**UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ**

**HUGO HENRIQUE FUMERO DE SOUZA**

# UM ESTUDO DE CARACTERIZAÇÃO DE CONTRIBUIDORES E SUAS CONTRIBUIÇÕES DE TESTE EM PROJETOS DE CÓDIGO ABERTO

**CAMPO MOURÃO**

**2023**

**HUGO HENRIQUE FUMERO DE SOUZA**

# UM ESTUDO DE CARACTERIZAÇÃO DE CONTRIBUIDORES E SUAS CONTRIBUIÇÕES DE TESTE EM PROJETOS DE CÓDIGO ABERTO

# A CHARACTERIZATION STUDY OF CONTRIBUTORS AND THEIR TESTING CONTRIBUTIONS IN OPEN SOURCE PROJECTS

Dissertação apresentada como requisito para obtenção do título de Mestre em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação - Campo Mourão da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Igor Scaliante Wiese

Coorientador: Prof.. Dr. Reginaldo Ré

**CAMPO MOURÃO**

**2023**

**Ministério da Educação**
**Universidade Tecnológica Federal do Paraná**
**Campus Campo Mourão**

HUGO HENRIQUE FUMERO DE SOUZA

# UM ESTUDO DE CARACTERIZAÇÃO DE CONTRIBUIDORES E SUAS CONTRIBUIÇÕES DE TESTE EM PROJETOS DE CÓDIGO ABERTO

Trabalho de pesquisa de mestrado apresentado como requisito para obtenção do título de Mestre Em Ciência Da Computação da Universidade Tecnológica Federal do Paraná (UTFPR). Área de concentração: Metodologia E Técnicas Da Computação.

Data de aprovação: 13 de Novembro de 2023

Dr. Igor Scaliante Wiese, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Eduardo Martins Guerra, Doutorado - Free University Of Bozen-Bolzano

Dr. Marco Aurelio Graciotto Silva, Doutorado - Universidade Tecnológica Federal do Paraná

*À minha esposa, Romi, e ao meu filho, Pedro.*

# AGRADECIMENTOS

Primeiramente, agradeço a Deus, o grande arquiteto do universo, que me proporcionou vida, força, paz, sabedoria e perseverança, fundamentais para a realização deste trabalho e superação dos inúmeros desafios ao longo da minha jornada.

À minha esposa, Romineide, e ao meu filho, Pedro, pilares da minha existência e que estiveram comigo em cada momento, bons ou difíceis, me apoiando incondicionalmente. Sem o amor e apoio de vocês, minhas vitórias não teriam o mesmo significado.

Um especial agradecimento aos meus pais, Welington e Jaqueline, que cativaram em mim o valor da determinação e sempre incentivaram meu crescimento pessoal e profissional.

Aos meus avôs, Moacir, Edis, Leone e Florice, sou eternamente grato pelo legado de vida e pelos incontáveis momentos de carinho e amor. Vocês são as raízes da minha história.

Ao meu orientador, Dr. Igor Wiese, expresso minha profunda gratidão pela confiança, oportunidades, orientação e conhecimento compartilhados. Foi uma honra ter você junto nessa caminhada.

Ao meu coorientador, o Prof. Dr. Reginaldo Ré, por sua inestimável contribuição em forma de orientação, apoio, oportunidades e sabedoria. Sua presença foi fundamental e fez uma diferença significativa ao longo desta jornada, pela qual serei eternamente grato. Um ícone no mundo acadêmico ao qual tenho grande admiração.

Ao Prof. Dr. Igor Steinmacher, é uma honra tê-lo conhecido. Sem dúvida um dos melhores professores que já conheci. Meus sinceros agradecimentos.

Ao meu amigo, Msc. Ricardo Satin, obrigado pelo encorajamento e pela força que me deu ao dar os primeiros passos rumo ao mestrado.

E, por fim, ao Prof. Dr. Eduardo Guerra e Prof. Dr. Marco Aurélio Graciotto Silva, agradeço sinceramente por disponibilizar seu tempo e expertise para avaliar e enriquecer este trabalho como membro da banca examinadora. Suas contribuições são altamente valorizadas.

# RESUMO

Mesmo projetos de software livre possuindo algumas características diferentes de projetos da indústria, o empenho de mantenedores e contribuidores para alcançar um alto nível de qualidade do software é constante. Para tanto, testes estão entre as principais práticas das comunidades.Assim, durante a atividade de desenvolvimento a maioria dos repositórios enfatizam a necessidade de enviar junto com o código desenvolvido, testes que garantem a qualidade e a exatidão do item codificado. No entanto, constatamos que não é tão comum a contribuição de testes durante o desenvolvimento. Buscamos neste trabalho caracterizar os contribuidores de testes em projetos de software livre e verificar o percentual *pull requests* de correções de bugs que contenham também testes automatizados. Para isto, conduzimos um estudo quantitativo – de caracterização dos projetos, contribuidores e suas contribuições – e, também, um estudo qualitativo com contribuidores selecionados a partir da caracterização. Identificamos que apenas 7,87% das PRs de correções de defeitos possuem algum tipo de evolução de cenários de testes. Além disto, foi possível identificar cinco perfis de contribuidores de teste: Contribuidor Movimentação, Atualizador de Cenários, Especialista em Bibliotecas, Analista Comentário e o estratégico Contribuidor de Evolução. Para chegar a essas conclusões, desenvolvemos a ferramenta PRAnalyzer, que permite analisar o código presente em PRs, extraindo informações cruciais, como a quantidade de *asserts*, *imports*, linhas em branco e código contribuído. Além disso, como um recurso valioso para a comunidade acadêmica, disponibilizamos um dataset abrangente, contendo informações de mais de 400.000 PRs e suas características.

**Palavras-chaves:** Teste de software, qualidade de software, projetos de software livre

# ABSTRACT

Even though open-source software projects exhibit some distinct characteristics from industry projects, the dedication of maintainers and contributors to achieving a high level of software quality remains constant. To this end, testing is among the primary practices in these communities. During the development activity, most repositories emphasize the need to submit, along with the developed code, tests that ensure the quality and accuracy of the coded item. We observed that the contribution of tests during development is not as common. In this work, we aim to characterize test contributors in open-source software projects and determine the percentage of pull requests containing bug fixes that also include automated tests. For this purpose, we conducted a quantitative study - characterizing projects, contributors, and their contributions - and a qualitative study with selected contributors identified from the characterization. We found that only 7.87% of bug fix PRs contain some form of test scenario improvement. Furthermore, we identified five test contributor profiles: Contributor in Motion, Scenario Updater, Library Specialist, Comment Analyst and the strategic Evolution Contributor. To reach these conclusions, we developed the PRAnalyzer tool, which allows for the analysis of code in PRs, extracting critical information such as the quantity of asserts, imports, blank lines, and contributed code. Additionally, as a valuable resource for the academic community, we provide a comprehensive dataset containing information on over 400,000 PRs and their characteristics.

**Keywords:** Software testing, software quality, open-source software projects

# LISTA DE ILUSTRAÇÕES

# LISTA DE TABELAS

# LIST OF SOURCE CODE

# SUMÁRIO

# 1 INTRODUÇÃO

A qualidade do software é uma meta importante para todos os projetos, sejam eles de código fechado ou de código aberto (KUMAR; MISHRA, 2016; Del Bianco et al., 2010; HOODA; CHHILLAR, 2015). O teste de software é uma prática essencial para melhorar a confiabilidade e a robustez de sistemas, enfatizado nos últimos anos pelo TDD e pelas metodologias ágeis. No contexto do desenvolvimento colaborativo de software, compreender o uso de testes em contribuições para projetos de código aberto é essencial para avaliar a qualidade e a efetividade das próprias práticas de teste adotadas. Em verdade, existem peculiaridades em projetos de código aberto (OSS) que os diferenciam de projetos tradicionais (I et al., 2008; NGUYEN-DUC et al., 2015). Muitos projetos de código aberto não possuem um processo de software estritamente definido pela comunidade para garantir a condução de atividades que melhoram a qualidade do software, dentre elas o teste de software (THOMPSON, 2017; ABERDOUR, 2007; BARHAM, 2013; KHANJANI; SULAIMAN, 2011; RIGBY et al., 2012; MORASCA et al., 2011). Além disso, diferentes comunidades adotam diferentes práticas. Isso afeta contribuidores que contribuem em diferentes projetos e também os próprios projetos, que recebem contribuições advindas de diferentes contribuidores.

Justamente buscando aumentar a qualidade de software por meio de evolução no uso de boas práticas, grande parte das contribuições para projetos de código aberto hospedados no GitHub[1] atualmente são feitas enfatizando teste de software e revisão de código. Inclusive, a presença de testes em contribuições — frequentemente realizadas por meio de *Pull Requests* (PRs) (GOUSIOS et al., 2014) – é geralmente verificada durante a revisão de código para aceitação da contribuição e aumenta as chances de que o PR seja aceito (PHAM et al., 2013). Além disso, algumas comunidades sequer aceitam contribuições que não acompanhem testes (ALAMI et al., 2020).

Empregar testes eficazes é fundamental para garantir a qualidade do software de projetos de código aberto (MORASCA et al., 2011; ABERDOUR, 2007; ALAMI et al., 2018; ABDOU et al., 2012). Estudos demonstraram que o teste durante a fase de desenvolvimento pode reduzir significativamente o número de defeitos encontrados na produção, melhorar a estabilidade do sistema e aumentar a confiança do usuário final (KARG et al., 2011). Além disso, o teste auxilia na manutenção e na melhoria contínua do software, permitindo que os desenvolvedores encontrem problemas no início do ciclo de vida do projeto e reduzam o esforço necessário para corrigi-los (KARG et al., 2011).

Embora o teste de software seja considerado uma atividade importante para a qualidade dos projetos de código aberto, esses projetos apresentam desafios específicos para realizar essa atividade e integrá-la ao seu processo de desenvolvimento (MICHLMAYR et al., 2005; KHANJANI; SULAIMAN, 2011). Vários estudos mostraram que a adoção de testes entre projetos de código aberto varia, e a qualidade dos testes implementados também pode ser inconsistente. Por exemplo, um estudo feito com base em 50 mil projetos de código aberto mostrou que apenas 42.66% possuíam

[1] <https://github.com/>

testes implementados. Dessas implementações, apenas 10% possuíam mais de 100 casos de testes (KOCHHAR et al., 2013a). Além disso, a qualidade do teste pode variar consideravelmente nos projetos de código aberto (TERRAGNI et al., 2020). Em outros casos, fora observado que cerca de 33% das PRs possuem algum tipo de contribuição de teste e que 4% das PRs continham apenas código de teste (GOUSIOS et al., 2014).

Além da importância de testar melhorias, a execução de testes em PRs que corrigem bugs ajuda a garantir que a correção funcione e principalmente que não introduza novos bugs ao software (BöHMER, 2012). A principal maneira de fazer testes em projetos de código aberto é utilizar testes automatizados (RAFI et al., 2012). Essa técnica pode executar diversos testes do código contribuído para verificar se está em conformidade com as regras de negócio existentes (KARHU et al., 2009). Outro método é executar de forma manual, passando cenário a cenário tentando identificar falhas no software. Independente de qual método é utilizado, é importante que a PR seja testada antes de ser mesclada ao código principal. Isso garantirá que o novo código esteja em boas condições e não irá introduzir novos defeitos ao sistema.

## 1.1 Objetivos

Portanto, este artigo tem como **objetivo geral** entender o quão frequente é a contribuição de teste em PRs resultantes de correções de defeitos. Para alcançar de maneira satisfatório nosso objetivo geral, pretendemos alcançar os seguintes **objetivos específicos**:

i) Conduzir um estudo quantitativo com objetivo de selecionar repositórios adequados, e caracterizar contribuições e contribuidores do ponto de vista de teste;
ii) Identificar ou desenvolver métodos capazes de analisar a existência de testes em PRs;
iii) Verificar a existência de perfis de contribuidores de código de teste;

## 1.2 Contribuições

A compreensão da frequência de contribuição de testes em solicitações de PRs resultantes de correções de bugs é de extrema importância para a comunidade de desenvolvimento de software (KHANJANI; SULAIMAN, 2011; ZHAO; ELBAUM, 2003). A seguir, são apresentadas justificativas para a execução desse estudo.

**Identificação de padrões de contribuição.** A análise da frequência de contribuição de testes em PRs de correções de bugs pode revelar padrões interessantes de participação dos desenvolvedores. Identificar se determinados grupos de desenvolvedores são mais propensos a contribuir com testes, se a contribuição de testes varia em relação ao tamanho do projeto ou se existem tendências relacionadas a linguagens de programação específicas. Essas informações são valiosas para entender a cultura de teste em diferentes comunidades de desenvolvimento de software e podem ajudar a incentivar a participação de mais desenvolvedores nos esforços de teste.

**Avaliação da maturidade dos processos de desenvolvimento.** A frequência de contribuição de testes em PRs de correções de bugs também pode ser um indicador da maturidade dos processos de desenvolvimento de software. Quanto mais frequente for a contribuição de testes, maior é a probabilidade de o projeto possuir práticas de desenvolvimento bem estabelecidas, focadas na qualidade e na prevenção de defeitos. Por outro lado, uma baixa frequência pode indicar a necessidade de melhorias nos processos de teste e na conscientização sobre a importância dos testes.

**Benchmarking e comparação de projetos.** Compreender a frequência de contribuição de testes em PRs relacionadas a correções de bugs permite comparar diferentes projetos de código aberto e estabelecer benchmarks de referência. Isso pode incentivar uma competição saudável entre projetos para aumentar a qualidade dos testes e atrair mais contribuidores interessados em garantir a confiabilidade do software.

## 1.3 Organização da dissertação

O restante deste projeto de pesquisa está organizado de tal forma que: no Capitulo 2 será apresentada toda a fundamentação teórica necessária para entendimento do estudo e os estudos co-relacionados ; no Capitulo 3 serão apresentados todos as etapas e métodos necessários para atingirmos os objetivos da pesquisa e os resultados obtidos na pesquisa; no Capitulo 4 iremos apresentar as conclusões finais do trabalhos e as ameaças a validade identificadas ao longo da pesquisa.

# 2 REFERENCIAL TEÓRICO

Para chegarmos a um bom entendimento do método e dos resultados obtidos nesta pesquisa, foi necessário um profundo entendimento nas áreas de: teste de software, mineração de repositórios e contribuição em projetos OSS.

Primeiramente, é apresentada, na seção 2.1, a definição de teste de software, bem como seus diferentes métodos e execuções para identificação de defeitos e garantia da qualidade. Devido ao objetivo do estudo, um foco especial é dado na subseção relacionada ao teste de regressão. A Seção 2.2 introduz os conceitos gerais sobre mineração de dados de repositórios e também aborda uma das principais plataformas de armazenamento de repositórios públicos, o GitHub. Nesta seção, também são introduzidos os conceitos de *regex*, visto que essa técnica é utilizada na extração de características ao longo do estudo. A Seção 2.3 apresenta um panorama geral sobre contribuição em projetos OSS e os fatores que motivam ou desmotivam os contribuidores a colaborar com os repositórios. Por fim, na Seção 2.4, são destacados os trabalhos que, de alguma forma, se relacionam com o tema desta dissertação.

## 2.1 Teste de Software

Na atualidade, o desenvolvimento de software é uma engrenagem central no mundo moderno, uma atividade complexa e multidisciplinar que exige rigor e precisão para garantir a entrega de produtos estáveis e confiáveis. Neste cenário, o teste de software surge como um componente vital no ciclo de desenvolvimento, assegurando que os softwares atendam às especificações técnicas e aos requisitos de negócios delineados (EICKELMANN; RICHARDSON, 1996; ALENEZI et al., 2016; ABERDOUR, 2007).

