Proposta de Trabalho de Conclusão de Curso

Vinícius Guimarães Pereira

Testify - uma plataforma para aprendizado de testes automatizados de software

Orientador: Prof. Dr. Alfredo Goldman
Coorientador: João Francisco Lino Daniel

1 Introdução

O desenvolvimento de testes é trabalhoso e muitas vezes burocrático de ser realizado, mas os benefícios que traz compensa o tempo utilizado em seu desenvolvimento e isto é fato para muitos desenvolvedores que conhecem do assunto.

Há muitos conceitos relacionados a testes automatizados, como boas práticas e técnicas que podem ajudar em seu desenvolvimento. Hoje nota-se uma carência de conhecimento sobre o assunto em muitos desenvolvedores, que aprenderam somente os tipos de testes clássicos e conceitos básicos. Isto ocorre pela falta de interesse no assunto e a falta de um conteúdo organizado na internet, que mostra como prosseguir com o aprendizado, como um guia para os desenvolvedores.

O *Testify* irá buscar resolver estes problemas, sendo uma plataforma que oferece um guia para o aprendizado em testes automatizados, exercícios práticos e a oportunidade de contribuir com o aprendizado de outros. Além disso, será desenvolvido utilizando conceitos importantes de arquitetura, boas práticas de desenvolvimento de software e ferramentas muito utilizadas no mercado.

A **seção 2** deste documento contém a revisão de literatura sobre assuntos relacionados a Código Limpo, Testes Automatizados e Arquitetura de Software, a **seção 3** possui a proposta deste projeto e a **seção 4** contém o plano de trabalho do projeto, com as tarefas a serem realizadas e as estimativas de tempo de cada uma delas.

2 Revisão de Literatura

2.1 Código Limpo

Um código é considerado sujo quando para entendê-lo é necessário que caminhemos por diversos pontos dele em busca de informações, quando há um alto grau de acoplamento, muitas dependências e que não segue as boas práticas de desenvolvimento de software [1].

Este tipo de código traz diversos malefícios, como diminuir a produtividade dos desenvolvedores que trabalham com ele, fazendo com que gastem muito tempo tentando entender as alterações. O surgimento de bugs aumenta rapidamente por conta do alto nível de acoplamento e torna-se praticamente impossível de ser consertado.

Na figura 1 está um gráfico de produtividade por tempo, que representa o nível de produtividade que os desenvolvedores possuem trabalhando com um código sujo ao longo do tempo. Pode-se notar que a produtividade diminui rapidamente a partir de certo ponto quando lidamos com código sujo. Este tipo de comportamento acontece bastante dentro de empresas, onde times de desenvolvedores que estão com a produtividade já diminuída, por conta da “sujeira” nos códigos do sistema, são forçados a aumentá-la, gerando um acúmulo ainda maior de código sujo [1].

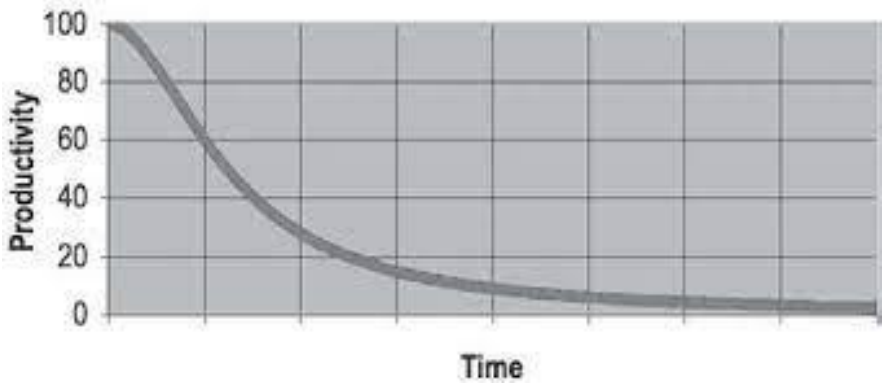

Figura 1 - Produtividade x Tempo em relação ao código sujo

Por outro lado, um código é considerado limpo quando o leitor consegue entendê-lo antes mesmo de procurar entender os detalhes de implementação, com a lógica entregando exatamente aquilo que é esperado. Este tipo de código possui poucos pontos de acoplamento, poucas dependências e tenta ao máximo respeitar as boas práticas de desenvolvimento de software.

