");
26          expect(response.body).toHaveProperty("createdAt");
27          expect(response.body).toHaveProperty("updatedAt");
28      });
29
30      it('1.2 - Should active the user that was created in 1.1', async () => {
31          const response = await request(app)
32              .get(`/users/active?user=${_id}`);
33
34          expect(response.status).toBe(200);
35          expect(response.body).toHaveProperty("token");
36          expect(response.body.user).toHaveProperty("active");
37          expect(response.body.user).toHaveProperty("_id");
38          expect(response.body.user).toHaveProperty("name");
39          expect(response.body.user).toHaveProperty("email");
40          expect(response.body.user).toHaveProperty("password");
41          expect(response.body.user).toHaveProperty("createdAt");
42          expect(response.body.user).toHaveProperty("updatedAt");
43      });
```

Fonte: o autor

ISSN: 2595-5942

Figura 10 – Testes de funcionalidade de credenciais do sistema, itens 1.3 e 1.4

```
45    it('1.3 - Should user to log in using valid credentials', async () => {
46        const response = await request(app)
47            .post('/users/authentication')
48            .set('Content-Type', 'application/json')
49            .send({
50                email: "test@jest.com",
51                password: "1234567!"
52            });
53
54        token = response.body.token;
55        expect(response.status).toBe(200);
56        expect(response.body).toHaveProperty("token");
57
58    });
59
60    it('1.4 - Should decode the token generated in 1.3', async () => {
61        const response = await request(app)
62            .post('/users/decoded')
63            .set('Content-Type', 'application/json')
64            .set('Authorization', token);
65
66        expect(response.status).toBe(200);
67        expect(response.body).toHaveProperty("id");
68        expect(response.body).toHaveProperty("name");
69        expect(response.body).toHaveProperty("email");
70        expect(response.body).toHaveProperty("iat");
71        expect(response.body).toHaveProperty("exp");
72    });
```

Fonte: o autor

Figura 11 – Testes de funcionalidade de credenciais do sistema, itens 1.5 e 1.6

```
74    it('1.5 - Should delete the user that was created in 1.1', async () => {
75        const response = await request(app)
76            .delete(`/users/${_id}`);
77
78        expect(response.status).toBe(200);
79    });
80
81    it('1.6 - Should does not send id to delete and return error 400', async () => {
82        const response = await request(app)
83            .delete(`/users/null`);
84
85        console.log(response.body);
86        expect(response.status).toBe(400);
87    });
```

Fonte: o autor

Como boas práticas de desenvolvimento de testes automatizados, deve-se cumprir o Ciclo do TDD. Na execução das validações, é de fundamental importância que, a cada verificação construída, se execute o sistema de testes e se analise o *feedback* retornado pelo *framework*. A execução dos testes é iniciada com o carregamento do código pelo comando npm test (Figura 12). Tendo sido concluídos os testes, é exibido o *feedback* conforme Figura 13.

ISSN: 2595-5942

Revista EduFatec: educação, tecnologia e gestão
V.2 N.2 – agosto-dezembro/2019

Figura 12 – Sistema carregando os testes

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

PS C:\Users\Cleber Spirlandeli\[          ]\backend> npm test

> backend@1.0.0 test C:\Users\Cleber Spirlandeli\[          ]\backend
> jest


 RUNS  src/__tests__/integration/token.test.js
 RUNS  src/__tests__/integration/users.test.js

Test Suites: 0 of 2 total
Tests:       0 total
Snapshots:   0 total
Time:        8s, estimated 10s
```

Fonte: o autor

Interpretando o *feedback*, segundo Singhal (2019), em primeiro momento, na legenda Test Suites, se verifica que dois testes foram executados e passaram com sucesso. Embora o primeiro *feedback* seja mostrado na tela de execução, também é gerado um arquivo de registro que pode ser acessado por um navegador *web* para a leitura facilitada dos resultados (Figura 14). As visualizações no terminal ou no navegador mostram cada resultado do teste com cores diferentes para destaque - verde, amarelo e vermelho - que equivalem respectivamente aos níveis de cobertura dos testes implementados como satisfatório, intermediário e insatisfatório.