Um estudo conduzido pelo *Consortium for Information & Software Quality* mostrou que, por conta de defeitos em software, custam a economia americana certa de US$ 2,08 trilhões anualmente (KRASNER, 2022). Embora seja uma etapa crítica, o teste de software é muitas vezes percebido como um componente caro no ciclo de vida do desenvolvimento de software. Estima-se que os custos relacionados aos testes podem consumir entre 20% e 50% do custo total de desenvolvimento (ANVIK et al., 2006; KOCHHAR et al., 2013b). Este custo pode ser atribuído não apenas à necessidade de ferramentas e ambientes de teste específicos, mas também ao tempo e expertise necessários para criar, executar e analisar os testes (KOCHHAR et al., 2013b).

O teste de software pode ser conceitualizado como um processo estruturado, que envolve a execução do programa sob análise com o intuito de encontrar defeitos e verificar se o sistema atende às expectativas e requisitos para os quais foi concebido (MYERS, 1979; MORI, 2020; GAROUSI; MäNTYLä, 2016). Adicionalmente, serve para validar a confiabilidade, a segurança e a performance do software, atributos intrinsecamente relacionados à sua qualidade (GAROUSI; MäNTYLä, 2016).

Esta etapa, portanto, é intrínseca à engenharia de software e serve como ferramenta de validação, verificação e, sobretudo, de garantia de qualidade (KOCHHAR et al., 2013b; ALENEZI et al., 2016; ABERDOUR, 2007).

Para compreender integralmente o processo de teste de software, é indispensável estar familiarizado com alguns conceitos-chave delineados no glossário padrão do IEEE (IEEE…, 1990). As definições são as seguintes:

- Defeito: qualquer imperfeição ou inconsistência no produto do software ou em seu processo; Algo que está implementado de forma errada.
- Erro: Se refere a ocorrência de um defeito causando ou não uma falha no produto final;
- Falha: Quebra no fluxo de uma aplicação decorrente de um erro; descreve uma operação de software que não atende às expectativas do usuário.
- Teste: este é o procedimento de avaliação do software para identificar disparidades entre as condições existentes e as desejadas, e para avaliar as funcionalidades do software. O objetivo primordial do teste é descobrir falhas potenciais.
- Depuração: é a fase subsequente ao teste, na qual os erros identificados são localizados e corrigidos, visando o aprimoramento contínuo do software.

Ao nos debruçarmos sobre a dinâmica dos testes de software, é imprescindível delinearmos suas entradas, métodos e saídas. As entradas referem-se aos dados e condições sob os quais o sistema será testado, geralmente derivados dos requisitos e especificações técnicas documentados anteriormente no processo de desenvolvimento (MYERS, 1979). Os métodos, por sua vez, são as estratégias e técnicas aplicadas para conduzir os testes, os quais podem variar de testes de unidade, passando por testes de integração até testes de sistema, cada um com sua abordagem e foco específico (BIASI; BECKER, 2006). Quanto às saídas, estas são os resultados obtidos após a execução dos testes, que podem indicar o sucesso ou a falha do software em atender aos critérios estabelecidos, fornecendo um panorama valioso sobre sua estabilidade e confiabilidade.

Os benefícios da implementação efetiva de testes de software são numerosos, indo além da detecção e correção de defeitos. Um produto bem testado transmite confiança aos *stakeholders*, reduzindo riscos e evitando custos subsequentes com manutenção e correção de *bugs* no ambiente de produção (GAROUSI; MäNTYLä, 2016). Adicionalmente, permite uma melhor adaptabilidade às mudanças, uma vez que facilita a identificação de áreas impactadas por alterações no código, conferindo uma maior segurança no processo evolutivo do software. Em última análise, os testes fortalecem a reputação da equipe de desenvolvimento e da organização, garantindo a entrega de um produto de alta qualidade que atende às expectativas dos usuários finais.

As principais fases de teste de software são (SOMMERVILLE, 2011):

- Teste de unidade ou Teste de módulo;
- Teste de Integração;
- Teste de Sistema;
- Teste de Aceitação;

- Teste de Regressão.

### 2.1.1 Testes de Unidade

O teste de unidade é uma fase crucial no processo de desenvolvimento de software, onde os componentes individuais de um software são testados isoladamente (RUNESON, 2006; DAKA; FRASER, 2014). O principal objetivo deste teste é validar que cada unidade do software funciona como projetado (RUNESON, 2006). Uma "unidade"é o menor parte testável de um software e pode ser uma função, um método, um procedimento, uma interface, entre outros.

Na estrutura de um teste de unidade, os desenvolvedores normalmente utilizam frameworks específicos para construir e gerir casos de teste para cada unidade de código. Esses frameworks podem incluir ferramentas e bibliotecas que facilitam a criação, a execução e a verificação dos testes. Entre os frameworks mais conhecidos estão o JUnit para Java, o Pytest para Python, e o Mocha para JavaScript (HAMILL, 2004).

Implementar testes de unidade pode trazer uma série de benefícios para o processo de desenvolvimento de software (RUNESON, 2006). Além de facilitar a identificação e correção de erros em estágios iniciais do desenvolvimento, esses testes também promovem um código mais limpo e bem estruturado (DAKA; FRASER, 2014). Isso facilita a manutenção e a extensão do software a longo prazo . Além disso, serve como uma espécie de documentação, auxiliando outros desenvolvedores a entenderem melhor o funcionamento do código (DAKA; FRASER, 2014).

Para maximizar a eficácia dos testes de unidade, é importante seguir algumas boas práticas. Entre elas está escrever testes atômicos, que testam apenas uma funcionalidade por vez, facilitando a identificação de falhas. É essencial também manter os testes atualizados à medida que o código é alterado ou expandido, garantindo que o conjunto de testes continue relevante e útil. Além disso, recomenda-se que os testes sejam escritos de forma clara e bem documentada, para facilitar o entendimento e a manutenção dos mesmos no futuro (RUNESON, 2006).

### 2.1.2 Testes de Integração

Teste de integração é uma fase de testes no processo de desenvolvimento de software que visa verificar a coesão entre as unidades que compõem o sistema . Após o teste de unidade, que confirma o funcionamento correto de cada componente de maneira isolada, o teste de integração entra em cena para garantir que essas unidades funcionem corretamente quando integradas. Ou seja, busca identificar problemas que possam ocorrer nas interações entre os diferentes módulos ou unidades do software (SOMMERVILLE, 2011; LEUNG; WHITE, 1990).

Existem 4 abordagens principais na execução de testes integrados, são elas (GELLER, 1978; PFLEEGER, 2004):

- Integração Big-Bang: neste método, uma vez que os exames dos componentes foram realizados individualmente, o teste de integração é executado de maneira não sequencial, isto é, todos os

componentes do sistema são testados simultaneamente. Contudo, frequentemente isso conduz a um estado caótico, já que várias falhas podem ser identificadas e o avaliador não tem como determinar em qual interseção de componentes o problema surgiu, tornando a identificação do erro uma tarefa árdua.

- Integração Top-Down: trata-se de um método de teste que busca avaliar a união dos módulos de maneira gradativa, no qual o estrato mais elevado da hierarquia, comumente o módulo de controle, é examinado de forma autônoma. Posteriormente, todos os elementos acionados por esse módulo são agregados e verificados como um conjunto único. A consolidação dos módulos ocorre através da hierarquia de controle;
- Integração Bottom-Up: os componentes são unidos e avaliados progressivamente do mais básico para o mais complexo. Ou seja, cada componente na base da estrutura é avaliado isoladamente, progredindo na avaliação de maneira crescente;
- Integração "sanduíche": essa metodologia combina as técnicas Top-Down e Bottom-UP. O sistema é visto como se fossem três estratos, semelhante a um misto: o estrato central é o foco, os estratos superiores aplicam a técnica Top-Down e os estratos inferiores utilizam a técnica Ascendente. Essa metodologia permite iniciar os testes de integração precocemente durante a fase de avaliação.

### 2.1.3 Testes de Sistema

O teste de sistema é uma fase crítica no desenvolvimento de software, garantindo que o produto final atenda precisamente às necessidades e expectativas dos clientes. Essa etapa é fundamentada em uma análise aprofundada dos requisitos especificados anteriormente no processo de desenvolvimento (PRESSMAN, 2021).

Nessa fase não estamos lidando com um único teste, mas com uma série de testes minuciosamente planejados e executados para avaliar cada componente do sistema baseado em computador. O objetivo é certificar-se de que todos os elementos estão integrados de forma harmoniosa e funcionando conforme determinado, criando um sistema coeso e confiável (PRESSMAN, 2021).

### 2.1.4 Testes de Aceitação

O teste de aceitação tem como objetivo avaliar se o produto atende às especificações e requisitos estabelecidos inicialmente pelo cliente ou pelos stakeholders (HAUGSET; HANSSEN, 2008; PRESSMAN, 2021). Essa etapa ocorre após os estágios de testes de sistemas e de integração, e serve como um aval final para garantir que o software está pronto para ser lançado ao mercado. Em muitos casos, os testes de aceitação são realizados em colaboração com os usuários finais, para garantir que as funcionalidades atendem às suas necessidades e expectativas práticas (PRESSMAN, 2021; IEEE..., 1990).

Os testes de aceitação podem ser conduzidos através de várias metodologias, que incluem tanto procedimentos manuais quanto automatizados. Algumas abordagens comuns são o teste de

aceitação do usuário (UAT), onde os usuários finais têm a chance de interagir com o sistema e avaliar suas funcionalidades, e os testes de aceitação de contrato, que focam em verificar se todas as estipulações contratuais foram atendidas (LEUNG; WONG, 1997).

### 2.1.5 Testes de Regressão

O teste de regressão visa garantir que regressões não ocorram, ou seja, que a introdução de novas funcionalidades ou correções não gere novos problemas (HERZIG et al., 2015). A identificação precoce de falhas e inconsistências permite não apenas uma economia de recursos, mas também contribui para a construção de um produto mais robusto e confiável, o que certamente é uma meta em qualquer projeto de desenvolvimento de software (HERZIG et al., 2015). Testes de regressão surgem como uma ferramenta de primeira linha na manutenção da integridade do código em face de modificações incessantes.

Essa é uma prática frequentemente empregada após a revisão de código e antes da integração de novas funções ao sistema principal, figurando como uma espécie de guarda-chuva de segurança que protege a funcionalidade estabelecida do software.

Há uma vasta diversidade de técnicas de teste de regressão disponíveis (LEUNG; WHITE, 1989). Dentre elas, podemos destacar a reexecução de todos os testes previamente estabelecidos, uma estratégia bastante conservadora e que, apesar de oferecer uma cobertura ampla, pode resultar em altos custos de tempo e recursos. Outra abordagem é a seleção de testes direcionados, uma estratégia que busca otimizar os esforços de teste focando em uma subconjunto de testes mais pertinentes às recentes modificações, promovendo assim, uma economia de recursos sem comprometer significativamente a eficácia da estratégia (ROTHERMEL; HARROLD, 1996).

Um bom processo de teste de regressão não apenas evita o surgimento de novos problemas, mas também contribui para uma evolução mais controlada e segura do software, minimizando riscos e garantindo uma experiência de usuário estável e confiável.

### 2.1.6 Testes Automatizados

A introdução de testes automatizados tem um impacto significativo na motivação dos desenvolvedores (DEAK et al., 2016; SANTOS et al., 2017). Essas práticas promovem a verificação rápida do código desenvolvido. Testes automatizados são uma extensão crítica no panorama dos testes de software, sendo ferramentas fundamentais para garantir a eficiência e a eficácia no ciclo de vida de desenvolvimento de softwares (MALEKZADEH; AINON, 2010). Esses testes consistem na automação do processo de verificação onde *scripts* de testes são criados para validar os recursos e funcionalidades do sistema de forma automatizada, sem necessidade de intervenção humana durante a execução dos testes, o que, por sua vez, reduz o tempo de teste e minimiza a possibilidade de erro humano.

A funcionalidade dos testes automatizados repousa no uso de *frameworks* e ferramentas especializadas que permitem a criação de *scripts* de teste que podem ser executados repetidamente em diferentes versões do software (MALEKZADEH; AINON, 2010). Este é um processo que, quando

bem implementado, promove uma série de benefícios, como a economia de tempo, a realização de testes mais complexos e a facilitação de testes de integração contínua, criando um ambiente mais dinâmico e adaptativo no processo de desenvolvimento de software.

Sua importância é acentuada quando observamos o panorama competitivo atual de desenvolvimento de software, onde a rapidez na entrega e a garantia de qualidade são imprescindíveis (KARHU et al., 2009). Os testes automatizados facilitam a identificação precoce de defeitos, o que possibilita correções rápidas e, consequentemente, um produto final mais robusto e confiável (DALLAL, 2009; SANTOS et al., 2017). Além disso, são ferramentas cruciais para a manutenção da qualidade em projetos de grande escala, onde a complexidade pode facilmente levar a falhas.

Um aspecto central dos testes automatizados é a sua interação com os testes de regressão . Em desenvolvimento ágil, onde as mudanças são frequentes, os testes de regressão garantem que as novas alterações não introduzam novos erros ou ressuscitem defeitos antigos (RAFI et al., 2012). A automação, neste contexto, permite uma execução rápida e eficiente desses testes a cada nova iteração, garantindo que a base de código existente mantenha sua integridade ao longo do tempo, uma prática que se tornou praticamente um padrão na indústria (DALLAL, 2009).

No entanto, é importante mencionar que a implementação de testes automatizados tem seus custos, principalmente no que tange à escrita e manutenção dos *scripts* de teste ao longo do tempo (BASHIR; BANURI, 2008). Esse investimento inicial pode ser significativo, dada a necessidade de expertise técnica e ferramentas adequadas. Porém, é um investimento que tende a se pagar a longo prazo, através da redução do tempo de teste e da garantia de uma base de código mais estável e confiável (DALLAL, 2009).

Quando voltamos nosso olhar para os projetos OSS, a automação de testes assume um papel ainda mais vital. Nestes projetos, onde as contribuições vêm de uma variedade de desenvolvedores independentes, a automação facilita a validação de contribuições de diferentes fontes, mantendo a integridade do código e garantindo que as novas adições não quebrem funcionalidades existentes. É uma ferramenta que promove a confiança e a colaboração, elementos chave para o sucesso de projetos OSS (MAKADY; WALKER, 2017).

Além disso, vale a pena mencionar que, no contexto de integração contínua e entrega contínua (CI/CD), os testes automatizados são uma parte inerente, possibilitando entregas mais rápidas e seguras. Esta abordagem, que une os testes automatizados com práticas ágeis e de integração contínua, promove um desenvolvimento mais fluido e coerente, estabelecendo um ciclo de feedback contínuo e auxiliando que os produtos sejam desenvolvidos com qualidade e eficiência.

### 2.1.7 Correção de defeitos

Embora a etapa de testes seja essencial, muitos desenvolvedores encontram uma série de desafios que podem desmotivá-los a participar ativamente desse processo crítico. Uma das principais razões por trás dessa relutância pode ser atribuída ao caráter muitas vezes repetitivo e meticuloso da tarefa

de corrigir defeitos, onde a motivação intrínseca pode diminuir ao longo do tempo (BEECHAM et al., 2008).