Manter o código de um sistema limpo é um trabalho difícil e trabalhoso, mas traz diversos benefícios, como por exemplo:

- A facilidade de manutenção;
- O aumento da produtividade dos desenvolvedores;

- Facilita a integração de novos programadores ao time, pois eles entendem com mais clareza o sistema;
- Previne a ocorrência de bugs pelo baixo nível de acoplamento e dependências.

Além disso, existem algumas regras que podem ser seguidas para escrever um código limpo, alguns exemplos são:

- Deve-se utilizar nomes significativos, mesmo que estes nomes pareçam grandes. Seguir esta regra facilita o entendimento do desenvolvedor que está implementando (que não se perde em sua lógica) e a busca de nomes nos editores de código atuais.
- Funções e métodos devem possuir uma única responsabilidade, para que o resultado de uma entrada seja sempre o esperado.
- Escreva o mínimo de comentários possível, pois o próprio código deve ser auto explicativo. Caso seja necessário adicionar um comentário para explicar um trecho de código, muito provavelmente é o próprio código que precisa de refatoração.

2.2 Testes automatizados de software

Testes são ferramentas utilizadas para verificar a aderência do código aos requisitos funcionais e não-funcionais do sistema. Hoje em dia os desenvolvedores utilizam diversas ferramentas para executarem estes testes de forma automática, criando o que alguns chamam de Códigos Auto-testáveis (Self-testing Code) [2].

Uma das principais vantagens de se utilizar testes automatizados é a prevenção de bugs. Qualquer mudança indesejada no comportamento do código é identificada durante a execução dos testes, e qualquer novo bug que seja encontrado no código faz com que um novo caso de teste seja implementado cobrindo este caso em específico. Além disso, também ajuda a manter a qualidade do código e a trazer mais confiança aos desenvolvedores para implementarem mudanças.

Hoje em dia existem diversos tipos de testes automatizados de software, cada um com seus próprios objetivos, vantagens e desvantagens. Aqui estão alguns exemplos de tipos de teste:

- Testes de unidade: Este geralmente é o primeiro tipo de teste que os desenvolvedores aprendem a implementar. O objetivo dele é testar uma unidade individual do sistema. A ideia deste tipo de teste automatizado é que apenas uma unidade deve ser testada por vez. Para que isto funcione, geralmente é aplicado o conceito de injeção de dependências. Uma outra forma é sobrepor a implementação de funções ou métodos que são chamados dentro daqueles que estão sendo testados - uma funcionalidade fornecida dentre as diversas ferramentas de testes unitários (JUnit, Jest, Mocha etc).
- Testes de integração: Os testes de integração, ao contrário dos testes de unidade, buscam testar a conexão de duas ou mais partes do sistema, para certificar-se de que a *integração* delas está funcionando corretamente.

- Testes de Ponta-a-Ponta: Diferente dos anteriores, testam o sistema por completo, com todos os componentes conectados. Utilizado para certificar-se de que o fluxo geral está funcionando conforme os requisitos, geralmente simulando um caso de uso para verificar se é possível concluí-lo sem erros. Verificando também se as chamadas aos servidores e a interação com outros sistemas estão sendo realizadas corretamente.
- Testes de aceitação: São testes realizados para verificarem se o produto atende a todos os requisitos especificados. Neste caso, são muito utilizados ambientes que simulam o ambiente de produção.

Dada a quantidade de tipos de teste de software que existem, é muito questionado qual deve receber a maior atenção dos desenvolvedores. Em 2009 foi apresentado o conceito de Pirâmide de Testes [3], uma pirâmide formada por 3 camadas: na base, testes de unidade; no meio, testes de serviço; e na ponta, testes de interface – a Figura 2 ilustra esse modelo. Em direção a base da pirâmide estão os tipos de teste com maior isolamento, isto é, em que o menor número de componentes do sistema é testado por vez, enquanto para a ponta estão aqueles com maior integração, em que mais de um componente do sistema é testado por vez. Outro conceito aplicado à pirâmide é que em direção à base estão os testes que devem ser executados de forma mais rápida, enquanto em direção à ponta estão os mais lentos.

