Qualidade de testes de software: um estudo sobre o impacto das Diretrizes de Contribuição

Laura Godinho Barroso
Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, Brasil
laura.barroso@dcc.ufmg.br

André Cavalcante Hora
Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, Brasil
andrehora@dcc.ufmg.br

***Abstract*—A presente pesquisa tem como objetivo analisar o impacto das diretrizes de contribuição na qualidade dos testes de unidade de projetos hospedados no GitHub, escritos em TypeScript, selecionados com base em critérios de popularidade e atividade recente. A metodologia envolve a coleta e organização manual dos trechos de guideline que especificam sobre como elaborar testes de unidade com qualidade; análise sobre o cumprimento de tais trechos nos respectivos projetos; e análise comparativa entre os resultados obtidos. Ao final da pesquisa, espera-se conseguir avaliar se a especificação de como escrever testes com qualidade nas Diretrizes de Contribuição realmente impacta a escrita dos testes.**

***Index Terms*—Testes de software, Qualidade de testes, Projetos colaborativos, Diretrizes de contribuição, Engenharia de software**

I. Introdução

A colaboração em projetos de código aberto tem se tornado uma prática cada vez mais presente no desenvolvimento de software. Plataformas como o GitHub viabilizam esse modelo, permitindo que desenvolvedores de diferentes perfis contribuam com código, documentação, correções de bugs e testes. Dada a grande quantidade de projetos com muitos colaboradores, aspectos importantes da engenharia de software como manutenabilidade, legibilidade e segurança enviados aos projetos ganham mais relevância. A manutenabilidade, por exemplo, é um dos atributos de qualidade definidos pela norma ISO/IEC 25010, sendo essencial para a sustentabilidade de um sistema ao longo do tempo. Como destaca Valente [2], "os custos de manutenção frequentemente representam até 90% do custo total de um sistema ao longo de sua vida útil", o que reforça a importância de garantir que o código seja fácil de compreender, modificar e testar. Em especial, a presente pesquisa busca contribuir para a resolução do problema de testes de software enviados sem qualidade.

Testes de software sem qualidade podem impactar não apenas em aspectos como compreensão do que está sendo testado, como na manutenabilidade dos projetos e, até mesmo, gargalos relacionados ao tempo decorrentes de tais tarefas. A fim de tentar resolver esse problema, alguns repositórios incluem nas Diretrizes de Contribuição boas práticas para ajudar os colaboradores ao elaborar testes de software. A corretude dos testes pode ser facilmente verificada por meio de diversos frameworks. Já a qualidade exige maior atenção e análise qualitativa.

Dessa forma, a presente pesquisa tem como objetivo geral analisar a qualidade de testes enviados por colaboradores para projetos de código aberto relevantes e recentes. De forma mais específica, tem-se como objetivo avaliar o quão eficiente é fazer Diretrizes de Contribuição que auxiliem como elaborar bons testes de software e se as boas práticas para elaboração de testes são conhecidas e usadas mesmo quando não apresentadas no projeto.

II. Referencial Teórico

Testes de software são formas de verificar a corretude de softwares. Tais verificações podem ser feitas de forma manual ou automatizada, sendo essa segunda o foco para a presente pesquisa. Em testes automatizados, existe ainda outra subdivisão de testes: testes de unidade, teste de integração e testes de sistema. Assim, testes de unidade verificam pequenas partes do código, testes de integração verificam uma funcionalidade ou transação completa de um sistema e testes de sistema são ainda mais amplos, pois simulam uma sessão de uso do sistema por um usuário real.

O principal benefício de testes de unidade é encontrar bugs ainda na fase de desenvolvimento e antes que o código entre em produção, quando os custos de correção e os prejuízos podem ser maiores. Portanto, se um sistema de software possui bons testes, é mais difícil que os usuários finais sejam surpreendidos com falhas. Além disso, testes de unidade funcionam como uma rede de proteção contra regressões no código, já que modificações podem introduzir erros em funcionalidades previamente corretas [3].

Para garantir eficácia e confiabilidade, os testes de unidade devem observar os princípios FIRST (Fast, Independent, Repeatable, Self-checking, Timely) propostos por Robert C. Martin em Código limpo: Habilidades práticas do Agile Software. Ou seja: devem ser executados rapidamente (fast); a ordem de execução não deve alterar os resultados, um teste não deve configurar condições para o próximo (independent); devem apresentar sempre os mesmos resultados em qualquer ambiente de execução (repeatable); os resultados devem ser facilmente verificáveis, assim, devem ser binários, indicando simplesmente se o teste passou ou não (self-checking); devem

ser escritos em tempo hábil, imediatamente antes do código de produção no qual serão aplicados (timely).