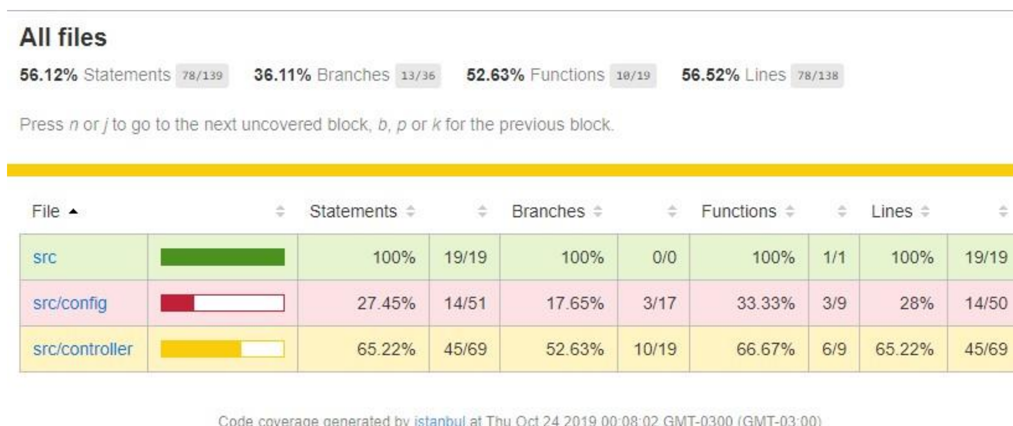
Figura 13 – *Feedback* no *prompt* de comando

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

PS C:\Users\Cleber Spirlandeli\[          ]\backend> npm test

> backend@1.0.0 test C:\Users\Cleber Spirlandeli\[          ]\backend
> jest

 PASS  src/__tests__/integration/users.test.js (9.461s)
 PASS  src/__tests__/integration/token.test.js (11.249s)
----------------------|----------|----------|----------|----------|-------------------|
File                  |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------------------|----------|----------|----------|----------|-------------------|
All files             |    56.12 |    36.11 |    52.63 |    56.52 |                   |
 src                  |      100 |      100 |      100 |      100 |                   |
  app.js              |      100 |      100 |      100 |      100 |                   |
 src/config           |    27.45 |    17.65 |    33.33 |       28 |                   |
  emails.js           |     9.52 |        0 |        0 |     9.52 |... 33,43,44,45,47 |
  token.js            |    42.11 |    23.08 |       60 |    42.11 |... 44,45,46,51,52 |
  upload.js           |    36.36 |        0 |        0 |       40 | 11,12,16,17,18,19 |
 src/controller       |    65.22 |    52.63 |    66.67 |    65.22 |                   |
  LikeController.js   |    28.57 |      100 |        0 |    28.57 |       7,9,11,13,15 |
  PostController.js   |    31.25 |      100 |        0 |    31.25 |... 21,28,30,38,40 |
  UsersController.js  |    82.61 |    52.63 |      100 |    82.61 |... 80,100,103,118 |
----------------------|----------|----------|----------|----------|-------------------|