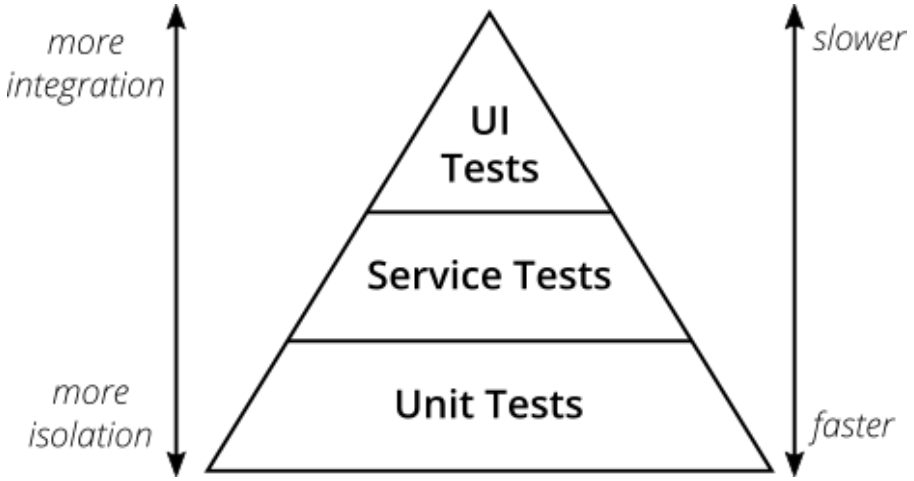

Figura 2 - Pirâmide de testes

Hoje em dia, porém, muitos acreditam que os nomes citados na pirâmide não representam todos os cenários. Em alguns casos o formato da pirâmide chega a mudar dependendo do contexto em que ela é aplicada. Um exemplo é a Pirâmide criada por Kent Dodds, apelidada de “O Troféu de Testes” [4], que representa o retorno do investimento nos tipos de testes em se tratando da linguagem de programação JavaScript (Figura 2). Um tema interessante a ser retratado, pois mostra a importância dos testes automatizados para empresas, trazendo grande retorno financeiro quando aplicado, já que depois de desenvolvidos salvam muito tempo de desenvolvimento a longo prazo, pois evitam bugs e mantém a qualidade do código alta e constante.

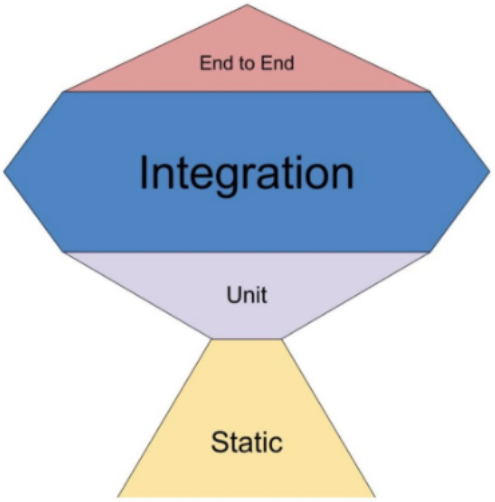

Figura 3 - Pirâmide de testes por Kent Dodds

2.3 Desenvolvimento orientado por testes (*Test Driven Development - TDD*)

O *TDD* é uma técnica de desenvolvimento de software em que os testes de unidade são escritos antes da implementação do código.

O processo do *TDD* pode ser descrito de forma abstrata pelos seguintes passos [5]:

1. É criado um teste de unidade para uma nova funcionalidade;
2. O teste é executado falhando inicialmente, como esperado;
3. Implementa-se o mínimo necessário da funcionalidade para que o teste seja aceito;
4. Depois que o teste passa a ser aceito, deve refatorar-se o código implementado para livrar-se de duplicidade, caso necessário;
5. O processo recomeça, caso o desenvolvedor julgue necessário o desenvolvimento de mais testes.