Outra métrica bastante comum é a cobertura, que avalia o quanto do código-fonte é verificado pelos testes automatizados. Existem diversas formas de medir essa cobertura, cada uma oferecendo uma perspectiva distinta sobre a eficácia dos testes. Entre as principais, a cobertura de instruções verifica se cada linha ou instrução do código foi executada ao menos uma vez durante os testes; já a cobertura de ramificações assegura que todas as possíveis decisões lógicas, como os caminhos verdadeiro e falso de estruturas condicionais, foram testadas; por fim, cobertura de funções verifica se todas as funções ou métodos definidos no código foram invocados durante os testes.

III. Trabalhos relacionados

Alguns estudos recentes investigam o papel das diretrizes de contribuição no desenvolvimento de testes de software em projetos de código aberto. No estudo ”What Do Contribution Guidelines Say About Software Testing?” [6] foram analisados 200 projetos populares no GitHub em Python e em JavaScript para verificar como as diretrizes de contribuição abordam testes. O estudo identificou que 78% dos projetos possuem alguma documentação sobre testes, sendo que a maioria descreve como executar testes e os testes unitários, enquanto outros aspectos, como cobertura, uso de mocks e melhores práticas, são menos abordados.

Os resultados sugerem que, embora a execução de testes seja frequentemente documentada, a qualidade dos testes nem sempre é enfatizada nas diretrizes. Tal constatação converge com um dos resultados da presente pesquisa, em que apenas 34% dos projetos analisados possuem diretrizes de contribuição que focam na qualidade dos testes. Tal cenário, portanto, reforça a importancia de avaliar se a inclusão de orientações explícitas sobre qualidade de testes impacta a prática de desenvolvimento em projetos TypeScript.

IV. Metodologia

A primeira parte da pesquisa consistiu em encontrar os projetos a serem analisados. Para isso, foi feita uma busca de projetos hospedados no GitHub usando a ferramenta https://seart-ghs.si.usi.ch/ filtrando projetos que usam a linguagem TypeScript, que o último commit foi realizado no último ano em relação à data de coleta (01/04/2024 e 01/04/2025), que possuem pelo menos cem colaboradores e que possuem diretrizes de contribuição. Os resultados foram ordenados de forma descendente em relação ao número de estrelas. O objetivo com esses filtros foi selecionar projetos populares, bem avaliados, atualizados e com uma linguagem de programação popular no cenário de testes de software. Ao final desta etapa, foram filtrados 50 repositórios.

Na segunda etapa, foi feita uma coleta manual dos trechos das guidelines que orientam sobre a elaboração de testes com qualidade nos projetos selecionados. Ao mesmo tempo, os projetos foram classificados entre os que possuem guidelines sobre elaboração de testes com qualidade e os que não possuem. O resultado desta etapa foi estruturado na tabela presente no Apêndice I, que possui o link para cada repositório analisado, a classificação e as guidelines extraídas. Para facilitar as análises, os trechos de guidelines foram traduzidos e padronizados.

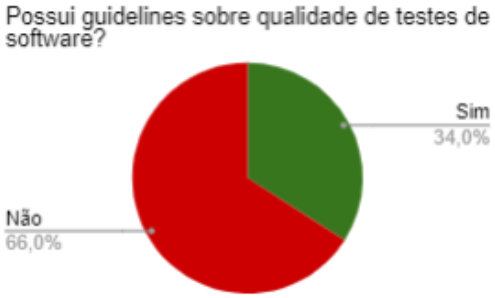

Fig. 1. Classificação dos repositórios em relação a possuir guidelines sobre qualidade de testes

A terceira etapa consistiu em identificar se os repositórios seguem as próprias guidelines sobre elaboração de testes com qualidade. Para isso foi necessário estabelecer diferentes metodologias de análise, conforme detalhado na seção V. Resultados. Por fim, foi realizada a análise dos resultados obtidos nas etapas anteriores e elaboração das conclusões.

V. Resultados

A. Classificação se repositório possui guidelines sobre testes com qualidade

O primeiro resultado obtido na pesquisa foi por meio da segunda etapa descrita na metodologia. Após realizar a coleta de trechos de guidelines que especificam sobre qualidade de testes, foi constatado que apenas 17 dos projetos analisados possuem guidelines que especificam sobre como escrever testes com qualidade, conforme ilustra o gráfico na Figura 1.