Uma preocupação central na correção de defeitos é que, sem a implementação de novas assertivas (do inglês, *asserts*) no código, existe a possibilidade do ressurgimento do mesmo bug no futuro (BöHMER, 2012). A inserção de novas assertivas serve como uma camada de segurança adicional, garantindo que, se um defeito reaparecer, ele possa ser prontamente identificado e resolvido. Desse modo, a falta de novas assertivas pode resultar em um ciclo vicioso de defeitos recorrentes, criando um ambiente de trabalho frustrante para os desenvolvedores.

Além disso, a motivação para corrigir bugs pode ser substancialmente reduzida devido à falta de reconhecimento e recompensa (BEECHAM et al., 2008). Em projetos OSS, os contribuintes muitas vezes dedicam seu tempo e esforço voluntariamente, sem compensação financeira (HARS; OU, 2001). Nesse contexto, o reconhecimento por seus pares e a comunidade em geral pode servir como uma forte motivação. Portanto, a falta de feedback e reconhecimento pode rapidamente transformar a correção de *bugs* em uma tarefa desagradável e desmotivadora (HARS; OU, 2001).

A natureza colaborativa dos projetos OSS, no entanto, pode atuar como um incentivo para os desenvolvedores. A oportunidade de colaborar e aprender com outros profissionais talentosos do campo pode oferecer uma motivação substancial (BONACCORSI; ROSSI, 2003). Essa colaboração pode não apenas facilitar a correção de defeitos, mas também ajudar a criar uma base de código mais robusta e eficiente, através da contribuição e do compartilhamento de conhecimento diversificado.

## 2.2 Mineração de Repositórios

Esta seção explica o conceito de repositórios de software de mineração (MSR) e como eles funcionam. Também esclareceremos o que é o GitHub e qual o papel que ele desempenha no cenário atual. Também discutiremos o uso de expressões regulares, uma das formas mais eficientes de extrair informações.

### 2.2.1 Teoria sobre Mineração

Mining Software Repositories (MSR) refere-se à prática de analisar e extrair informações valiosas e úteis de grandes repositórios de dados que mantêm o histórico de desenvolvimento de software, incluindo, mas não limitado a, sistemas de controle de versão, relatórios de bugs, e mais (van der Aalst et al., 2007; ROZINAT et al., 2009). Com a rápida expansão do desenvolvimento de software, o MSR tornou-se uma área de pesquisa importante que apoia a evolução e a manutenção sustentável dos sistemas de software (ROZINAT et al., 2009; CHATURVEDI et al., 2013).

Um dos principais benefícios da MSR é que ela facilita a compreensão dos padrões e tendências predominantes na evolução do software (HASSAN, 2008). Por meio da análise das alterações do código ao longo do tempo, os pesquisadores podem identificar padrões de design, antipatterns e até mesmo prever futuros problemas que podem surgir (CHATURVEDI et al., 2013).

Além disso, é possível explorar a maneira como os desenvolvedores interagem entre si, permitindo a identificação de melhores práticas e áreas onde podem ocorrer fricções.

Além de identificar padrões e antipatterns, o MSR é uma ferramenta poderosa para aprimorar a segurança e a robustez dos softwares. Através da análise histórica dos bugs e das correções realizadas, é possível criar sistemas que são mais resistentes a falhas, promovendo um nível maior de confiabilidade para os usuários finais (HASSAN, 2008). Além disso, a pesquisa na área de MSR pode oferecer insights cruciais para a criação de ferramentas automatizadas que auxiliam os desenvolvedores na identificação e correção de bugs de forma mais eficiente (CHATURVEDI et al., 2013).

Os aplicativos mais comumente usados em MSR são modelagem e previsão de mudanças, detecção de módulos propensos a defeitos, localização de bugs e seus tempos de vida, previsão de dependência e co-mudança, experiência do desenvolvedor, previsão de tipo de mudanças de código, gravidade, atribuição, estimativa de esforço, tarefa automatizada de minerar repositórios e vincular informações entre esses repositórios, etc. Em seu trabalho, (CHATURVEDI et al., 2013) faz um excelente trabalho ao catalogar as principais ferramentas de MSR.

### 2.2.2 Aspectos Práticos

#### Github

O GitHub[1], criado em 2008 por Chris Wanstrath, P. J. Hyett, Tom Preston-Werner e Scott Chacon, é uma plataforma baseada em nuvem que facilita o controle de versões de código e a colaboração para desenvolvedores de software. Utilizando o sistema de controle de versão Git, desenvolvido por Linus Torvalds, o GitHub não apenas facilitou a colaboração em projetos de software, mas também se tornou um repositório para uma vasta gama de projetos, desde simples repositórios de código até grandes projetos empresariais. A essência de sua existência reside em facilitar a colaboração e a administração de projetos de software, proporcionando uma plataforma onde os desenvolvedores podem trabalhar juntos de forma eficiente e organizada (GITHUB, 2023).

A vitalidade do GitHub na popularização de projetos OSS é inegável. Tornou-se um lugar de referência onde os desenvolvedores podem descobrir projetos OSS inovadores e contribuir para eles, bem como compartilhar seus próprios projetos com a comunidade global (GOUSIOS et al., 2014). Esta plataforma tem facilitado a adoção de uma cultura de colaboração e aprendizagem contínua, fornecendo as ferramentas necessárias para manter um fluxo de trabalho eficaz e produtivo (KOCHHAR et al., 2013a; GOUSIOS et al., 2014). PRs, ou Pull Requests, são uma funcionalidade central do GitHub. São propostas de modificações no código que outros desenvolvedores podem revisar e discutir antes de serem eventualmente integradas ao projeto. Eles são uma ferramenta poderosa para fomentar a colaboração e melhorar a qualidade do código.

[1] <https://github.com/>

A mecânica de um PR inicia-se com um desenvolvedor criando uma bifurcação, ou *fork*, do repositório original, onde podem trabalhar em suas melhorias ou correções de forma isolada (GOUSIOS et al., 2014). Após concluir as alterações, o desenvolvedor então submete um PR para o repositório original, indicando as mudanças propostas e iniciando uma discussão com outros colaboradores sobre a validade e eficácia das alterações. Esta natureza colaborativa dos PRs é fundamental para o sucesso dos projetos hospedados no GitHub (BIRD; ZIMMERMANN, 2012).

Dentro deste ecossistema de colaboração, as *issues* ocupam um papel fundamental. Elas funcionam como uma espécie de fórum onde problemas, *bugs*, melhorias e outras discussões relacionadas ao projeto são articuladas e rastreadas. Através das *issues*, a comunidade pode se organizar para endereçar diferentes aspectos do projeto, seja corrigindo erros ou adicionando novas funcionalidades.

Além disso, as *issues* podem ser categorizadas através de *tags*, que ajudam a sinalizar e organizar diferentes tipos de discussões, facilitando a filtragem e a busca por tópicos específicos. As *tags* podem indicar a severidade de um bug, o tipo de solicitação de recurso ou até mesmo o nível de experiência necessário para abordar a tarefa em questão. Essas etiquetas são cruciais para manter o projeto organizado e garantir que os colaboradores possam navegar facilmente através das diversas discussões e tarefas em andamento (KALLIS et al., 2021). No entanto, ocasionalmente, os colaboradores podem ser negligentes ou esquecer de fazer essa vinculação, o que ocasiona a perda de rastreabilidade (ALSHARA et al., 2023; KALLIS et al., 2021).

Essas *tags* nas *issues* assumem uma função de triagem, permitindo que os colaboradores identifiquem rapidamente áreas onde podem contribuir de forma mais eficaz. Além disso, promovem uma distribuição de tarefas mais equitativa e focada, facilitando a colaboração e a comunicação entre os membros da equipe. Assim, elas não apenas ajudam a manter o projeto organizado, mas também funcionam como um guia para os colaboradores, apontando para onde eles podem direcionar seus esforços de forma mais eficaz (ANDAM et al., 2017; HERZIG et al., 2013).

### Regex

Expressões regulares, conhecidas também como *regex* ou *regexp*, são sequências de caracteres que formam um padrão de busca e são utilizadas para realizar operações de busca, substituição e manipulação de textos em strings por meio de padrões definidos. Essa ferramenta poderosa e flexível pode ser empregada em diversas linguagens de programação, bem como em editores de texto e sistemas de bancos de dados, para executar tarefas como validação de formatos de dados, busca de palavras específicas em grandes volumes de texto, entre outras operações, de maneira eficiente e rápida. Um dos pontos fortes das expressões regulares é a sua capacidade de representar conjuntos infinitos de strings através de padrões concisos, o que facilita enormemente o processo de manipulação de texto, permitindo aos programadores e desenvolvedores executarem tarefas complexas de manipulação de texto com relativamente poucas linhas de código.

Ao utilizar expressões regulares de forma consciente, é possível não apenas identificar padrões em textos, mas também substituir, dividir ou recuperar subconjuntos de strings de maneira muito mais ágil e direcionada, otimizando processos e facilitando a resolução de problemas complexos na manipulação e análise de dados textuais.

## 2.3 Contribuições em Projetos OSS

Projetos OSS são iniciativas em que o código-fonte do software está disponível para o público, o que permite a qualquer desenvolvedor acessá-lo, modificá-lo e distribuí-lo (SCACCHI; JENSEN, 2012). Essa transparência não só fomenta a inovação e a colaboração entre os desenvolvedores, mas também promove um ambiente em que a qualidade e a segurança do software podem ser continuamente aprimoradas pela comunidade em geral (BONACCORSI; ROSSI, 2003). A natureza aberta desses projetos convida a uma diversidade de perspectivas e habilidades, o que muitas vezes resulta em soluções mais robustas e criativas (BONACCORSI; ROSSI, 2003).

Alguns exemplos notáveis de projetos OSS que alcançaram grande sucesso incluem o sistema operacional Linux[2], o servidor web Apache[3] e o sistema de gerenciamento de banco de dados MySQL [4]. Esses projetos não apenas dominaram suas respectivas áreas, mas também facilitaram o surgimento de numerosos outros projetos, demonstrando a vitalidade e a capacidade de inovação que a abordagem OSS pode oferecer (LEE et al., 2009).

É aqui que a importância do teste de software em projetos OSS se torna evidente. Dada a natureza aberta e colaborativa desses projetos, a necessidade de manter padrões de qualidade é ainda mais crítica (KOCHHAR et al., 2013b). Testes de software bem planejados e executados garantem que, apesar das contribuições vindas de diversas fontes, o software mantém um alto nível de qualidade, segurança e desempenho (ABERDOUR, 2007).

Além disso, os testes automatizados se destacam como ferramentas indispensáveis em projetos OSS. Dada a frequência de contribuições e alterações no código, ter um conjunto de testes automatizados pode ajudar a garantir que o software não apenas funcione como esperado, mas que novas contribuições não introduzam erros ou quebrem funcionalidades existentes. A automação permite que os testes sejam executados de forma rápida e eficiente, promovendo uma integridade sustentável do projeto a longo prazo.

No contexto dos projetos OSS, os testes de regressão mantêm um papel central na garantia da qualidade do software. Eles garantem que as alterações recentemente integradas não comprometem as funcionalidades existentes, uma tarefa essencial quando múltiplos desenvolvedores estão colaborando e contribuindo de diferentes formas e perspectivas. Assim, esses testes funcionam como uma rede de segurança, permitindo que os desenvolvedores modifiquem e expandam o código com confiança, sabendo que qualquer regressão será rapidamente identificada e corrigida.

[2] https://github.com/torvalds/linux
[3] https://github.com/apache/httpd
[4] https://github.com/mysql/mysql-server

Os testes de regressão, em união com práticas ágeis e integradas, formam uma linha de defesa robusta contra bugs e defeitos em projetos OSS. Este é um processo contínuo e iterativo que promove não apenas a qualidade e a confiabilidade, mas também facilita a inovação contínua, permitindo que os projetos cresçam e evoluam sem perder a integridade (HERZIG et al., 2015).

A gestão de projetos OSS, assim, torna-se um equilíbrio delicado entre inovação e qualidade, onde os testes desempenham um papel central na manutenção da harmonia. Cada contribuição, enquanto traz uma nova perspectiva, também carrega o potencial de introduzir novos bugs ou vulnerabilidades, fazendo dos testes uma parte vital do processo de integração.

Além disso, ao considerarmos a importância do feedback contínuo na manutenção da qualidade em projetos OSS, fica claro que as estratégias de teste precisam ser incorporadas desde o início do ciclo de vida do desenvolvimento. Isso não apenas facilita a identificação e correção de bugs, mas também promove uma cultura de qualidade e responsabilidade, características essenciais para o sucesso a longo prazo de qualquer projeto OSS

Os projetos OSS precisam da força de trabalho de desenvolvedores que, muitas vezes, não recebem retorno financeiro pelas contribuições. Muitos contribuidores sentem-se motivados a contribuir mesmo sem a compensação financeira. É necessário entendermos que a palavra motivação é usada como um termo genérico para definir comportamentos que não são necessariamente o desejo ou vontade de um profissional a executar seu trabalho de forma efetiva e com qualidade (FRANÇA et al., 2014). De maneira a compreender melhor como a motivação afeta o trabalho no desenvolvimento de software, alguns estudos buscaram compreender o significado deste conceito dentro da engenharia de software (FRANÇA; Da Silva, 2009). Podemos compreender a motivação como uma vontade interna de executar seu trabalho de uma forma mais intensa, por longos períodos e com uma ótima qualidade (FRANÇA; Da Silva, 2009). O indivíduo está motivado intrinsecamente quando executa determinada atividade apenas pela satisfação de executá-la, e não por alguma recompensa que a atividade pode trazer. Por outro lado, pode-se definir a motivação como sendo extrínseca quando o indivíduo executa uma atividade com o objetivo de alcançar alguma consequência de sua execução (RYAN; DECI, 2000).

Paralelo a isto, estudos mostram que a atividade de testes no processo de engenharia de software é considerada uma atividade destrutiva, tediosa e uma atividade secundária durante o desenvolvimento (RIANSYAH, 2015). Por conta das definições de motivações e as características das atividades de teste, estudos foram conduzidos para identificar os fatores motivacionais e desmotivações que levam o profissional de teste a executar sua atividade de forma mais adequada, pois segundo Herzberg (HERZBERG et al., 1959) a motivação tem forte influência no trabalho entregue nas empresas.

Por se tratar de investigações de características humanas, os trabalhos encontrados sobre o tema aplicaram questionários ou conduziram entrevistas para identificar os fatores motivacionais e desmotivacionais das atividades de teste. Os trabalhos abordaram três perspectivas diferentes: testadores de software (DEAK et al., 2016; SANTOS et al., 2017), desenvolvedores (DEAK, 2014; BELLER et al., 2015) e futuros profissionais (CAPRETZ et al., 2019; CAPRETZ; WAYCHAL, 2016;

CAPRETZ et al., 2015). Os trabalhos que tratam especificamente de teste de software (DEAK et al., 2016; SANTOS et al., 2017; DEAK, 2014) aplicaram os questionários na indústria, em empresas previamente escolhidas pelos pesquisadores, diferentemente deste artigo, com foco na motivação de contribuidores de projetos OSS.

Além dos diversos fatores identificados na literatura como motivadores, também podem ser encontrados trabalhos que apresentam detalhes da influência das tarefas recorrentes dos desenvolvedores em sua motivação. Por exemplo, a execução de testes manuais desmotiva mais o profissional do que a utilização de testes automatizados (SANTOS et al., 2017; BELLER et al., 2015). Além disso, escrever testes automáticos faz com que o testador se empenhe em aumentar a cobertura de testes e a executar mais vezes os testes criados afim de identificar novos em novas versões do sistema (DEAK et al., 2016; SANTOS et al., 2017). O tempo destas execuções também tem influência na motivação do profissional. Para os testadores, o tempo de execução da suíte completa e o tempo de evolução dos cenários criados influencia na motivação de executar suas atividades (BELLER et al., 2015).