Test Suites: 2 passed, 2 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        13.988s
Ran all test suites.
```

Fonte: o autor

 ISSN: 2595-5942

Figura 14 – Página Inicial Feedback Web



All files

56.12% Statements 78/139 36.11% Branches 13/36 52.63% Functions 10/19 56.52% Lines 78/138

Press n or j to go to the next uncovered block, b, p or k for the previous block.

File		Statements		Branches		Functions		Lines	
src		100%	19/19	100%	0/0	100%	1/1	100%	19/19
src/config		27.45%	14/51	17.65%	3/17	33.33%	3/9	28%	14/50
src/controller		65.22%	45/69	52.63%	10/19	66.67%	6/9	65.22%	45/69

Code coverage generated by istanbul at Thu Oct 24 2019 00:08:02 GMT-0300 (GMT-03:00)

Fonte: o autor

Os elementos informativos do *feedback*, são: File, Statements, Branches, Functions e Lines, que representam as coberturas de:

- Files: pastas e arquivos do sistema testado;
- Statements: arquivos executados que contém instruções ou declarações prováveis de serem testados;
- Branches: instruções condicionais criam ramificações de código que podem não ser executadas (por exemplo, if / else). Essa métrica informa quantas das ramificações foram executadas;
- Functions: função prováveis de serem testadas que existem no sistema;
- Lines: porcentagem de linhas do sistema que foram validados.

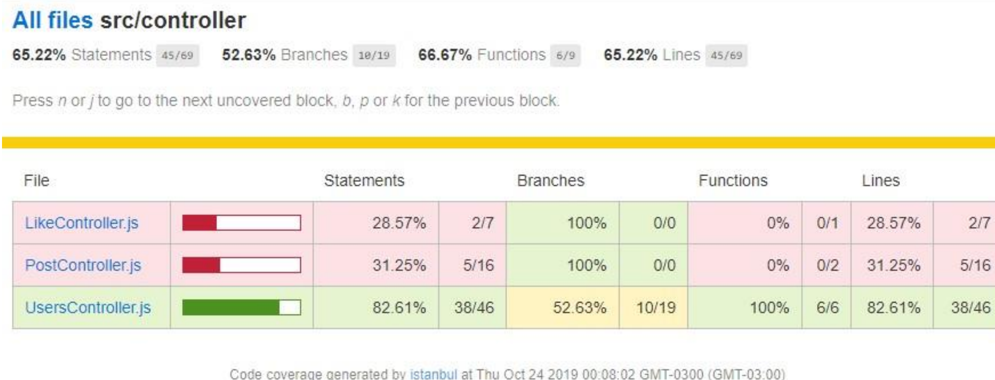
No exemplo da Figura 14, os testes implementados em src têm uma cobertura satisfatória, enquanto src/config cobre insatisfatoriamente e src/controller cobre em nível intermediário.

É possível ainda se obter relatório de um teste específico com maior detalhe individual dos componentes ou arquivos de codificação, segundo mostrado para o src/controller na Figura 15.

Para cada teste são representados o resultado em forma gráfica, em porcentagem de comandos (Statements) usados, bem como pela relação numérica dos comandos executados sobre o total de comandos definidos no teste, dentre outras informações. Para se visualizar os detalhes do teste, é permitido acessar o arquivo (clicando sobre seu nome na tabela) como mostra a Figura 16 para o arquivo UsersController.js.

ISSN: 2595-5942

Figura 15 – *Feedback* da página web dos arquivos da pasta src/controller



All files src/controller

65.22% Statements 45/69 52.63% Branches 10/19 66.67% Functions 6/9 65.22% Lines 45/69

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
LikeController.js		28.57%	2/7	100%	0/0	0%	0/1	28.57%	2/7
PostController.js		31.25%	5/16	100%	0/0	0%	0/2	31.25%	5/16
UsersController.js		82.61%	38/46	52.63%	10/19	100%	6/6	82.61%	38/46

Code coverage generated by istanbul at Thu Oct 24 2019 00:08:02 GMT-0300 (GMT-03:00)

Fonte: o autor

Figura 16 – *Feedback* individual do arquivo UsersController.js

```
1  2x const md5 = require('md5');
2  2x const Users = require('../models/Users');
3  2x const Email = require('./../config/emails');
4  2x const { GenerateToken, DecodedToken } = require('./../config/token');
5
6  2x module.exports = {
7
8        // GET - all users
9        async index(req, res) {
10
11 1x        const users = await Users.find().sort('-createdAt');
12
13 1x        return res.status(200).json({ users });
14       },
15
16       // POST - register
17       async register(req, res) {
18
19 2x        try {
20 2x            let { name, email, password } = req.body;
21 2x            password = md5(password + process.env.SALT_KEY_PASSWORD);
22
23 2x            const userValid = await Users.find({ email });
24
25 2x          E if (userValid.length === 0) {
26 2x                const user = await Users.create({
27                        name,
28                        email,
29                        password
30                    });
```

Fonte: o autor

O relatório individual mostra, na coluna à esquerda, a quantidade de vezes que a linha de código do teste foi executada. Neste relatório também são visíveis, como mostra a Figura 17 nas cores vermelho e amarelo, as linhas não validadas e trechos do código que não foram usados em testes. Ainda se recebe a informação, pelo símbolo E (na linha 45), que indica qual ramificação da instrução condicional o teste executou.

Nos testes exemplificados não houve ocorrências de falhas, entretanto quando há falha o *feedback* informa qual teste não obteve sucesso. Para demonstrar esse tipo de ocorrência, na pasta unit foi criado o arquivo error.test.js para gerar uma falha simples (Figura 18). No item 1.1, o teste compara dois valores que são iguais (linha 5) e no item 1.2, são comparados dois valores que são diferentes e, portanto, este teste deve falhar. Observa-se que quando ocorre uma falha (Figura 19), no *feedback* é apresentado em qual arquivo de teste qual item falhou (linha 9). Também são informados os valores esperados (Expected) e o recebido da aplicação (Received) que neste caso foi definido como o valor constante 400. Como o esperado é o valor constante 200 o teste reportou a falha.

Figura 17 – *Feedback* individual do arquivo UsersController.js