Ainda na segunda etapa, foi elaborada uma tabela, apresentada no Apêndice I, que especifica quais trechos foram extraídos das guidelines de cada projeto. O commit possui todas as tabelas elaboradas durante a pesquisa, entre elas, tabela “overview_guideline_repos”, que mostra quantos e quais repositórios analisados possuem cada guideline encontrada. O principal resultado obtido ao elaborar a tabela “overview_guideline_repos” foi que as guidelines “Testes de unidade devem ser criados para funções e elementos que podem ser testados isoladamente.” e “Ao criar testes, use testes já existentes como base” foram as que tiveram mais ocorrências, sendo, respectivamente, em 5 e 4 repositórios.

B. Análise se cada repositório segue as próprias guidelines

1) ***Angular****:* Na terceira etapa, o primeiro repositório analisado foi o Angular, que possui guideline orientando que os testes idealmente devem ser lidos como uma frase, preferencialmente no formato ”it should”. Esse tipo de guideline permitiu uma metodologia automatizada de busca textual. Para isso, foi feita uma busca no VS Code pelo termo ”it(’” filtrando

por arquivos que tenham a extensão ”spec” para identificar todos os testes e, em seguida, foi feita outra busca pelo termo ”it(’should” filtrando por arquivos com a extensão ”spec” para encontrar os testes que seguem a boa prática especificada. O resultado encontrado foi que 72% dos testes segue o formato ’it should’ . Demais testes, apesar de não seguirem a estrutura ”should...”, em geral, podem ser lidos como uma frase. Dessa forma, portanto, o repositório foi classificado como ”Segue as próprias diretrizes”.

Outros dois repositórios analisados possuem guidelines que exigiram uma análise manual, assim como também foi necessário definir uma quantidade de elementos a serem analisados devido à grande quantidade de testes.

2) ***Ant-design****:* O projeto ”ant-design” tem como guideline: ”Escreva testes com base no comportamento, não na implementação”. Nesse caso, a metodologia usada foi a seleção de 50 testes para análise e classificação manual em ”baseado no comportamento” ou ”baseado na implementação”, tendo como critérios: ”não depender de variáveis internas”, ”não acessar métodos internos” e ”testar um resultado observável” para o teste ser classificado como ”baseado no comportamento”. Além disso, foram selecionados 50 testes para análise.

O teste a seguir é um exemplo que foi classificado como ”baseado no comportamento”, já que não depende de variáveis internas nem de detalhes de implementação e testa o resultado observável.

Exemplo de teste baseado no comportamento

```
it('should capitalize the first character of a string',
↪  () => {
  expect(capitalize('antd')).toBe('Antd');
  expect(capitalize('Antd')).toBe('Antd');
  expect(capitalize(' antd')).toBe(' antd');
  expect(capitalize('')).toBe('');
});
```

Já o teste a seguir é um exemplo que foi classificado como ”baseado na implementação” por depender de propriedades internas.

Exemplo de teste baseado na implementação

```
it('getScroll window', () => {
  const scrollToSpy = jest.spyOn(window,
  ↪  'scrollTo').mockImplementation((x, y) => {
    window.pageXOffset = x;
    window.pageYOffset = y;
  });
  window.scrollTo(0, 400);
  expect(getScroll(window)).toBe(400);
  scrollToSpy.mockRestore();
});
```

Cada um dos 50 testes analisados e as respectivas justificativas de classificação podem ser visualizados na tabela ”ant-design”, disponível no link do Apêndice II. O resultado da análise desse repositório foi que 84% dos testes analisados seguem a guideline ”Escreva testes com base no comportamento, não na implementação” e, portanto, o repositório foi classificado como ”Segue as próprias diretrizes.

3) ***Astro****:* O outro repositório que exigiu uma análise manual foi o Astro, que possui como guideline ”Ao criar testes, use testes já existentes como base” e ”Ao criar testes que reutilizam recursos com diferentes configurações, defina valores únicos para caminhos e builds para evitar que resultados sejam compartilhados ou armazenados em cache entre as execuções”. Nesse caso, a metodologia adotada foi a análise de commits identificados com ”feat” ou com ”fix” na mensagem e que modificasse algum arquivo de testes. Commits cuja alteração no arquivo de testes era apenas refatoração foram ignorados. 50 commits foram considerados para classificação.