Em contrapartida, existem trabalhos que apontam detalhes das desmotivações. Como já mencionado, é comum os testadores verem suas atividades como de segunda classe, ou seja, não essencial ao desenvolvimento (DEAK et al., 2016; CAPRETZ et al., 2019; BELLER et al., 2015). Além disso, existem desenvolvedores que acreditam que há um menor desenvolvimento de carreira e baixo retorno financeiro (DEAK et al., 2016; CAPRETZ et al., 2019; BELLER et al., 2015).

Os testadores que se motivam entendem que, ao testar uma aplicação, estão contribuindo com a qualidade do software que será liberado e se sentem desafiados a executarem novos cenários. Além disto, eles entendem que há uma variação maior de trabalho e que requer um conhecimento maior no domínio do nicho explorado pelo software. Quando observamos profissionais que desejam iniciar na carreira de teste, as principais motivações estão relacionadas a facilidade de iniciar na atividade e a facilidade da execução do trabalho (CAPRETZ; WAYCHAL, 2016; DEAK, 2014; CAPRETZ et al., 2015).

Deak e colegas (DEAK et al., 2016) buscaram identificar quais eram os fatores motivacionais e os desmotivacionais que influenciavam os testadores nas suas atividades e quais ações as empresas poderiam executar afim de incentiva-los. Para identificar os fatores, eles realizaram uma entrevista guiada por um questionário em 12 empresas, entrevistando 39 pessoas. O nível de reconhecimento, problemas técnicos, pouco tempo para a atividade e atividade tediosa foram os principais fatores que desmotivam os testadores. Já os seguintes fatores foram identificados como motivacionais: atividades desafiadoras, foco na melhoria da qualidade, variedade de trabalho e bom gerenciamento.

## 2.4 Trabalhos Relacionados

Com o objetivo de identificar se a presença ou ausência de casos de testes pode afetar ou não a qualidade do software em termos de quantidade de bugs reportados, (KOCHHAR et al., 2013a) realizou um estudo com base em 20.000 projetos OSS. Este trabalho fez uma relação entre quantidade de linhas

de código, casos de testes e quantidade de bugs reportados. Foi identificado que aproximadamente 12.835 (61%) dos projetos possuem casos de testes, porém, deste número, 48,26% possuem de 1 a 9 casos de testes e 29,36% possuem entre 10 e 49 casos de teste. Além disso, identificou que apenas 17 projetos, com uma média de 2 milhões de linhas de código possuíam mais de 10.000 casos de teste. Quando analisado no âmbito de contribuidores, os autores verificaram que o número de casos de testes tende a aumentar quando há mais desenvolvedores no projeto.

Essa verificação do número de casos de teste em projetos é um desafio significativo. Adicionalmente, cada linguagem de programação possui sua metodologia específica para escrever testes. Diante dessa complexidade, (KOCHHAR et al., 2013b) conduziu uma análise detalhada, vasculhando todos os arquivos do projeto em busca da palavra "test". A cada ocorrência dessa palavra, o autor assinalou a presença de um possível caso de teste.

O trabalho de (BELLER et al., 2015) identificou que 47% dos projetos que fazer uso da biblioteca JUnit realmente possuem casos de testes. Identificaram também que a execução dos testes, em 50% dos casos são rápidas levando em média meio segundo. O estudo também analisou as reações dos contribuidores após uma falha nos testes. Após a falha, mais de 60% dos contribuidores analisaram o código de produção e cerca de 17% analizaram o código de teste. O trabalho chegou na conclusão que certa de 25% do tempo de desenvolvimento é gasto na etapa de desenvolvimentos dos testes.

No trabalho (GOUSIOS et al., 2014), os autores procuraram entender quais fatores afetam a decisão e o tempo para que um PR seja aceito e também os fatores que levam a rejeição através de extração de caraterísticas de PRs . Chegaram a conclusão que 33% dos PRs possuíam modificações em código de teste e apenas 4% dos PRs alteravam exclusivamente código de teste.

(ZHANG et al., 2023) fezeram uma revisão sistemática para levantar os fatores que influenciam nas decisões de aceitação de PRs utilizando projetos em Javascript, Python, Java, Ruby, Go e Scala. Foi identificado que as características do desenvolvedor e do PR são mais importantes do que a característica do projeto. A relação entre o contribuidor e os mantenedores tem total influência na decisão de aceitação. Além disso, PRs que possuem comentários, uso de ferramentas de CI e indicadores de teste são importantes na análise de aceitação. Para chegar em tais resultados, os autores realizaram uma revisão da literatura para consolidar quais fatores tem influencia durante a revisão de um PR.

Nossas buscas por trabalhos relacionados focados em análise de testes em pull request não obteve bons resultados, de maneira que até o presente momento não foram encontrados estudos específicos sobre PRs e testes. Há sim diversos estudos que tem como objetivo analisar PRs, porém com foco nos dados do pull request, como data de criação, usuário responsável, mesclado ou não, entre outros (ZAMPETTI et al., 2019). Um trabalho escrito por (CHATURVEDI et al., 2013) compilou as bibliotecas que realizam algum tipo de análise de pull request até o ano de 2013, ao todo, o trabalho catalogou 73 ferramentas.

Ao analisar ferramentas que avaliavam a qualidade de PRs, (LAMKANFI et al., 2010) investigou a real severidade dos defeitos com base nas informações fornecidas na abertura do PR. Esse mesmo estudo também examinou a presença de código binário nos PRs. Em um estudo separado,

(CHEN et al., 2012) aplicou métodos estatísticos, considerando fatores como total de linhas de código e relevância das classes, para determinar o grau de defeitos em um repositório. Por sua vez, (HINDLE et al., 2008) fez uma análise manual de PRs, diferenciando entre commits que alteram muitos arquivos e commits menores. A conclusão foi que commits maiores tendem a ser mais evolutivos, enquanto os menores são predominantemente corretivos.

## 2.5 Considerações Finais

Nesta seção, introduzimos brevemente os conceitos de teste de software, testes automatizados, correção de defeitos, mineração de repositórios e Github. Elucidamos as distinções entre as principais técnicas de teste de software e estabelecemos os fundamentos para a extração de informações de repositórios de software livre. Essas informações são vitais para apoiar os mantenedores na evolução e manutenção sustentável de softwares livres.

No que se refere à literatura existente, não identificamos estudos que extraíssem informações diretamente associadas ao teste do software a partir do PR contribuído. No contexto de testes de regressão em projetos de software livre, há vários estudos focados na priorização e seleção de cenários de teste, mas não encontramos nenhum que se alinhasse precisamente ao escopo deste trabalho. Em relação à identificação de casos de teste, (BELLER et al., 2015) investigou a correlação entre importações de bibliotecas de testes automatizados e a presença de testes no projeto. Já (KOCHHAR et al., 2013b) explorou a presença da palavra "test"como indicativo de cenários de teste. Esta última abordagem foi adotada em nossa metodologia, detalhada na Seção 3.

# 3 O ESTUDO DE CARACTERIZAÇÃO

Nosso trabalho de caracterização das contribuições está dividido em três grandes etapas: obtenção e tratamento dos dados dos repositórios; estudo de caracterização dos repositórios; estudo de caracterização dos contribuidores. A primeira etapa foi conduzida com vistas ao descobrimento de quais repositórios possuem teste e quais são os contribuidores destes repositórios. A segunda etapa foi feita para a caracterização dos contribuidores e suas contribuições, com o principal objetivo de descobrir quais seriam bons contribuidores a serem levados para a terceira etapa. A terceira etapa consiste na análise qualitativa das contribuições.

Os passos conduzidos e os passos futuros do estudo são ilustrados detalhadamente na Figura 3.1. De maneira geral, existem três conjuntos de passos: a obtenção e tratamento dos dados, que tem por objetivo produzir um conjunto de dados adequado e útil para vários estudos; o estudo de caracterização, cujo objetivo é produzir resultados a respeito de contribuidores de teste e suas contribuições; e, o terceiro é analisar qualitativamente contribuidores e suas contribuições.

## 3.1 Obtenção e Tratamento dos Dados

A obtenção e tratamento dos dados contém passos que visam a dar suporte ao estudo de caracterização por meio de atividades típicas de mineração de dados. Escolhemos a estratégia de usar PRs (do inglês *Pull Requests*) como átomos de trabalho de contribuição, uma vez que são mais adequados para estudos comportamentais em projetos OSS (BERTONCELLO et al., 2020).

### 3.1.1 Escolher Linguagens de Programação

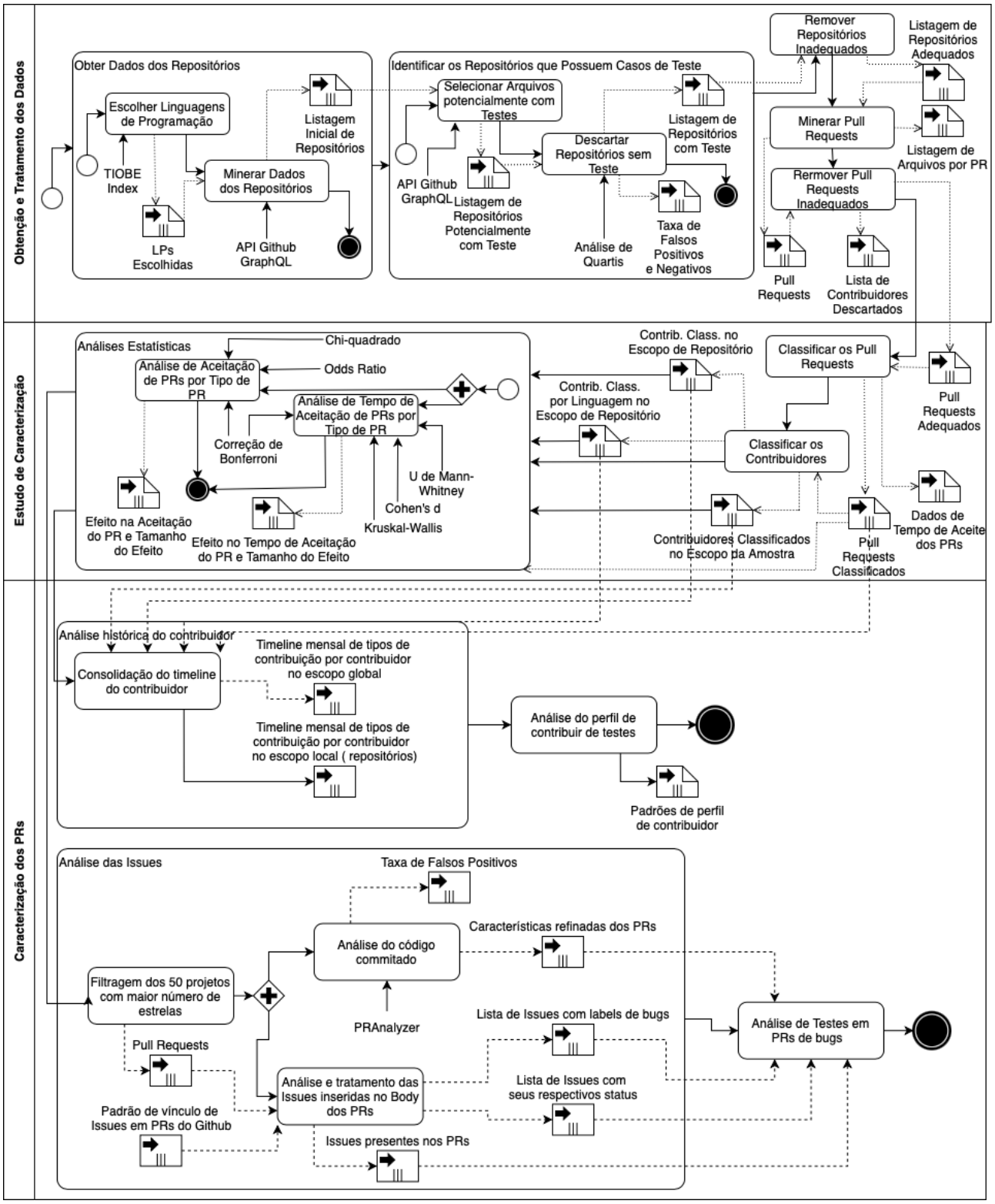
O primeiro passo de nosso trabalho para obter dados dos repositórios no GitHub foi definir quais linguagens principais dos repositórios utilizar. Para isso, analisou-se a popularidade das linguagens de acordo com a pesquisa TIOBE[1], que se baseia na quantidade de resultados obtidos nos mais diversos buscadores considerando a quantidade de profissionais qualificados que usam as linguagens e de cursos disponíveis. Em janeiro de 2022 a pesquisa com desenvolvedores de software mostrou que Python, C, Java, C++, C#, Ruby e JavaScript são as 7 linguagens mais populares em desenvolvimento de software. Por esse motivo, escolhemos tais linguagens como objeto de estudo desta pesquisa.

### 3.1.2 Minerar Dados dos Repositórios

Utilizamos então a API GitHub GraphQL [2] para coletar os dados do GitHub. Esta ferramenta possibilitou consultas agrupadas e mais específicas, diminuindo assim a faixa de repositórios com as características requeridas. Por ser mais moderna e flexível nas consultas, em contrapartida à API REST

[1] <https://www.tiobe.com/tiobe-index/>
[2] <https://docs.github.com/pt/graphql>

**Figura 3.1.** Passos executados na condução do estudo.




do GitHub [3], a API GraphQL se mostrou eficiente com relação ao tempo de obtenção dos dados dos projetos. É importante ressaltar que a coleta considerou apenas a linguagem principal do repositório—já que cada repositório pode hospedar códigos em múltiplas linguagens. Além disso, selecionamos

[3] <https://docs.github.com/en/rest>

apenas projetos não arquivados e com mais de 3.000 estrelas, como forma de minimamente considerar projetos com alguma relevância e visibilidade.

Ao total foram selecionados 5.238 repositórios. A Tabela 3.1 apresenta uma listagem com a quantidade de repositórios coletados para cada linguagem escolhida. Pode-se constatar que grande maioria, 75,45%, é de projetos implementados em Javascript, Python e Java, com destaque para Javascript com 37,08%.

**Tabela 3.1.** Listagem de repositórios recuperados por linguagem de programação e segundo os parâmetros selecionados.

| Linguagem | # Repositorios | % Repositorios |
|---|---|---|
| Geral | 5238 | 100,00 |
| C | 369 | 7,04 |
| C# | 220 | 4,20 |
| C++ | 471 | 8,99 |
| Java | 819 | 15,64 |
| Javascript | 1942 | 37,08 |
| Python | 1191 | 22,74 |
| Ruby | 226 | 4,31 |

### 3.1.3 Selecionar Arquivos Potencialmente com Teste

Utilizando a lista de repositórios coletados na etapa anterior, conduzimos outra busca usando a API GitHub GraphQL para identificar a subpalavra “test” no nome dos arquivos ou em seu conteúdo e armazenamos a quantidade de arquivos encontrados. Este mesmo método foi utilizado por (KOCHHAR et al., 2013b) para identificação de casos teste. Dentre os 5.238 repositórios, identificamos que 4.047 (76,50%) potencialmente apresentam casos de teste por possuírem mais de uma menção à subpalavra “test”.