```
28                    email,
29                    password
30                });
31
32                // send email
33                //Email.send(user);
34
35  2x            return res.status(201).json(user);
36            }
37
38            return res.status(400).json(userValid);
39
40        } catch (error) {
41            return res.status(500).json(error.message);
42        }
43    },
44
45    async decoded(req, res) {
46  1x    const token = req.headers['authorization'] || req.body.token || req.query.token || null;
47
48  1x    E if (token) {
49  1x        DecodedToken(token, res);
50        } else {
51            res.status(200).json({ token: false });
52        }
53    },
54
55    // POST - authentication
56    async authentication(req, res) {
57
58  1x    try {
59
60  1x        let { email, password } = req.body;
61  1x        password = md5(password + process.env.SALT_KEY_PASSWORD);
62
63  1x        const user = await Users.find({
64                $and: [
65                    { "email": email },
66                    { "password": password },
```

Fonte: o autor

Dessa forma é possível se identificar qual teste falhou e em que trecho do código, e o desenvolvedor precisa analisar qual foi o motivo da falha para corrigi-lo, evitando que a falha passe despercebida e o erro ocorra com o sistema em produção.

Figura 18 – Estrutura de arquivo de teste para gerar falha

```
1
2   describe('1.0 - Failed test example', () => {
3
4       it('1.1 - Example to pass', () => {
5           expect(200).toBe(200);
6       });
7
8       it('1.2 - Example not to pass',  () => {
9           expect(200).toBe(400);
10      });
11
12  });
```

Fonte: o autor

Figura 19 – Feedback de uma falha

```
> jest

FAIL src/__tests__/unit/error.test.js (5.824s)
  ● 1.0 - Failed test example › 1.2 - Example not to pass

    expect(received).toBe(expected) // Object.is equality

    Expected: 200
    Received: 400

       7 |
       8 |     it('1.2 - Example not to pass',  () => {
    >  9 |         expect(400).toBe(200);
         |                     ^
      10 |     });
      11 |
      12 | });

      at Object.toBe (src/__tests__/unit/error.test.js:9:21)