Este mesmo processo se segue quando surgem novas funcionalidades no sistema e ele é conhecido como ciclo **vermelho-verde-refatora** (*red-green-refactor*) [5], mostrado na Figura 4:

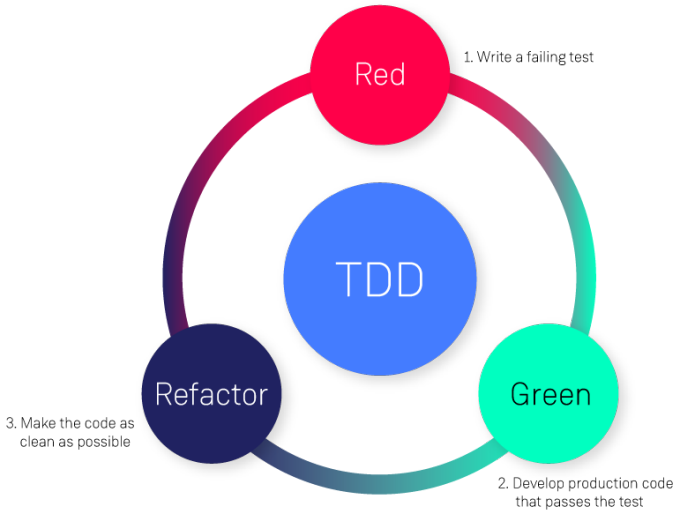

Figura 4 - Ciclo verde-vermelho-refatora

Existem uma série de princípios que ajudam o desenvolvedor a praticar o *TDD* no durante o desenvolvimento de um sistema, estes são:

- *Mantenha isso simples, estúpido (KISS - Keep it simple, stupid)*: valoriza a simplicidade da implementação de um código;
- *Você não vai precisar disso (YAGNI - You ain't gonna need it)*: diz que uma funcionalidade não deve ser implementada enquanto não é necessária;
- *Simule enquanto puder (Fake it, till you make it)*: sugere que deve-se simular uma funcionalidade até que a real implementação seja necessária, levando o desenvolvedor a deixar o código o mais simples possível;
- *Não se repita (DRY - Don't repeat yourself)*: deve-se diminuir a duplicação de código. A reutilização de implementações através de funções e classes é encorajada.

Esta prática traz maior foco no comportamento desejado, fazendo com que o desenvolvedor foque no que a funcionalidade **deve** fazer e não na implementação. Além disso, *TDD* ajuda a deixar o código mais simples, pois os requisitos são implementados somente para que satisfaçam os testes. E, por fim, esta metodologia traz uma reflexão maior sobre a responsabilidade daquilo que está sendo implementado.

2.4 Arquitetura de Software

Pode-se entender arquitetura de software como a organização dos componentes do sistema, quais suas características e como irão interagir entre si [6].

Uma boa arquitetura cria cada componente com uma única responsabilidade e busca diminuir ao máximo o nível de acoplamento do sistema, facilitando a sua extensão.

A arquitetura define propriedades que irão perdurar por muito tempo em um sistema. Muito esforço é gasto quando há necessidade de uma mudança na estrutura conceitual de um sistema, pois a implementação surgiu a partir destas definições, sendo necessário uma grande quantidade de refatorações para que a mudança seja implementada. Por isso

deve-se dar a devida importância ao período de design do sistema, para que uma arquitetura limpa e extensível seja criada.

Há diversos benefícios a longo prazo que uma boa arquitetura traz, mas um dos mais importantes, ainda mais para empresas, é a economia de esforço para a implementação de novas funcionalidades no futuro, gerando resultados mais rápidos [6].

2.5 Arquitetura em Microsserviços

A arquitetura em microsserviços busca separar o sistema em componentes que serão executados em processos independentes, chamados de microsserviços, que trabalham em conjunto. Cada microsserviço deve ter uma única responsabilidade [7].