### 3.1.4 Descartar Repositórios Sem Teste

Salvar localmente e analisar o conteúdo de cada arquivo seria bastante custoso, assim optamos por usar a estratégia de buscar a subpalavra “test” diretamente por meio da API GitHub GraphQL. No entanto, para minimizar a quantidade de arquivos escolhidos de maneira incorreta, uma vez que a busca talvez resultasse em muitos falsos positivos, fomos conservadores e fizemos particionadamente para cada linguagem de programação uma análise de quartis no total de menções à subpalavra “test” e descartamos todos os repositórios do primeiro quartil. O resultado pode ser visto na Tabela 3.3.

Para mensurar os efeitos de nossa estratégia conservadora, fizemos uma verificação manual de falsos negativos presentes no primeiro quartil e falsos positivos nos $2^o$, $3^o$ e $4^o$ quartis. Na análise de falsos negativos, com nível de confiança de 95% e margem de erro de 5%, obtivemos um percentual de 8,59% (25 projetos de um total de 291) que, apesar de descartados, possuem casos de teste. Na análise de falsos positivos, com nível de confiança de 95% e margem de erro de 5%, obtivemos um percentual de 5,70% (20 projetos de um total de 351) que, apesar de mantidos, não possuem casos de teste. Não fizemos qualquer tratamento para retirada de *outliers*. A Tabela 3.2 apresenta detalhadamente a distribuição dos falsos positivos e negativos segundo as linguagens de programação.

**Tabela 3.2.** Detalhamento da Taxa de Falsos Positivos e Negativos.

| **Linguagem** | **# FP** | **% FP** | **# FN** | **% FN** |
|---|---|---|---|---|
| Geral | 20 | 5,70 | 25 | 8,59 |
| C | 2 | 7,69 | 2 | 8,70 |
| C# | 0 | 0,00 | 2 | 22,22 |
| C++ | 0 | 0,00 | 4 | 16,00 |
| JAVA | 4 | 8,00 | 5 | 9,80 |
| Javascript | 9 | 7,38 | 9 | 8,33 |
| Python | 5 | 5,43 | 3 | 4,92 |
| Ruby | 0 | 0,00 | 0 | 0,00 |

### 3.1.5 Remover Repositórios Inadequados

Identificamos em uma análise inicial a existência de projetos cujo principal objetivo é conter exemplos de código fonte, como por exemplo, repositórios de cursos ou repositórios de estudo. Para que esse fator não influencie nos resultados da pesquisa, utilizando a saída da etapa anterior, fizemos uma análise manual em todos os repositórios que possuíam arquivos com a subparlavra *test*. Identificamos 112 (2,13%) projetos com essas características. Vale ressaltar que muitos projetos com fins educacionais possuíam testes. Como saída deste passo do processo, identificamos um total de 3.936 repositórios que não são educacionais e possuem testes.

### 3.1.6 Minerar *Pull Requests*

Para que fosse possível identificar os contribuidores e a quantidade de contribuições de código de produção e código de teste, recuperamos todos os PRs abertos e fechados. Em seguida, recuperamos todos os arquivos alterados em cada PR. Levantamos um total de 4.682.412 PRs divididos da seguinte maneira: 353.404 PRs em projetos C, 290.970 PRs em projetos C#, 912.095 PRs em projetos C++, 671.875 PRs em projetos JAVA, 1.158.466 PRs em projetos Javascript, 1.032.581 PRs em projetos Python; e, 263.021 em projetos Ruby.

**Tabela 3.3.** Caracterização dos projetos com relação à presença ou ausência de teste antes e depois da remoção de repositórios inadequados.

| | | Inadequados | | Adequados | |
|---|---|---|---|---|---|
| **Linguagem** | **Categoria** | **# Rep.** | **% Rep.** | **# Rep.** | **% Rep.** |
| Geral | Com teste | 4.048 | 77,28 | 3.936 | 76,79 |
| | Descart. | 1.190 | 22,72 | 1.190 | 23,21 |
| C | Com teste | 277 | 75,07 | 273 | 74,79 |
| | Descart. | 92 | 24,93 | 92 | 25,21 |
| C# | Com teste | 165 | 75,00 | 163 | 74,77 |
| | Descart. | 55 | 25,00 | 55 | 25,23 |
| C++ | Com teste | 355 | 75,37 | 351 | 75,16 |
| | Descart. | 116 | 24,63 | 116 | 24,84 |
| JAVA | Com teste | 633 | 77,29 | 600 | 76,34 |
| | Descart. | 186 | 22,71 | 186 | 23,66 |
| Javascript | Com teste | 1.508 | 77,65 | 1.472 | 77,23 |
| | Descart. | 434 | 22,35 | 434 | 22,77 |
| Python | Com teste | 937 | 78,67 | 906 | 78,10 |
| | Descart. | 254 | 21,33 | 254 | 21,90 |
| Ruby | Com teste | 173 | 76,55 | 171 | 76,34 |
| | Descart. | 53 | 23,45 | 53 | 23,66 |

### 3.1.7 Remover Pull Requests Inadequados

Primeiramente foram excluídos 23.471 PRs que não fazem alterações em arquivos do repositório. Posteriormente, em uma análise inicial, identificamos a existência de PRs criados por *bots*. Para que isso não influenciasse os resultados, fizemos uma busca manual através dos nomes dos usuários. Dentre os 472.321 contribuidores, identificamos analisando os nomes dos usuários e posteriormente pelos PRs 70 contribuidores *bot* nos repositórios. Todos os 249.799 PRs destes usuários e também os usuários foram excluídos. Identificamos também que um usuário chamado *"ghost"* concentrou os PRs de todos os usuários que já foram excluídos no Github. Decidimos mantê-los pois em algum momento da vida dos projetos houve um usuário real solicitando um PR, dessa forma consideramos todos os PRs de usuários reais, mesmo se o usuário não existir mais. Ao final, identificamos 4.432.613 PRs criados por 472.251 diferentes contribuidores nos repositórios selecionados.

## 3.2 Resultados de Caracterização

Após a obtenção e tratamento dos dados classificamos os projetos, as contribuições feitas, e os contribuidores: os projetos foram classificados pela presença ou ausência de testes; as contribuições foram classificadas pela presença apenas de testes, ou pela presença apenas de código de produção, ou pela presença de ambos, teste e código de produção; e, similarmente às contribuições, os

contribuidores foram classificados pela presença de teste, de código de produção, ou por ambos. Após a classificação, caracterizamos as contribuições e os contribuidores de maneira unicamente quantitativa, e conduzimos algumas análises estatísticas que ajudam a explicar o comportamento dos contribuidores de teste e suas contribuições.

### 3.2.1 Classificar os *Pull Requests*

Para classificar os PRs adicionamos três marcadores indicando se o PR possui teste (`hasTest`), possui código (`hasCode`) e/ou outros tipos de contribuição (`hasOthers`). Para isso, executamos um *script* que calcula as seguintes características para cada PR: 1) número de arquivos de testes alterados; 2) o número de arquivos de código alterados; 3) quantidade de linhas adicionadas e removidas; 4) quantidade de outros tipos de arquivos. A atribuição dos marcadores nos PRs segue as seguintes regras:

`hasTest`: O PR possui arquivos adicionados ou alterados, cujo conteúdo é código fonte escrito na linguagem principal do repositório (C, C++, C#, ...), e tais arquivos possuem a palavra "test" em seu nome ou possui menções à palavra *test* em seu conteúdo;

`hasCode`: O PR possui arquivos adicionados, alterados ou excluídos, cujo conteúdo é código fonte escrito na linguagem principal do repositório (C, C++, C#, ...), e tais arquivos não possuem a palavra "test" nem existem menções à palavra *test* em seu conteúdo;

`hasOthers`: O PR possui arquivos adicionados, alterados ou excluídos, cujo conteúdo é outro qualquer que não código fonte escrito na linguagem principal do repositório (C, C++, C#, ...);

De tal maneira, pudemos classificar os PRs em AT (Apenas Teste), AC (Apenas Código) e CT (Código e Teste), usando as seguintes regras:

- PR AT: `(hasTest && !(hasCode || hasOthers))`
- PR AC: `((hasCode || hasOthers) && !hasTest)`
- PR CT: `((hasCode || hasOthers) && hasTest)`

Novamente, fomos bastante conservadores, e tentamos classificar um PR como sendo AT de maneira que qualquer outra contribuição que não for estritamente de teste seja atribuída a AC ou CT. Por exemplo, uma combinação pouco usual, mas que pudemos constatar que existe, é quando um PR tem arquivos de teste alterados e também arquivos que não são código fonte (documentação) da linguagem principal do repositório. Neste caso específico, o PR é classificado como CT.

Além de classificar os PRs em AT, AC e CT, classificamos os PRs em aceitos e não aceitos, o que nos possibilita comparar se existe diferença na aceitação dos PRs AT, AC e CT. Do ponto de vista de aceitação, cujos detalhes da classificação executada são mostrados na Tabela 3.4, pode-se observar que 75,88% dos PRs AT são aceitos, frente a 72,32% dos PRs CT e 73,38% de PRs AC.

Considerando todos os PRs, aqueles classificados como AT (112.969) representam somente 2.56% do total de PRs dos repositórios, enquanto todos PRs que possuem teste, ou seja, AT e CT, representam 22,53% do total de PRs dos repositórios. Uma distribuição similar é observada se

considerarmos unicamente os PRs aceitos: PRs AT representam 2,65%, enquanto os PRs que possuem teste, ou seja, AT e CT, representam 22,38% do total de PRs aceitos dos repositórios.

**Tabela 3.4.** Caracterização dos PRs com relação ao tipo de contribuição – AT, AC e CT – e aceite. Resultados em relação ao total de projetos sem considerar linguagens e, também, considerando cada uma das linguagens.

| | | Pull Requests | | |
|---|---|---|---|---|
| **Linguagem** | **Categ.** | **# Total** | **# Aceitas** | **% Aceitas** |
| Geral | AC | 3.415.626 | 2.506.450 | 73,38 |
| | AT | 112.969 | 85.722 | 75,88 |
| | CT | 880.547 | 636.846 | 72,32 |
| C | AC | 319.142 | 224.379 | 70,31 |
| | AT | 3.861 | 2.710 | 70,19 |
| | CT | 26.185 | 17.254 | 65,89 |
| C# | AC | 223.802 | 184.688 | 82,52 |
| | AT | 3.987 | 3.551 | 89,06 |
| | CT | 30.775 | 25.465 | 82,75 |
| C++ | AC | 805.645 | 593.614 | 73,68 |
| | AT | 9.207 | 6.521 | 70,83 |
| | CT | 73.824 | 50.982 | 69,06 |
| JAVA | AC | 389.620 | 277.889 | 71,32 |
| | AT | 30.374 | 22.514 | 74,12 |
| | CT | 221.507 | 152.593 | 68,89 |
| Javascript | AC | 804.408 | 553.350 | 68,79 |
| | AT | 21.257 | 13.833 | 65,08 |
| | CT | 197.998 | 139.864 | 70,64 |
| Python | AC | 666.690 | 521.387 | 78,21 |
| | AT | 37.891 | 31.696 | 83,65 |
| | CT | 293.458 | 227.013 | 77,36 |
| Ruby | AC | 206.319 | 151.143 | 73,26 |
| | AT | 6.392 | 4.897 | 76,61 |
| | CT | 36.800 | 23.675 | 64,33 |

Ainda, para caracterizar melhor os PRs, refinamos os resultados apenas dos PRs aceitos, mostrando a média e a mediana do tempo de aceite nas categorias AC, AT, CT. Os resultados detalhados são apresentados na Tabela 3.5.

### 3.2.2 Classificar os Contribuidores

Classificamos os contribuidores em três categorias de acordo com a categoria dos PRs processados no passo anterior: MC AT (Membros Contribuidores de Apenas Teste), são aqueles que contribuíram com PRs que contém apenas código de teste; MC AC (Membros Contribuidores de Apenas Código) são aqueles que contribuíram com PRs que contém código de produção ou outros arquivos do

**Tabela 3.5.** Caracterização dos PRs aceitos com relação ao tempo de aceite em horas – AT, AC, CT. Resultados em relação ao total de projetos e também considerando cada uma das linguagens de programação.

| | Média em horas | | | Mediana em horas | | |
|---|---|---|---|---|---|---|
| **Linguagem** | **PR AC** | **PR AT** | **PR CT** | **PR AC** | **PR AT** | **PR CT** |
| Geral | 202,18 | 119,66 | 302,34 | 14,84 | 8,32 | 34,76 |
| C | 224,65 | 171,31 | 452,97 | 17,82 | 16,12 | 71,56 |
| C# | 182,93 | 77,31 | 207,96 | 18,83 | 8,39 | 37,69 |
| C++ | 192,66 | 178,60 | 307,00 | 17,18 | 12,43 | 41,82 |
| JAVA | 154,49 | 97,16 | 219,53 | 10,45 | 8,67 | 25,07 |
| Javascript | 253,08 | 139,88 | 293,05 | 14,97 | 7,51 | 29,57 |
| Python | 169,61 | 108,40 | 335,07 | 11,55 | 8,00 | 42,54 |
| Ruby | 243,51 | 162,41 | 558,88 | 12,80 | 4,39 | 29,81 |

repositório que não são destinados a teste; e MC CT (Membros Contribuidores de Código e Teste) que contribuíram com código e teste e, eventualmente, outros arquivos dentro do repositório, inclusive, por exemplo, um PR AT e um outro PR AC.

Ao todo identificamos 468.007 membros contribuidores nos repositórios selecionados. Destes, 128.868 membros (27,54%) contribuíram em mais de 1 repositório, enquanto a grande maioria dos contribuidores, ou seja, 339.139 usuários (72,46%), contribuíram apenas em um repositório. Por isso, além de considerar o tipo de contribuição para classificar os contribuidores como MC AT, MC AC, MC CT, decidimos analisa-los em dois escopos distintos: o primeiro escopo, que chamamos de Escopo de Repositório, considera individualmente cada repositório, de maneira que, por exemplo, um usuário pode ser MC AT em um repositório e MC AC ou MC CT em outros repositórios; o segundo escopo, que chamamos de Escopo da Amostra, considera toda nossa amostra de 3.936 repositórios adequados ao estudo. Com isso, por exemplo, para um contribuidor ser considerado um MC AT, ele deve ter apenas PRs AT em toda a amostra.

A Tabela 3.6 apresenta a quantidade de repositórios por MC no escopo de repositório. Considerando o escopo de repositório, dos 468.007 MCs, 1,61% (7.524) são MC ATs, enquanto 37,74% (176.616) são MCs que fazem teste (AT ou CT). Nas três classificações, AC, AT e CT, percebe-se uma concentração de contribuidores em apenas um repositório, respectivamente 75,32%, 97,11% e 82,42%. Considerando os MC AC, apesar da concentração em apenas um repositório, existe um certo espalhamento de contribuidores, em relação à quantidade de repositórios, quando comparados aos MC AT e MC CT. Por sua vez, considerando apenas MC AT, percebe-se que os contribuidores estão mais concentrados em apenas 1 repositório do que os contribuidores AC e CT, mas existem peculiaridades, como por exemplo um contribuidor AT que contribuiu em 55 diferentes repositórios. Porém, é possível notar que tanto os MC AT quanto os MC CT possuem espalhamento menor do que os MC AC.

Segmentamos a caracterização dos MC AT (do escopo de repositórios) segundo as linguagens de programação, cujos resultados são mostrados na Tabela 3.7. Vale ressaltar que os dados mostrados nessa tabela (em qualquer uma das colunas) foram conjuntos não disjuntos de contribuidores: um mesmo contribuidor pode ser AT em repositórios C, C++. C#, Java, Javascript, Python ou Ruby.