Este commit é um exemplo que foi classificado como ”Segue as diretrizes”, já que o conjunto de testes adicionado segue a mesma estrutura dos conjuntos de testes já existentes, não houve duplicação considerável de código já existente e foram incluidas funções que podem ser reutilizadas em testes futuros. Enquanto este commit é um exemplo que foi classificado como ”Não segue as diretrizes”, já que o teste adicionado gera duplicação de código em relação a outros testes, como o teste ’renders custom 500’.

Cada um dos 50 commits analisados e as respectivas justificativas de classificação podem ser visualizados na tabela ”astro”, disponível no link do Apêndice II. O resultado da análise desse repositório foi que 94% dos commits seguem as diretrizes de contribuição analisadas e, portanto, o repositório foi classificado como ”Segue as próprias diretrizes”.

4) ***Code-Server****:* O quarto repositório analisado foi o Code-server, cuja guideline extraída foi “Testes de unidade devem ser criados para funções e elementos que podem ser testados isoladamente”. Para esse caso, foi escolhida uma metodologia que mistura a análise manual com processo automatizado. A parte automatizada foi executar as suítes de testes com o objetivo de analisar o tempo de execução a fim de identificar indícios de integrações nos testes de unidade caso os tempos de execução fossem altos. Após executar os testes, os tempos de execução foram longos para testes de unidade em todos os suítes do projeto, de modo que o menor tempo foi 7.908 segundos, com o arquivo unit/comon/http.test.js e o maior foi 150.505 segundos com o arquivo unit/comon/util.test.ts . Entretanto, o tempo de execução é apenas um indício de que pode haver integrações nos testes de unidade. Para completar a análise, foi feita a parte manual, que consistiu em analisar trechos de códigos de testes de unidade. Como resultado, foi observado que os testes não possuíam mocks suficientes e realizavam operações de I/O. Dessa forma, o repositório foi classificado como “Não segue as próprias diretrizes”.

O arquivo health.test.ts, apesar de estar localizado na pasta test/unit, não segue a diretriz ”Testes de unidade devem ser criados para funções e elementos que podem ser testados isoladamente”, fundamental para que um teste possa ser classificado como teste de unidade. Alguns trechos que mostram a falta de isolamento nos testes são:

As linhas abaixo executam chamadas de rede reais (HTTP e WebSocket), tornando o teste dependente da pilha de rede e da latência da comunicação.

```
const resp = await codeServer.fetch("/healthz")
// ...
const ws = codeServer.ws("/healthz")
```

Neste segundo exemplo, a dependência codeServer.ws não é mockada, o que força o teste a estabelecer uma conexão real, tornando-o lento e não determinístico.

```
 const ws = codeServer.ws("/healthz")
// ...
ws.on("message", (message) => { ... })
```

VI. Conclusões

A partir da amostra de 50 repositórios analisados na presente pesquisa, foi constatado que apenas 34% possui alguma guideline sobre como escrever testes com qualidade. Além disso, os que possuem, possuem poucas. Entre elas, duas se destacaram por estarem presentes em mais repositórios: a guideline "Testes de unidade devem ser criados para funções e elementos que podem ser testados isoladamente" estava em 5 dos 17 repositórios analisados e a guideline "Ao criar testes, use testes já existentes como base" estava presente em 4 dos 17 repositórios analisados. Em relação à classificação de o repositório seguir ou não as próprias guidelines, 3 dos 4 repositórios analisados foram classificados como "segue as próprias guidelines" por apresentarem entre 72% e 94% de casos positivos entre os analisados.

Dessa forma, a presente pesquisa resultou em indícios de que guidelines sobre qualidade testes de software contribuem com a qualidade dele. Entretanto, é necessário destacar uma continuidade da pesquisa para resultados mais sólidos sobre o real impacto das guidelines. Para isso, uma etapa importante seria aumentar a quantidade de amostras, especialmente para a parte de classificar se as próprias guidelines são seguidas. Outro processo importante seria fazer um cruzamento de guidelines, ou seja, analisar se um repositório segue guidelines que estão especificadas em outros projetos ou que são boas práticas já conhecidas na engenharia de software. Com isso, o objetivo seria obter dados para analisar se a qualidade dos testes está relacionada à especificação nas diretrizes de contribuição ou se é por conhecimento do desenvolvedor.

VII. Apêndices

Apêndice I. Tabela com a classificação se repositórios possuem ou não guidelines sobre qualidade de testes de software e respectivos trechos de guideline.