Este formato de arquitetura traz diversos benefícios para o sistema e para a própria organização que o utiliza, sendo alguns exemplos:

- Traz a oportunidade de fazer com que cada microsserviço seja implantado de forma isolada dos outros, portanto quando ocorre uma mudança em um dos serviços, somente ele é construído novamente e implantado, enquanto os outros permanecem constantes.
- Cria um sistema extensível, pois novos microsserviços podem ser integrados ao sistema facilmente.
- Traz escalabilidade para organizações que decidem o utilizar, pois pode-se alocar pequenos times para cada microsserviço, tornando o trabalho mais organizado e concentrado, já que cada time pode tomar suas próprias decisões, ter suas próprias metas e prazos.

Algumas empresas de software gigantes do mercado utilizam este método de arquitetura por conta dos benefícios que ela traz, alguns exemplos são a *Amazon*, *Google* e *Facebook*.

Existem diversas formas de se estabelecer a comunicação entre os microsserviços, algumas delas são:

- Síncrona através de chamadas de API (Requisição/Resposta): um serviço realiza uma chamada para outro e permanece bloqueado enquanto aguarda por uma resposta.
- Assíncrona baseada em eventos: ao finalizar um processo, o serviço publica um evento com o resultado. Todos os outros serviços inscritos para receber o evento receberão este resultado e trabalharão com ele. Este tipo de comunicação traz o benefício de ser altamente desacoplado, pois um serviço que publica eventos não conhece os serviços que irão recebê-los. Este formato de comunicação é geralmente feito através de *Message Brokers* [7].

A arquitetura de microsserviços tornou-se muito popular nos últimos anos, porém é um conceito que se aplica à camada de backend. Com isso, surgiu-se a ideia de trazer este conceito para o desenvolvimento da interface de aplicações - camada de frontend - abordagem que ficou conhecida como Micro Frontends, onde aplicam-se as mesmas diretrizes determinadas pela arquitetura de microsserviços [8].

3 Proposta

Sabemos que testes automatizados são extremamente importantes e trazem diversos benefícios a um sistema, porém a falta de conteúdo mais completos sobre testes na internet faz com que muitos desenvolvedores se limitem a aprender apenas os conceitos básicos. A falta de conhecimento no assunto faz com que os testes sejam mal desenvolvidos, tornando os sistemas passíveis de bugs e outros problemas. Além disso, faz com que os desenvolvedores não entendam os benefícios que os testes automatizados trazem, quando na verdade deveria ser um dos pontos de foco no desenvolvimento de um sistema.

O objetivo proposto por este projeto é fazer com que o aprendizado sobre testes automatizados se torne uma tarefa menos trabalhosa, centralizando conteúdos que ensinam sobre conceitos e metodologias relacionados a testes, exercícios práticos para fixação do aprendizado e guias criados por outros usuários. Dessa forma, desenvolvedores que desejam aprender sobre o tema poderão fazer isso sem a necessidade de acessar múltiplas referências.

O projeto propõe atingir seu objetivo criando o *Testify*, uma plataforma aberta com o objetivo de servir como um centralizador de conhecimentos sobre testes automatizados. Ela tentará resolver os problemas descritos anteriormente servindo como um guia para os desenvolvedores. As principais funcionalidades que o *Testify* irá possuir serão:

- Uma seção específica para o aprendizado de conceitos e ferramentas, com o conteúdo baseado em livros de autores renomados no assunto e em documentações de ferramentas amplamente difundidas no mercado.
- Uma seção contendo exercícios práticos onde os usuários poderão escrever testes ou códigos que atendam testes, em um formato baseado no *Koans*.
- Um fórum onde programadores poderão contribuir com o aprendizado de outros através de guias escritos, em um formato baseado na comunidade *Dev.to* e nos cursos do *Educative.io*.

Em relação ao desenvolvimento da plataforma, serão aplicados conceitos relacionados à arquitetura em microsserviços, micro frontends e de boas práticas de desenvolvimento de software. Dessa forma a plataforma poderá ser atualizada e estendida facilmente no futuro, com novas seções, funcionalidades, conteúdo etc, possibilidades fornecidas por estas metodologias, como mencionado anteriormente na revisão de literatura. Também serão utilizadas ferramentas de *Cloud* (como o AWS, por exemplo) e frameworks muito utilizados no mercado (tal como o React.js).