**Tabela 3.6.** Caracterização dos contribuidores segundo o escopo do repositório (% MC >= 0,01). As contribuições (PRs abertas) também são classificadas segundo seu tipo (AT, AC e CT).

| Apenas Código | | | Apenas Teste | | | Código e Teste | | |
|---|---|---|---|---|---|---|---|---|
| # MC | % MC | # Rep. | # MC | % MC | # Rep. | # MC | % MC | # Rep. |
| 294.433 | 75,32 | 1 | 6.850 | 97,11 | 1 | 106.082 | 82,42 | 1 |
| 55.248 | 14,13 | 2 | 147 | 2,08 | 2 | 14.837 | 11,53 | 2 |
| 19.533 | 5,00 | 3 | 26 | 0,37 | 3 | 4.239 | 3,29 | 3 |
| 8.730 | 2,23 | 4 | 8 | 0,11 | 4 | 1.648 | 1,28 | 4 |
| 4.603 | 1,18 | 5 | 6 | 0,09 | 5 | 815 | 0,63 | 5 |
| 2.569 | 0,66 | 6 | 2 | 0,03 | 6 | 371 | 0,29 | 6 |
| 1.609 | 0,41 | 7 | 5 | 0,07 | 7 | 245 | 0,19 | 7 |
| 1.065 | 0,27 | 8 | 1 | 0,01 | 8 | 145 | 0,11 | 8 |
| 746 | 0,19 | 9 | 2 | 0,03 | 9 | 86 | 0,07 | 9 |
| 512 | 0,13 | 10 | 1 | 0,01 | 11 | 58 | 0,05 | 10 |
| 390 | 0,10 | 11 | 1 | 0,01 | 13 | 56 | 0,04 | 11 |
| 247 | 0,06 | 12 | 1 | 0,01 | 14 | 28 | 0,02 | 12 |
| 201 | 0,05 | 13 | 1 | 0,01 | 19 | 28 | 0,02 | 13 |
| 155 | 0,04 | 14 | 1 | 0,01 | 27 | 17 | 0,01 | 14 |
| 132 | 0,03 | 15 | 1 | 0,01 | 28 | 7 | 0,01 | 15 |
| 97 | 0,02 | 16 | 1 | 0,01 | 55 | 11 | 0,01 | 16 |
| 100 | 0,03 | 17 | – | – | – | – | – | – |
| 66 | 0,02 | 18 | – | – | – | – | – | – |
| 57 | 0,01 | 19 | – | – | – | – | – | – |
| 48 | 0,01 | 20 | – | – | – | – | – | – |
| 35 | 0,01 | 21 | – | – | – | – | – | – |
| 36 | 0,01 | 22 | – | – | – | – | – | – |
| 28 | 0,01 | 23 | – | – | – | – | – | – |
| 30 | 0,01 | 24 | – | – | – | – | – | – |
| 23 | 0,01 | 25 | – | – | – | – | – | – |

Conforme pode ser observado na tabela com os detalhes para as diferentes linguagens, mantém-se a tendência de apenas uma colaboração nos MC que colaboram com PRs AT. Em algumas linguagens isso é mais facilmente observado, por exemplo, C#, C++, em outras existe um espalhamento maior, como, Java e Python.

**Tabela 3.7.** Caracterização dos MC AT segundo o escopo do projeto em todas as Linguagens.

| Apenas Teste | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C | | | C# | | | C++ | | | JAVA | | | Javascript | | | Python | | | Ruby | | |
| # MC | % MC | # Rep. | # MC | % MC | # Rep. | # MC | % MC | # Rep. | # MC | % MC | # Rep. | # MC | % MC | # Rep. | # MC | % MC | # Rep. | # MC | % MC | # Rep. |
| 147 | 97,35 | 1 | 99 | 99 | 1 | 443 | 99,55 | 1 | 1.440 | 96,58 | 1 | 2.441 | 98,03 | 1 | 1.746 | 96,95 | 1 | 608 | 97,59 | 1 |
| 2 | 1,32 | 2 | 1 | 1 | 2 | 2 | 0,45 | 2 | 25 | 1,68 | 2 | 45 | 1,81 | 2 | 37 | 2,05 | 2 | 13 | 2,09 | 2 |
| 1 | 0,66 | 3 | – | – | – | – | – | – | 9 | 0,60 | 3 | 1 | 0,04 | 3 | 10 | 0,56 | 3 | 2 | 0,32 | 3 |
| 1 | 0,66 | 4 | – | – | – | – | – | – | 4 | 0,27 | 4 | 1 | 0,04 | 5 | 1 | 0,06 | 4 | – | – | – |
| – | – | – | – | – | – | – | – | – | 4 | 0,27 | 5 | 1 | 0,04 | 8 | 2 | 0,11 | 6 | – | – | – |
| – | – | – | – | – | – | – | – | – | 1 | 0,07 | 6 | 1 | 0,04 | 17 | 1 | 0,06 | 7 | – | – | – |
| – | – | – | – | – | – | – | – | – | 3 | 0,20 | 7 | – | – | – | 1 | 0,06 | 8 | – | – | – |
| – | – | – | – | – | – | – | – | – | 1 | 0,07 | 9 | – | – | – | 1 | 0,06 | 9 | – | – | – |
| – | – | – | – | – | – | – | – | – | 1 | 0,07 | 11 | – | – | – | 1 | 0,06 | 13 | – | – | – |
| – | – | – | – | – | – | – | – | – | 1 | 0,07 | 19 | – | – | – | 1 | 0,06 | 37 | – | – | – |
| – | – | – | – | – | – | – | – | – | 1 | 0,07 | 27 | – | – | – | – | – | – | – | – | – |
| – | – | – | – | – | – | – | – | – | 1 | 0,07 | 28 | – | – | – | – | – | – | – | – | – |

Considerando o escopo da amostra, cujos detalhes são mostrados na Tabela 3.8 dos 468.007 MCs, 0,69% (3.212) são MC ATs, enquanto 28,55% (133.626) são MCs que fazem teste (AT ou CT). De tal forma, para detalhar as contribuições AT, calculamos os PRs feitos pelos contribuidores. Conforme o esperado, existe uma concentração de contribuidores AT em apenas um PR. Tal concentração diminui gradativamente nos contribuidores AC e CT. Existem, portanto, 2.708 contribuidores AT que contribuem com um único PR (84,31% dos contribuidores).

**Tabela 3.8.** Caracterização dos contribuidores – AT, AC, CT – segundo o escopo da amostra (20 contribuições mais relevantes quando considerados o número de MCs).

| **Apenas Código** | | | **Apenas Teste** | | | **Código e Teste** | | |
|---|---|---|---|---|---|---|---|---|
| **# MC** | **% MC** | **# PR** | **# MC** | **% MC** | **# PR** | **# MC** | **% MC** | **# PR** |
| 196.743 | 58,84 | 1 | 2.708 | 84,31 | 1 | 32.182 | 24,68 | 1 |
| 58.823 | 17,59 | 2 | 348 | 10,83 | 2 | 19.233 | 14,75 | 2 |
| 26.384 | 7,89 | 3 | 94 | 2,93 | 3 | 12.617 | 9,67 | 3 |
| 14.443 | 4,32 | 4 | 24 | 0,75 | 4 | 8.841 | 6,78 | 4 |
| 9.596 | 2,87 | 5 | 8 | 0,25 | 5 | 6.613 | 5,07 | 5 |
| 6.005 | 1,80 | 6 | 10 | 0,31 | 6 | 5.180 | 3,97 | 6 |
| 3.987 | 1,19 | 7 | 5 | 0,16 | 7 | 4.061 | 3,11 | 7 |
| 2.725 | 0,81 | 8 | 4 | 0,12 | 8 | 3.405 | 2,61 | 8 |
| 2.203 | 0,66 | 9 | 3 | 0,09 | 9 | 2.744 | 2,10 | 9 |
| 1.681 | 0,50 | 10 | 1 | 0,03 | 10 | 2.294 | 1,76 | 10 |
| 1.320 | 0,39 | 11 | 2 | 0,06 | 11 | 1.980 | 1,52 | 11 |
| 1.080 | 0,32 | 12 | 2 | 0,06 | 12 | 1.702 | 1,31 | 12 |
| 905 | 0,27 | 13 | 1 | 0,03 | 14 | 1.513 | 1,16 | 13 |
| 736 | 0,22 | 14 | 1 | 0,03 | 18 | 1.288 | 0,99 | 14 |
| 650 | 0,19 | 15 | 1 | 0,03 | 41 | 1.201 | 0,92 | 15 |
| 571 | 0,17 | 16 | – | – | – | 1.077 | 0,83 | 16 |
| 462 | 0,14 | 17 | – | – | – | 934 | 0,72 | 17 |
| 430 | 0,13 | 18 | – | – | – | 829 | 0,64 | 18 |
| 335 | 0,10 | 19 | – | – | – | 789 | 0,60 | 19 |
| 325 | 0,10 | 20 | – | – | – | 734 | 0,56 | 20 |

### 3.2.3 Análise de Aceitação de PRs por Tipo de PR

Executamos o teste chi-quadrado nos PRs para verificar se existe diferença nas aceitações considerando se o PR é AC, AT ou CT. O resultado do valor de p foi menor que 0,05, indicando que há evidências o suficiente para mostrar a não homogeneidade da aceitação dos PRs nas diferentes categorias – existe, portanto, um efeito de associação entre aceitação e o tipo de contribuição. Analisando os valores residuais, pudemos constatar que os PRs CT são os que mais influenciam na não aceitação de um PR e, os PRs AT são os que mais influenciam a aceitação de um PR. Usamos *odds ratio* para medir o tamanho do efeito da associação: os PRs AC possuem uma razão de chance de 1,05 de serem aceitos em relação aos PRs CT; e, os PRs AT possuem uma razão de chance de 1,20 de serem aceitos em relação aos PRs CT. Assim, segundo Cohen (COHEN, 1988) o tamanho do efeito é considerado muito pequeno.

Com o intuito de investigar o mesmo questionamento mas para cada linguagem de programação, aplicamos o teste chi-quadrado usando correção de Bonferroni. Para todas as linguagens também existe efeito de associação entre aceitação do PR e o tipo de contribuição. Novamente, usamos *odds ratio* para medir o tamanho do efeito da associação em cada uma das linguagens, conforme pode ser visto na Tabela 3.9. *Odds ratio* é uma medida de tamanho de feito relativa, ou seja, ela mostra a razão de chances de um fator em relação a um outro fator.

No nosso caso, que usamos três fatores, as razões de chance de PRs AC e PRs AT são relativas às razões de chance de PRs CT. Por exemplo, para cada uma chance de um PR CT na linguagem C ser aceito, existe 1,23 chances de um PR AC ser aceito e 1,22 chances de um PR AT ser aceito. Assim, pode-se verificar que o tamanho de efeito é muito pequeno ou pequeno para todas as linguagens de programação.

Observando os resultados das razões de chances pode-se perceber que o efeito de associação entre o tipo de PR e aceitação varia segundo a linguagem de programação, por exemplo: em C, as razões de chance de PRs AC e AT são quase iguais; em C#, as razões de chance de PRs AT são maiores que os de PRs AC; em C++, as razões de chance de PRs AC são maiores que os de PRs AT; e, em Javascript, PRs AC e AT são menores do que a referência, os PRs CT.

**Tabela 3.9.** Resultado de teste estatístico de associação entre tipos de PRs e aceitação ou não dos PRs e tamanhos de efeito segundo cada linguagem de programação.

| | | Odds Ratio | | | Interpretação | |
|---|---|---|---|---|---|---|
| **Linguagem** | **Valor p** | **AC** | **AT** | **CT** | **Int. AC** | **Int. AT** |
| C | 0,00 | 1,23 | 1,22 | 1,00 | M.Peq. | M.Peq. |
| C# | 0,00 | 0,98 | 1,70 | 1,00 | M.Peq. | Peq. |
| C++ | 0,00 | 1,25 | 1,09 | 1,00 | M.Peq. | M.Peq. |
| JAVA | 0,00 | 1,12 | 1,29 | 1,00 | M.Peq. | M.Peq. |
| Javascript | 0,00 | 0,92 | 0,77 | 1,00 | M.Peq. | M.Peq. |
| Python | 0,00 | 1,05 | 1,50 | 1,00 | M.Peq. | Peq. |
| Ruby | 0,00 | 1,52 | 1,82 | 1,00 | Peq. | Peq. |

### 3.2.4 Análise de Tempo de Aceitação de PRs por Tipo de PR

O teste de Kruskal-Wallis para verificar se existe diferença no tempo de aceitação de PRs nas três categorias independentemente da linguagem resultou um valor de p igual a zero, o que nos permite descartar a hipótese nula de que não há diferença no tempo de aceite entre os tipo de PR. O tamanho do efeito foi computado usando Cohen's d, e em todas as combinações de tipos de PR e foi encontrado um tamanho de efeito muito pequeno (COHEN, 1988). Como o valor de p do teste de Kruskal-Wallis foi menor do que 0,05, executamos o teste U de Mann-Whitney par-a-par com os tipos de PR, e o valor p em todas as combinações foi menor que 0,05.

Também verificamos se existe diferença no tempo de aceitação de PRs nas três categorias para cada linguagem de programação. Os detalhes da execução do teste U de Mann-Whitney par-a-par

com correção de Bonferroni nos tipos de PR e segundo as linguagens de programação podem ser analisados por meio da Tabela 3.10. Apenas em um caso não há evidências suficientes para concluir que existe diferença significa, ou seja, valor de p maior que 0,05: na comparação dos tempos de aceitação de PR AC e de PR AT usando C como linguagem de programação. Nos outros casos o valor de p é menor que 0,05. Novamente calculamos o tamanho do efeito usando Cohen's d, e os resultados são em todos os casos muito pequeno ou pequeno (COHEN, 1988).

**Tabela 3.10.** Resultado de teste estatístico entre tipos de PRs e tempo de aceitação segundo cada linguagem de programação: "V.p" é o valor de p; "Cohen's d" é o valor de Cohen's d; e, "Int. C.'d" é a interpretação de Cohen's d.

| | PR AC x PR AT | | | PR AT x PR CT | | | PR AC x PR CT | | |
|---|---|---|---|---|---|---|---|---|---|
| **Ling.** | **Valor de p** | **Cohen's d** | **Int. C.'d** | **Valor de p** | **Cohen's d** | **Int. C.'d** | **Valor de p** | **Cohen's d** | **Int. C.'d** |
| C | 0,80 | —— | ——- | 0,00 | 0,21 | Peq. | 0,00 | 0,21 | Peq. |
| C# | 0,00 | 0,14 | M.Peq. | 0,00 | 0,19 | M.Peq. | 0,00 | 0,03 | M.Peq. |
| C++ | 0,00 | 0,02 | M.Peq. | 0,00 | 0,12 | M.Peq. | 0,00 | 0,12 | M.Peq. |
| JAVA | 0,00 | 0,07 | M.Peq. | 0,00 | 0,15 | M.Peq. | 0,00 | 0,08 | M.Peq. |
| Javascript | 0,00 | 0,10 | M.Peq. | 0,00 | 0,13 | M.Peq. | 0,00 | 0,03 | M.Peq. |
| Python | 0,00 | 0,07 | M.Peq. | 0,00 | 0,19 | M.Peq. | 0,00 | 0,16 | M.Peq. |
| Ruby | 0,00 | 0,07 | M.Peq. | 0,00 | 0,19 | M.Peq. | 0,00 | 0,23 | Peq. |

## 3.3 Análise histórica do contribuidor

Após analisarmos os pull requests e os contribuidores, buscamos mapear a evolução das contribuições de testes desses usuários ao longo do tempo. É importante destacar que esses colaboradores podem ter começado com testes e, posteriormente, evoluído para a codificação. Com o objetivo de coletar esses dados, dividimos o processo em duas etapas.