Link	**Guideline sobre qualidade de testes?**	**Trechos das guidelines (traduzidos e padronizados)**
https://github.com/angular/angular.git	✓	Use nomes descritivos para os testes do Jasmine. Idealmente, os nomes dos testes devem ser lidos como uma frase, geralmente no formato 'it should...'..
https://github.com/ant-design/ant-design.git	✓	Use o GitHub Actions para realizar verificações de qualidade em issues/PRs e melhorar a eficiência da colaboração por meio de comentários e labels; Use o ESLint e o Prettier para realizar verificações de padrões de código, garantindo qualidade e consistência; Use o Jest e o Testing Library para realizar testes unitários e testes de snapshot, garantindo a correção e a estabilidade do código; Escreva testes com base no comportamento, não na implementação
https://github.com/appwrite/appwrite.git	□	
https://github.com/withastro/astro.git	✓	Ao criar testes, use testes já existentes como base; Ao criar testes que reutilizam recursos com diferentes configurações, defina valores únicos para caminhos e builds para evitar que resultados sejam compartilhados ou armazenados em cache entre as execuções
https://github.com/clash-verge-rev/clash-verge-rev.git	□	
https://github.com/coder/code-server.git	✓	Testes de unidade devem ser criados para funções e elementos que podem ser testados isoladamente
https://github.com/vuejs/core.git	✓	Ao criar testes, use testes já existentes como base; Use a API mínima necessária para garantir estabilidade; Cobertura de testes como ferramenta de análise, não como objetivo
https://github.com/cypress-io/cypress.git	✓	Use Docker para replicar o ambiente de CI localmente, garantindo que os testes que passam na sua máquina também passem no pipeline; Modifique e execute os testes dentro do ambiente Docker para manter isolamento e consistência, evitando que diferenças de configuração local impactem os resultados; Use Docker como ambiente com restrição de desempenho para simular condições limitadas de recursos, facilitando a identificação de problemas de performance
https://github.com/kamranahmedse/developer-roadmap.git	□	
https://github.com/langgenius/dify.git	□	
https://github.com/facebook/docusaurus.git	✓	Teste suas mudanças e descreva o plano de testes no Pull Request; Testes locais são recomendados para mudanças significativas
https://github.com/apache/echarts.git	□	
https://github.com/excalidraw/excalidraw.git	□	

Link	Guideline sobre qualidade de testes?	Trechos das guidelines (traduzidos e padronizados)
https : / / github.com / FuelLabs / fuels - ts.git	☐	
https://github.com/grafana/grafana.git	☐	
https : / / github.com / hoppscotch / hoppscotch.git	☐	
https://github.com/vercel/hyper.git	☐	
https : / / github.com / immich - app / immich.git	☐	
https://github.com/ionic-team/ionic-framework.git	☑	Ao criar testes, use testes já existentes como base
https://github.com/jestjs/jest.git	☑	O código escrito precisa ser testado para garantir que atinja o comportamento desejado. Os testes se dividem em testes unitários ou testes de integração.; Testes de integração são necessários quando o teste unitário não é suficiente
https://github.com/laurent22/joplin.git	☑	Quando testes unitários não forem suficientes, forneça um plano de testes manuais detalhado; Para testar se sua funcionalidade está realmente funcionando, inclua pelo menos cinco testes, considerando diferentes tipos de entradas e cenários; Sempre verifique se alterações em uma funcionalidade não quebram outras partes relacionadas da aplicação
https : / / github.com / lobehub / lobe - chat.git	☐	
https://github.com/mui/material-ui.git	☑	Use renderizadores e matchers; Se o seu teste precisar de vários componentes da biblioteca, crie um novo teste de integração - Quando um teste depende excessivamente de identificadores ou estilos, prefira criar um componente específico em testes de regressão em vez de manter essa lógica no teste unitário - Se for necessário disparar e combinar muitos eventos do DOM, prefira usar testes de ponta a ponta (end-to-end); Garanta que seus testes falhem ao registrar chamadas inesperadas de 'console.error' ou 'console.warn', e use expects explícitos de console em testes de regressão para manter a legibilidade e a confiabilidade
https : / / github.com / mermaid - js / mermaid.git	☑	Ao adicionar uma nova funcionalidade, é necessário criar testes; dependendo do tamanho da funcionalidade, pode ser preciso incluir testes de integração; Testes unitários verificam uma única função ou módulo, sendo os mais fáceis de escrever e os mais rápidos de executar. Eles são obrigatórios para todo o código, exceto para os renderizadores, que são testados com testes de integração
https://github.com/n8n-io/n8n.git	☑	Pull requests devem incluir testes unitários, testes de fluxo para nós e testes de interface quando aplicável
https://github.com/nestjs/nest.git	☐	
https://github.com/ChatGPTNextWeb/NextChat.git	☐	
https://github.com/nocodb/nocodb.git	☐	
https://github.com/nuxt/nuxt.git	☐	