A construção da plataforma se iniciará com um estudo aprofundado sobre testes automatizados, para que posteriormente o conteúdo conceitual seja definido e escrito. Os resultados desta primeira etapa devem ser armazenados em um local que permita o acesso pela plataforma, a facilidade na edição do conteúdo e a contribuição de usuários. O desenvolvimento dos exercícios práticos também seguirá da mesma forma.

Depois disso, haverá um aprofundamento nos conceitos de arquitetura mencionados anteriormente (microsserviços e micro frontends), para que o planejamento da plataforma possa ser feito com mais embasamento.

O planejamento da plataforma deverá decidir os seguintes tópicos:

- Os requisitos da interface da plataforma;
- Arquitetura do frontend;
- Os requisitos do backend da plataforma;
- Arquitetura do backend;
- Definição das ferramentas que serão utilizadas;

A implementação começará pelo desenvolvimento das seções estáticas da plataforma, ou seja, a seção do conteúdo conceitual e dos exercícios práticos, utilizando os conteúdos construídos nas primeiras etapas do trabalho. Posteriormente será implementado a seção do fórum de contribuições dos usuários que, por ser mais complexa, necessitará de um maior tempo de desenvolvimento comparado às outras seções. Além disso, todo o desenvolvimento seguirá de acordo com os requisitos definidos no planejamento.

A escrita da monografia acompanhará praticamente todo o trabalho e a preparação da apresentação será feita nos últimos meses.

O resultado esperado deste projeto é que a ferramenta criada sirva tanto como ponto de partida, para aqueles que desejam começar a aprender sobre testes automatizados, quanto uma área para aprofundar seus conhecimentos no assunto. Também que os desenvolvedores utilizem a plataforma para contribuir para o aprendizado dos outros. Dessa forma, é esperado que o projeto possa contribuir para o aumento do interesse e do conhecimento dos desenvolvedores em testes automatizados.

4 Plano de trabalho

Determinando as estimativas de cada etapa do trabalho, temos os seguintes itens que definem as etapas de forma mais simples e em seguida a tabela contendo o período de trabalho de cada uma:

1. Estudo e aprofundamento em testes automatizados para a criação do conteúdo;
2. Definição e escrita do conteúdo conceitual da plataforma;
3. Estudo e aprofundamento nos conceitos e ferramentas necessários para o desenvolvimento da plataforma;
4. Planejamento geral para o desenvolvimento;
5. Implementação das seções estáticas (conceitual e de exercícios práticos);
6. Implementação da seção de contribuições dos desenvolvedores;
7. Escrita da monografia;
8. Preparação da apresentação.

	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
1	x	x	x					
2		x	x					
3			x	x	x			
4				x	x			
5					x	x		
6					x	x		
7						x	x	x
8			x	x	x	x	x	x
9							x	x

Referências

[1] MARTIN, Robert. **Clean Code: A Handbook of Agile Software Craftsmanship**. 2009.

[2] FOWLER, Martin**. Self Testing Code.** MartinFowler. Disponível em https://martinfowler.com/bliki/SelfTestingCode.html. Acesso em 01/05/2014.

[3] COHN, Mike. **Succeeding with Agile: Software Development Using Scrum**. 2009.

[4] VOCKE, Ham**. The Practical Test Pyramid.** MartinFowler. Disponível em https://martinfowler.com/articles/practical-test-pyramid.html. Acesso em 26/02/2018.

[5] ANICHE, Mauricio. **Test-Driven Development - Teste e Design no Mundo Real**. 2012.

[6] FOWLER, Martin**. Software Architecture Guide**. MartinFowler. Disponível em https://martinfowler.com/architecture/. Acesso em 01/08/2019.

[7] NEWMAN, Sam. **Building Microservices.** 2012.

[8] GEERS, Michael. **Micro Frontends in Action**. 2020.

[9] ANICHE, Mauricio. **Effective Software Testing - A developer's guide**. 2021.