A primeira etapa teve como foco o escopo local, ou seja, levamos em consideração o repositório em que cada contribuidor trabalhava ao montar a linha do tempo de suas contribuições. Isso permitiu que pudéssemos visualizar de forma mais precisa a evolução dos testes em cada projeto. Identificamos que, ao considerarmos o repositório na análise, 5.201 contribuidores fizeram seu primeiro PR com foco em apenas testes e 131.596 contribuidores que fizeram algum tipo de teste em seu primeiro PR no repositório. Além disso, há 51 contribuidores que fizeram o primeiro PR em repositórios diferentes contendo apenas testes.

Na segunda etapa, o foco foi no escopo global. Nesse caso, levamos em consideração apenas o contribuidor e a evolução de suas contribuições de testes em todos os projetos em que ele participou. Essa análise permitiu entender a evolução do colaborador em relação à sua atuação em diferentes projetos e repositórios. Com base nessa análise, foi possível identificar que de 482.771 contribuidores, 4.923 contribuidores fizeram seu primeiro PR com foco em apenas testes. Observamos também que houveram 84.609 contribuidores que fizeram algum tipo de teste em seu primeiro PR considerando o escopo global dos repositórios selecionados para a pesquisa.

Empreender um esforço para criar um histórico de transações entre tipos de contribuição para cada mês do contribuidor. Primeiro foi necessário agrupar por mês a quantidade de contribuições um contribuidor fez em cada mês de vida desde o primeiro PR identificado na pesquisa até o ultimo. Em seguida, classificamos o mês em AT, AC, CT usando como base o agrupamento da etapa anterior. Com essa classificação fizemos uma análise histórica de cada contribuidor registrando, mês a mês, os tipos de contribuição que foram sendo executadas. O resultado desses dados podem ser vistos na figura 3.3.

### 3.3.1 Análise do perfil de contribuir de testes

Após a conclusão da análise histórica do contribuidor, constatou-se a existência de padrões entre os contribuidores e suas contribuições. Por conta disso, foi realizada uma análise manual dos contribuidores, com o objetivo de criar perfis de contribuição. Para essa análise, foram considerados o número de asserts, linhas de código, comentário e linhas em branco removidos ou adicionados. Para executar essa análise, foi necessário fazer uso da ferramenta PRAnalyzer, conforme descrita na seção seguinte.

## 3.4 PRAnalyzer

Para responder à principal questão de pesquisa, foi indispensável a construção de uma ferramenta apta a analisar meticulosamente as linhas alteradas de uma contribuição, objetivando obter nuances mais refinadas e detalhadas sobre os elementos constituintes de um PR (Pull Request). No período que antecedeu a confecção desta dissertação, realizamos uma busca extensiva por ferramentas já existentes que pudessem desempenhar essa função. Entretanto, até o momento da publicação deste material, não identificamos nenhum instrumento disponível que atendesse a esta necessidade específica.

Diante desta lacuna, tomamos a iniciativa de desenvolver, durante o processo de pesquisa, uma aplicação inédita denominada PRAnalyzer. Esta ferramenta tem a capacidade de capturar todas as linhas de código alteradas presentes em um PR, proporcionando uma análise detalhada que identifica não apenas modificações em asserts — sejam eles adicionados ou removidos —, mas também percebe alterações em linhas de código distintas, variando desde comentários até linhas em branco.

Para alcançar essa funcionalidade, a ferramenta PRAnalyzer opera através de um conjunto de expressões regulares. Essas expressões foram projetadas para identificar uma variedade de elementos cruciais em códigos escritos em seis linguagens de programação distintas. Dentre os elementos que a ferramenta é capaz de reconhecer, estão inclusos métodos ou funções, classes, asserts, imports, comentários e linhas em branco, conferindo assim um caráter multidimensional e multifacetado à análise proporcionada.

Esta inovação representou um marco significativo na nossa pesquisa, dado que permitiu a identificação precisa de PRs que, a princípio, eram classificados como AT, mas que, sob uma inspeção mais detalhada, revelavam alterações em outros elementos, como comentários ou itens que não se enquadravam necessariamente como um teste. Desta forma, o PRAnalyzer serviu como um filtro

refinado, possibilitando uma categorização mais acurada e evitando classificações equivocadas que poderiam comprometer a validade dos resultados da nossa investigação.

Além disso, a ferramenta demonstrou ser fundamental para promover uma compreensão mais aprofundada da natureza das contribuições, oferecendo insights valiosos sobre a composição e a estrutura dos PRs analisados. Como resultado, foi possível discernir, as múltiplas camadas de alterações presentes em cada PR, criando um panorama mais rico e detalhado do ecossistema de contribuições em projetos de software.

### 3.4.1 Codificação

O PRAnalyzer foi desenvolvido utilizando Python, uma linguagem que atualmente se destaca como referência no campo de análise de código fonte, não apenas por sua robustez, mas também pela facilidade de uso que oferece aos desenvolvedores. Esta escolha foi feita estrategicamente para aproveitar as potencialidades intrínsecas da linguagem, que facilita a implementação de funcionalidades complexas por meio de uma sintaxe mais clara e acessível. O PRAnalyzer configura-se como uma biblioteca que pode ser facilmente integrada em projetos Python existentes, sendo possível importá-la diretamente em seu código para enriquecer e facilitar os processos de análise de PRs.

Dado o volume significativo de dados que necessitávamos analisar durante nossa pesquisa, recomendamos a utilização do PRAnalyzer em combinação com um sistema de gestão de filas eficiente, facilitando assim a análise de múltiplos itens de maneira paralela e distribuída. Esta estratégia mostrou-se não apenas viável, mas extremamente eficaz em nossos estudos, permitindo uma análise mais rápida e coordenada de grandes conjuntos de dados. Nos parágrafos subsequentes, delinearemos os principais métodos implementados na biblioteca, oferecendo uma visão detalhada das funcionalidades que o PRAnalyzer está equipado para realizar, de modo a auxiliar desenvolvedores e pesquisadores na exploração profunda e detalhada dos PRs que desejam analisar.

```
1 class PRAnalizer():
2   def __init__(self, language):
3         pass
4     def verify(self, text):
5         pass
6     def checkModifierType(self, text):
7         pass
8     def checkIfModifier(self, text):
9         pass
10    def retornaEstrutura(self):
11        pass
```

**Código 3.1.** Métodos existentes na classe principal do PRAnalyzer

0 PRAnalyzer emprega expressões regulares como mecanismo para identificar e categorizar os diferentes tipos de contribuições codificadas. Este procedimento necessita que, no momento da

instanciação da classe principal da biblioteca, seja fornecida a linguagem predominante utilizada no PR em questão. A especificação da linguagem permite o carregamento apropriado das expressões regulares pertinentes, assegurando assim uma análise sintática precisa e afinada à sintaxe da linguagem em foco.

Para facilitar a flexibilidade e evolução contínua do sistema, optamos por adotar um formato chave/valor na definição dos elementos que necessitam ser identificados no PR. Esta abordagem possibilita não somente a identificação eficaz de características pré-definidas, tais como "asserts", "imports"e "comentários", mas também a incorporação futura de novos atributos de análise de maneira simplificada e estruturada, abrindo portas para uma ferramenta cada vez mais abrangente e ajustável às demandas crescentes e dinâmicas no campo de análise de códigos.

Além das expressões regulares previamente carregadas, incorporamos uma estratégia específica para identificar linhas em branco e segmentos de código de produto. A tática adotada inicia com a seleção das linhas alteradas, seguida da eliminação de todos os espaços presentes. Posteriormente, realizamos uma análise do conteúdo residual para determinar a natureza da linha em foco.

Delineamos assim, que uma linha resultante com zero caracteres é categoricamente identificada como uma "linha em branco". Este processo assegura que todas as porções do texto desprovidas de conteúdo substantivo sejam corretamente categorizadas, facilitando a análise subsequente e evitando análises inapropriadas de espaços vazios. Por outro lado, se a linha alterada, após a remoção dos espaços, preserva um ou mais caracteres válidos e não corresponde a nenhum padrão identificado pelas expressões regulares pré-definidas, ela é classificada como uma linha contendo "código de produto".

Até o momento da publicação deste trabalho, o PRAnalyzer conta com as seguintes expressões:

```
1 def __init__(self, language):
2     self.language = language
3
4     if(language == "Python"):
5       self.possibilidades = {
6         '.*assert.*\(.*'           : 'tests',
7         '.*expect.*\(.*'         : 'tests',
8         '.*import .*'          : 'imports',
9         '..*from .*import .*'      : 'imports',
10        '[\-|\+]\s{0,}#.*'         : 'comments',
11        '[\-|\+]\s{0,}""".*"""'    : 'comments',
12        "[\-|\+]\s{0,}'''.*'''"    : 'comments',
13      }
14
15    if(language == "JAVA"):
16      self.possibilidades = {
17        '.*assert.*\(.*'       : 'tests',
```

```
18          '.*import .*\;'          : 'imports',
19          '[\-|\+]\s{0,}#.*'       : 'comments',
20          '[\-|\+]\s{0,}\/\*\*.*'  : 'comments',
21          '[\-|\+]\s{0,}\*\/.*'    : 'comments',
22          '[\-|\+]\s{0,}\/\/.*'    : 'comments',
23        }
24
25      if(language == "C"):
26        self.possibilidades = {
27          '.*assert.*\(.*'           : 'tests',
28          '.*\#include.*\.h'           : 'imports',
29          '[\-|\+]\s{0,}\/\*.*\*\/'  : 'comments',
30          '[\-|\+]\s{0,}\/\/.*'      : 'comments',
31        }
32
33      if(language == "C++"):
34        self.possibilidades = {
35          '.*assert.*\(.*'             : 'tests',
36          '.*TEST_EQUAL\(.*'             : 'tests',
37          '.*import .*\;'              : 'imports',
38          '.*[\-|\+]\s{0,}\/\*.*\*\/'    : 'comments',
39          '[\-|\+]\s{0,}\/\/.*'      : 'comments',
40        }
41
42      if(language == "C#"):
43        self.possibilidades = {
44          '.*assert.*\(.*'             : 'tests',
45          '.*Assert.*\(.*'             : 'tests',
46          '.*import .*\;'              : 'imports',
47          '.*using .*\;'               : 'imports',
48          '.*[\-|\+]\s{0,}\/\*.*\*\/'    : 'comments',
49          '[\-|\+]\s{0,}\/\/.*'      : 'comments',
50        }
51
52      if(language == "Javascript"):
53        self.possibilidades = {
54          '.*assert.*\(.*'             : 'tests',
55          '.*expect\(.*\).*'             : 'tests',
56          '.*import .*\;'              : 'imports',
57          '.*[\-|\+]\s{0,}\/\*.*\*\/'    : 'comments',
58          '[\-|\+]\s{0,}\/\/.*'      : 'comments',
59        }
60
61      if(language == "Ruby"):
62        self.possibilidades = {
63          '.*assert.*'                 : 'tests',
64          '.*require .*\;'               : 'imports',
```

```
65          '.*[\-|\+]\s{0,}\/\*.*\*\/'     : 'comments',
66          '[\-|\+]\s{0,}\/\/.*'           : 'comments',
67      }
```

**Código 3.2.** Expressões de busca existentes na primeira versão da biblioteca

Além da especificação da linguagem principal do repositório, a nossa ferramenta requer que todos os arquivos alterados no PR sejam capturados. Esse procedimento se mostra fundamental, pois garante uma visão integral de todas as contribuições feitas pelo contribuidor em uma única vez.

Ao obter o conjunto completo dos arquivos que sofreram modificações, o PRAnalyzer inicia um processo de inspeção, analisando cada arquivo individualmente, linha por linha, para identificar e catalogar todas as alterações implementadas. Neste contexto, o método **checkIfModifier** é responsável por verificar se uma linha específica experimentou alguma forma de modificação. Esta verificação assegura que cada contribuição seja avaliada com a devida atenção aos detalhes e que linhas não alteradas sejam desconsideradas na análise.

```
1 def checkIfModifier(self, text):
2     if(len(text) >= 1):
3      if(text.strip()[0] == "-" or text.strip()[0] == "+"):
4          return True
5      else:
6          return False
```

**Código 3.3.** Método responsável pela identificação se houve uma alteração

Uma vez confirmada a presença de modificações, o método **checkModifierType** é acionado para verificar a natureza da alteração efetuada, identificando se houve adição ou remoção de código. Essa distinção é vital, pois fornece informações que serão importantes em análises futuras, especialmente ao avaliar aspectos como a evolução da cobertura de testes. Por exemplo, é possível antecipar se as alterações tenderiam a ampliar ou diminuir a cobertura de testes, oferecendo assim uma prévia importante para os revisores sobre os impactos potenciais da PR no desempenho e na robustez do software.

```
1 def checkModifierType(self, text):
2     if(len(text) >= 1):
3         if(text.strip()[0] == "-"):
4             return 'removed'
5
6         if(text.strip()[0] == "+"):
7             return 'added'
8
9     return 'others'
```

**Código 3.4.** Método de verificação do tipo da alteração (adição ou exclusão)

Finalizando o processo, o método **retornaEstrutura** se encarrega de apresentar a estrutura de resposta gerada pelo método **verify**. Inicialmente, **verify** define cada linha modificada como uma alteração de código. Para refinar essa classificação, é efetuado um processo de limpeza que remove todos os espaços da linha; se não restar nenhum caráter válido, então identifica-se aquela linha como uma linha em branco.