Link	Guideline sobre qualidade de testes?	Trechos das guidelines (traduzidos e padronizados)
https://github.com/pixijs/pixijs.git	☐	
https : / / github.com / automician / playright.git	✓	Os testes devem ser herméticos, sem depender de serviços externos
https://github.com/prisma/prisma.git	☐	
https : / / github.com / puppeteer / puppeteer.git	✓	Toda funcionalidade deve ter testes correspondentes; Todo evento ou método de API pública deve ter testes correspondentes; Os testes não devem depender de serviços externos; Os testes devem funcionar nos diversos SOs
https://github.com/TanStack/query.git	☐	
https://github.com/slab/quill.git	☐	
https : / / github.com / infiniflow / ragflow.git	☐	
https : / / github.com / streamich / react - use.git	☐	
https : / / github.com / gothinkster / realworld.git	☐	
https://github.com/reduxjs/redux.git	✓	Ao criar testes, use testes existentes como base; Testes unitários e de integração conforme necessário; Incluir pelo menos cinco testes por funcionalidade; Garantir que alterações não quebrem partes relacionadas.
https : / / github.com / antiwork / shortest.git	✓	Adicionar stories genéricas; Verificar existência de stories antes de novas; Adicionar stories ao modificar renderizadores.
https : / / github.com / socketio / socket.io.git	☐	
https : / / github.com / storybookjs / storybook.git	✓	Adicione uma story ou um conjunto de stories genéricas à suíte de testes para ajudar a validar seu trabalho; Verifique se já existe um conjunto completo de stories para funcionalidades essenciais antes de criar novas; Ao modificar um renderizador específico (React, Vue 3, etc.), adicione suas stories no diretório 'template/stories' correspondente
https://github.com/strapi/strapi.git	☐	
https : / / github.com / supabase / supabase.git	☐	
https : / / github.com / tailwindlabs / tailwindcss.git	☐	
https : / / github.com / microsoft / TypeScript.git	☐	
https://github.com/shadcn-ui/ui.git	☐	
https://github.com/vitejs/vite.git	☐	
https://github.com/vuejs/vue.git	☐	

Link	**Guideline sobre qualidade de testes?**	**Trechos das guidelines (traduzidos e padronizados)**
https://github.com/pmndrs/zustand.git	✓	Escreva testes que reflitam o uso real do software; Use Mock Service Worker (MSW) para simular requisições de rede sem alterar a lógica da aplicação; Isole estados entre testes para evitar vazamentos; Reutilize código nos testes para evitar duplicação;Simule interações reais do usuário com 'userEvent'; Use 'act()' para garantir que atualizações de estado sejam processadas corretamente no React

Apêndice II. Todas as tabelas elaboradas

REFERENCES

[1] MARTIN, Robert C. Código limpo: Habilidades práticas do Agile Software. Rio de Janeiro: Editora Alta Books, 2009. E-book. p.166. ISBN 9788550816043

[2] VALENTE, Marco Túlio de Oliveira. Compreensão, Manutenção e Evolução de Software. [s.d.]. Cap. 1 – Introdução. Disponível em: https://docs.google.com/document/d/e/2PACX-1vQXQRlC1pNEZbU4gCbrm5SzC9KdOTtukn7o24PdV1zo2cVMZkmYlwW_gex3pV77_mk3qP9At2Nogp-5/pub

[3] VALENTE, Marco Túlio de Oliveira. Engenharia de software moderna: princípios e práticas para desenvolvimento de software com produtividade. Belo Horizonte: [s.n.], 2020. p. 249–280. Disponível em: https://engsoftmoderna.info/.

[4] NAIK, Kshirasagar; TRIPATHY, Priyadarshi. Software Testing and Quality Assurance: Theory and Practice. Hoboken: Wiley, 2006

[5] HAMILL, Paul. Unit Test Frameworks: Tools for High-Quality Software Development. 1. ed. Sebastopol: O'Reilly Media, 2004.

[6] FALCUCCI, Bruna; GOMIDE, Felipe; HORA, Andre. What do contribution guidelines say about software testing? Belo Horizonte: Department of Computer Science, UFMG.