---------------------|----------|----------|----------|----------|-------------------|
File                 |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
---------------------|----------|----------|----------|----------|-------------------|
All files            |        0 |        0 |        0 |        0 |                   |
 src                 |        0 |      100 |        0 |        0 |                   |
  app.js             |        0 |      100 |        0 |        0 |... 24,27,30,33,39 |
 src/config          |        0 |        0 |        0 |        0 |                   |
  emails.js          |        0 |        0 |        0 |        0 |... 33,43,44,45,47 |
  token.js           |        0 |        0 |        0 |        0 |... 44,45,46,51,52 |
  upload.js          |        0 |        0 |        0 |        0 |... 12,16,17,18,19 |
 src/controller      |        0 |        0 |        0 |        0 |                   |
  LikeController.js  |        0 |      100 |        0 |        0 |  1,3,7,9,11,13,15 |
  PostController.js  |        0 |      100 |        0 |        0 |... 21,28,30,38,40 |
  UsersController.js |        0 |        0 |        0 |        0 |... 12,113,115,118 |
---------------------|----------|----------|----------|----------|-------------------|
Test Suites: 1 failed, 1 of 3 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        13.863s
Ran all test suites.
npm ERR! Test failed.  See above for more details.
```

Fonte: O autor.

Considerações finais

Desenvolver *software* é uma atividade fundamentalmente de raciocínio lógico para instruir uma máquina a executar processos de cálculos numéricos. É uma tarefa que exige concentração e atenção a detalhes, e muitas vezes, dada a complexidade

ISSN: 2595-5942

das regras a serem implementadas no código, falhas de lógica passam despercebidas. Com o objetivo de tornar o exercício da programação mais fácil e seguro, foram criadas ferramentas de automação de testes de *software* que foram o objeto desta pesquisa. Buscou-se, através de pesquisa bibliográfica exploratória e estudo de caso, aprofundar o conhecimento sobre os conceitos de Desenvolvimento Guiado por Testes, as características e aplicações de uma das ferramentas disponíveis para implementar testes automatizados, bem como aplicar os métodos em um caso real.

Foram utilizadas as principais referências mundiais sobre os tópicos relacionados ao tema, bem como estudado o *framework* de testes Jest.JS.

A partir do entendimento de como se aplicar a ferramenta buscou-se implementar, em um contexto real, rotinas de testes para se verificar e entender os resultados oferecidos pelos testes.

Considera-se que a pesquisa alcançou seus objetivos como apresentado e discutido neste artigo, e que ampliou a visão dos autores sobre o uso de TDD no desenvolvimento de *software*.

Como projeto futuro, sugere-se que seja testado o uso de Jest.JS em um cenário real de desenvolvimento de um sistema inteiro ou, pelo menos, no desenvolvimento de um módulo completo de um sistema, coletando dados sobre os tempos usados na implementação dos testes em relação aos tempos necessários para atualizações e manutenções do sistema na fase de utilização. Neste caso será possível verificar resultados de performance dos desenvolvedores em prazos mais longos, envolvendo outras atividades do ciclo de desenvolvimento posteriores à de construção do código.

Referências

4LINUX. **Diferenças entre integração, entrega e implantação contínua.** 2017. Disponível em: https://blog.4linux.com.br/integracao-entrega-implantacao-continua/. Acesso em: 2 out.2019

ANICHE, Maurício. **Test-Driven Development: Teste e Design no Mundo Real.** 1. ed. São Paulo: Casa do Código, 2012.

______. **Test-Driven Development: Teste e Design no Mundo Real com .NET.** 1. ed.: Casa do Código, 2014.

DEVMEDIA. **TDD: fundamentos do desenvolvimento orientado a teste.** 2013. Disponível em: https://www.devmedia.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/28151. Acesso em: 2 out.2019.

FOWLER, Martin. **Test Driven Development.** Disponível em: https://martinfowler.com. Acesso em: 2 out.2019

GEORGE, Boby; WILLIAMS, Laurie. **An Initial Investigation of Test-Driven Development in Industry**. 2003. Disponível em: https://dl.acm.org/citation.cfm?id=952753. Acesso em: 28 out.2019.

GITHUB. **Facebook/jest.** 2019. Disponível em: https://github.com/facebook/jest/network/dependents. Acesso em: 6 nov.2019.

GOMES, André Faria. **Agile - Desenvolvimento de software com entregas frequentes e foco no valor de negócio.** 1. ed. São Paulo: Casa do Código, 2013.

HANSSON, David Heinemeier, **TDD is dead. Long live testing.** 2014. Disponível em: https://dhh.dk/2014/tdd-is-dead-long-live-testing.html. Acesso em: 07 out.2019.

iMASTERS**,** 7 motivos por que TDD falhou em ser mais utilizado. 21/07/2014. Disponível em: https://imasters.com.br/agile/7-motivos-por-que-tdd-falhou-em-ser-mais-utilizado. Acesso em: 07 out.2019.

JANZEN, David Scott. 2006. **An Empirical Evaluation of the Impact os Tes-Driven Development on Software Quality.** Disponível em: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.9412&rep=rep1&type=pdf. Acesso em: 28 out.2019.

JESTJS. **JEST 24.9.** Disponível em: https://jestjs.io. Acesso em: 16 out.2019.

MYKLEBUST, Thor; *et al.* **Scrum, documentation and the IEC 61508-3:2010 software standard.** sd. Disponível em: http://meetingsandconferences.com/psam12/proceedings/paper/paper_31_1.pdf. Acesso em: 01 out.2019.

NAGAPPAN, Nachiappan; et al. **Realizing Quality Improvement Through Tes-Drive Development: resultas and experiences of four industrial teams.** 2008. Disponível em: https://people.engr.ncsu.edu/gjin2/Classes/591/Spring2017/case-tdd-b.pdf. Acesso em: 29 out.2019.

NODEBR. **O que é Node.js?** 2016. Disponível em: http://nodebr.com/o-que-e-node-js/. Acesso em: 12 out.2019.

NPM. **Jest.** 2019. Disponível em: https://www.npmjs.com/package/jest. Acesso em: 6 nov.2019.

PRESSMAN, Roger S. **Engenharia de Software Uma Abordagem Profissional.** 7. ed. São Paulo: McGraw-Hill, 2011.

ISSN: 2595-5942

SAUVÉ, Jacques P. **Frameworks: o que é um Framework.** sd. Disponível em: http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm. Acesso em: 12 out.2019.

SINGHAL, Krishankant. **How to read Test Coverage report generated using Jest.** 2019. Disponível em https://medium.com/@krishankantsinghal/how-to-read-test-coverage-report-generated-using-jest-c2d1cb70da8b. Acesso em: 28 out.2019.

TABLELESS. **O que é um framework.** sd. Disponível em: https://tableless.github.io/iniciantes/manual/js/o-que-framework.html Acesso em 12 out.2019.

TREINAWEB. **O que é front-end e back-end?** 2017. Disponível em https://www.treinaweb.com.br/blog/o-que-e-front-end-e-back-end. Acesso em: 13 out.2019.

ISSN: 2595-5942