Seguindo adiante, a função procede com uma análise utilizando o conjunto de expressões regulares especificamente elaboradas para a linguagem determinada no momento da instanciação da classe, promovendo uma identificação dos tipos de alterações executadas no PR. A conclusão desse procedimento tem como saída o retorno do tipo de modificação identificado, fornecendo assim uma vista detalhada e bem fundamentada das alterações realizadas. A utilização de expressões regulares permite com que possamos reutilizar o mesmo código de análise em diferentes *frameworks* de testes

```
1 def verify(self, text):
2     tipo = "code"
3
4     if (len(text.strip()[1::]) == 0):
5         tipo = 'whiteLines'
6
7     for expressao in self.possibilidades:
8         validador    = re.compile(expressao)
9
10        if validador.match(text.strip()):
11            tipo = self.possibilidades[expressao]
12
13
14    return tipo
```

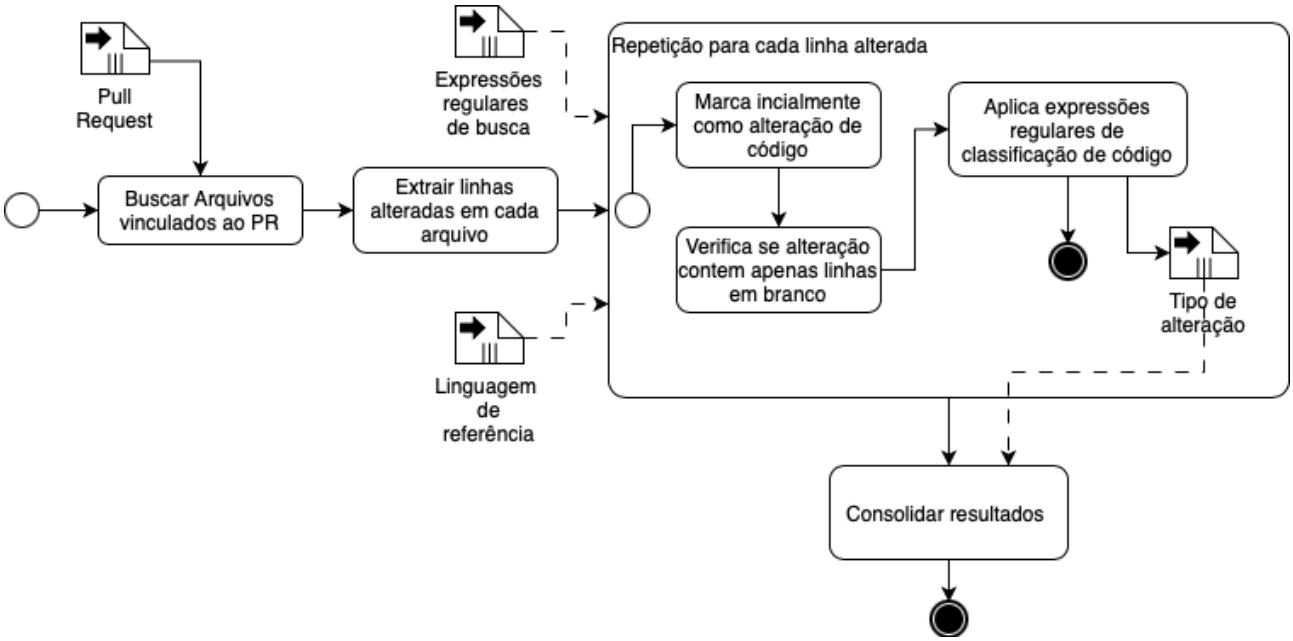
**Código 3.5.** Método principal de análise

### 3.4.2 Fluxo de análise

O fluxo utilizado pelo PRAnalyzer para analisar e classificar os tipos de contribuições pode ser visto na figura-3.2.

Primeiramente é necessário que se faça uma busca por todos os arquivos contidos no PR e suas respectivas alterações. Após a coleta de todos os arquivos alterados, é necessário que haja a extração de todas as linhas alteradas e o tipo da alteração (remoção ou inserção). Essa última informação somada com a linguagem referencia do repositório, permite com que o PRAnalyzer passa linha a linha e execute 3 grandes passos: 1) Definir como padrão uma alteração de código. Isso se mostrou mais eficiente do que criar expressões para todo código possível de ser alterado; 2) Eliminar todos os espaços em branco e calculando a quantidade de caracteres restantes. Se o resultado desta operação for zero, significa que a alteração contém apenas espaços em branco; 3) Por

**Figura 3.2.** Passos executados pelo PRAnalyzer.




último, o algoritmo utiliza todas as expressões regulares específicas da linguagem de referência para determinar se foi alterado a assinatura de um método, alterada uma classe, criado um novo assert etc.

Após a utilização do PRAnalyzer na base da pesquisa, foi possível adicionarmos mais um filtro de alterações em teste. O primeiro filtro utilizado chegamos a afirmação que um PR possui alteração nos testes com base no arquivo alterado. Dessa forma, se houve qualquer alteração em um arquivo que obedeça as regras já citadas na sessão-3.2.1 o PR possui um indicio de alteração em códigos destinados a teste de software. Quando utilizamos o PRAnalyzer neste mesmo PR, é possível identificar qual parte do código ele alterou, se foi um novo assert, um novo import ou se apenas adicionou um novo comentário.

Para ilustrar melhor o funcionamento, apresentamos a seguir o trecho de código desenvolvido especificamente para a execução deste processo:

```
1 def analisarPR(arrayFiles, pr_id, linguagemReferencia):
2   analizer = PRAnalizer(linguagemReferencia)
3   dadosDoPR  = analizer.retornaEstrutura();
4
5   for item in arrayFiles:
6     if 'patch' in item:
7       itensAlterados = item['patch']
8
9       aux = itensAlterados.split("\n");
10
11      for item in aux:
12        if(analizer.checkIfModifier(item.strip())):
13          result       = analizer.verify(item.strip())
14          modifierType  = analizer.checkModifierType(item.strip())
```

```
15            dadosDoPR[result][modifierType] += 1
16            dadosDoPR['all'][modifierType] += 1
17
18    return dadosDoPR
```

**Código 3.6.** Exemplo de utilização do PRAnalyzer

Ao executar o método **analisarPR**, obtemos a seguinte saída:

*{'functions': {'removed': 0, 'added': 0, 'others': 0}, 'tests': {'removed': 0, 'added': 0, 'others': 0}, 'class': {'removed': 0, 'added': 0, 'others': 0}, 'code': {'removed': 0, 'added': 1, 'others': 0}, 'imports': {'removed': 0, 'added': 0, 'others': 0}, 'comments': {'removed': 0, 'added': 1, 'others': 0}, 'whiteLines': {'removed': 0, 'added': 1, 'others': 0}, 'all': {'removed': 0, 'added': 3, 'others': 0}}*

Durante o desenvolvimento deste trabalho, o registro do software foi solicitado junto a Universidade Tecnológica Federal do Paraná - Campo Mourão.

## 3.5 Análise das Issues

Nesta seção iremos explicar foi como feita a análise das issues para conseguimos chegar na resposta da principal questão de pesquisa.

### 3.5.1 Filtragem dos 50 projetos com mais número de estrelas

Para garantir uma análise mais apurada, que naturalmente requer um tempo mais prolongado, optamos por reduzir a amostra para os 50 projetos que possuíam o maior número de PRs. A distribuição destes projetos por linguagem de programação ficou da seguinte forma: 21 em JavaScript, 9 em C++, 8 em Python, 7 em Java, 3 em C e 1 em C#. Ao longo deste estudo, examinamos um total de 460.866 PRs.

Dessa quantidade substancial, a grande maioria, correspondendo a 97,34% ou 448.638 PRs, já encontrava-se concluída. O percentual restante, que representa 2,66% ou 12.228 PRs, ainda estava pendente de ações por parte dos mantenedores.

### 3.5.2 Análise do código commitado

Para cada PR, fizemos um levantamento abrangente de todos os arquivos que sofreram alterações, foram excluídos ou inseridos, além das linhas que foram modificadas. Essa coleta detalhada de dados serviu como insumo para o PRAnalyzer, facilitando a análise e extração das características distintas de cada tipo de contribuição.

Esse processo meticuloso nos permitiu identificar com precisão os PRs que continham modificações em "asserts", referindo-se às alterações, criações ou exclusões destes. Com isso, constatamos que, do total de 460.866 PRs avaliados, apenas 15.852 (ou 3,43%) apresentavam alterações em algum "assert".

### 3.5.3 Análise e tratamento das Issues Inseridas no Body dos PRs

Para entender melhor a regularidade das contribuições de testes em PRs resultado de correções de bugs, o conteúdo conteúdo de cada PR. Nosso objetivo era identificar os padrões de associação entre uma PR e uma Issue, seguindo as orientações delineadas na documentação do GitHub.

No decorrer dessa investigação, catalogamos um total de 264.841 Issues relacionadas aos PRs que escolhemos como foco de nossa pesquisa. Nos dedicamos, então, a rastrear o estado atual de cada Issue identificada, com um olhar atento para qualquer etiqueta que indicasse a presença de um bug. O resultado desta análise culminou na identificação de 25.193 Issues devidamente etiquetadas como "BUG".

## 3.6 Considerações finais

Ao longo da seção de metodologia, delineamos as etapas e ferramentas adotadas na condução de nossa pesquisa. A priorização das linguagens de programação investigadas, fundamentada no índice TIOBE, conferiu relevância à pesquisa, dando foco nas linguagens mais empregadas na atualidade, refletindo as demandas reais do mercado.

Na captação dos dados, direcionamos nossos esforços para a análise dos repositórios, dos PRs e suas respectivas alterações. Identificamos repositórios que continham testes, ao buscarmos a palavra "test"em nomes de arquivos ou em seus caminhos. O resultado desta busca - 4.048 repositórios com testes, representando impressionantes 77,28% da amostra - revela uma tendência clara na comunidade de desenvolvimento em direção à implementação de práticas de testes.

Um aspecto crucial do nosso trabalho foi a curadoria dos dados. A remoção de PRs que não apresentavam alterações garantiu a precisão e relevância de nossa análise. Avançando nesse campo, a classificação dos PRs quanto a AC, AT e CT possibilitou uma análise detalhada de cada tipo. Ao observarmos a taxa de aceitação e o tempo associado a cada classificação de PR, conseguimos discernir nuances na maneira como diferentes contribuições são percebidas e administradas.

Dando continuidade ao nosso esforço, mergulhamos na análise do fluxo e do histórico dos contribuidores. Tal perspectiva proporcionou uma compreensão de como os contribuidores evoluem ao longo do tempo e como os padrões de contribuição são estabelecidos e mantidos por indivíduos e equipes.

Um dos marcos mais notáveis da nossa investigação foi a criação e implementação do PRanalyzer. Esta biblioteca, desenvolvida para atender aos objetivos desta dissertação, mostrou-se uma ferramenta valiosa. Ao detalharmos sua criação e funcionamento, esperamos que possa ser útil para pesquisas futuras, servindo como instrumento para outros acadêmicos e profissionais.

## 3.7 Resultados e Discussão dos Resultados

Os contribuidores AT podem ser encontrados de maneira consistente entre os diversos repositórios e linguagens de programação. Comparado com as PRs AC e CT, suas contribuições AT são acentuadamente mais concentradas em apenas um repositório e apenas um PR, embora existam casos peculiares.

Considerando como escopo o repositório, dos 468.007 contribuidores, 7.054 contribuidores são AT (1,51%) e 128.710 são CT (27,50%), totalizando 135.764 que contribuem de alguma maneira com testes (29,01%). Considerando o escopo da amostra, dos 468.007 contribuidores, 3.212 são contribuidores AT (0,69%) e 130.414 são CT (27,87%), totalizando 138.976 contribuidores que colaboram com teste (29,70%). Considerando o escopo da amostra, das 4.409.142 PRs, 112.969 são PRs AT (2,56%) e 880.547 são PRs CT (19,97%), totalizando 993.516 PRs que contém teste (22,53%). Nas PRs aceitas, de um total de 3.229.018 de PRs, 85.722 PRs são AT (2,65%) e 636.846 (19,72%) são CT, totalizando 722.568 PRs que contém testes (22,38%).

**Proporção de Projetos e PRs por Membro Contribuidor.**

Observamos que há uma concentração expressiva de contribuidores que participam em um único projeto OSS e, também, de contribuidores com apenas um único PR. Na literatura existem diversas barreiras para entrar em projetos OSS, tais como problemas na comunicação com os mantenedores, habilidades necessárias, orientações quanto a o que é necessário ser produzido e até mesmo barreiras por questões culturais (STEINMACHER et al., 2015). A grande maioria de MC ATs (97,11%) fizeram contribuições em apenas um repositório, e com apenas um PR (84,31%). Contribuidores CT também são concentrados em apenas um repositório (82,42%), embora de uma maneira mais branda, mas com um espalhamento maior no número de PRs, apenas 24,68% em com apenas um PR. Por sua vez, contribuidores AC são menos concentrados ainda nos repositórios (75,32%) e ainda com número de contribuições menos espalhadas em diferentes PRs (58,84%) comparado com contribuidores CT.

Além dos problemas para iniciar em um novo projeto, há também barreiras para se continuar contribuindo com um projeto. Os usuários que contribuem uma única vez são chamados de contribuidores episódicos ou casuais (BARCOMB et al., 2022; PINTO et al., 2016). Na maioria dos casos, a comunicação entre os mantenedores e os contribuidores fazem com que haja apenas um envolvimento do usuário em todo o projeto. Além disso, podemos encontrar também a recusa de novas ideias, falta de tempo dos mantenedores e até mesmo elitismo (LEE et al., 2017). Somados a estes problemas, estão as desmotivações ligadas aos testes já citadas, fazendo com que haja mais PRs únicos do que contribuições de um mesmo usuário ao longo do tempo.

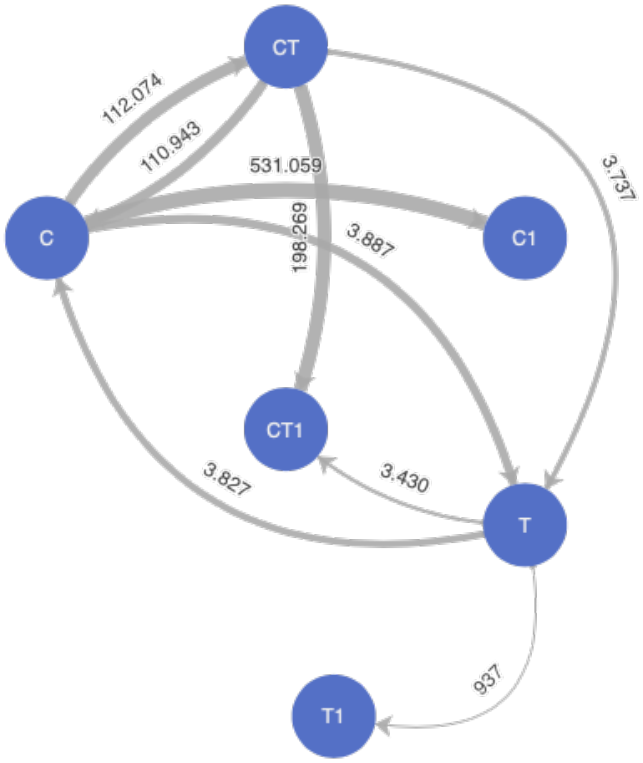
Considerando PRs abertas *versus* PRs aceitas, como pode ser observado na Tabela 3.4, há um número expressivo de PRs que não possuem testes sendo aprovadas durante o procedimento de revisão de código. Esse padrão se mantém em todas as linguagens, sendo que em Javascript o

percentual de PRs aceitas é o menor (68,79%). Nos projetos utilizando Python, Java, Ruby e C# os PRs categorizados como AT tiveram uma maior taxa de aprovação. Observando de maneira geral, essa taxa chega a 75,88% de aprovação contra 73,38% (PRs AC) e 72,32% (PRs CT). Essa percepção de diferença na aceitação ou não de PRs com teste foi confirmada pelos resultados das análises estatísticas, no entanto, pelos testes de tamanho de efeito podemos concluir que o tipo de PR influência de maneira fraca, tanto na aceitação ou não de um PR, quanto no tempo de aceitação de um PR.

**Timeline de contribuição.**

A Figura 3.3 ilustra a dinâmica de transição entre os diferentes tipos de contribuição feitas por contribuidores em uma análise abrangente dos repositórios. Nota-se, predominantemente, que os indivíduos que contribuem exclusivamente com código tendem a manter esse padrão nas PRs subsequentes. Além disso, é importante destacar que uma fração mínima mantém-se realizando contribuições que envolvem apenas testes. Ao analisar outros padrões, observamos que os contribuidores de CT têm uma tendência maior a persistir com PRs de CT, comparativamente aos que se dedicam a AC e AT.

**Figura 3.3.** Fluxo de transições mensais dos contribuidores no escopo global, onde C e C1 significam contribuições de apenas código, CT e CT1 significam contribuições de código e teste e T e T1 significam contribuições de teste.




Fica evidente que um contribuidor, categorizado como AC, tende a persistir contribuindo apenas com código de produto, em vez de expandir sua contribuição para código de produto e testes.

A situação em que um contribuidor AC dedica-se exclusivamente aos testes é ainda mais rara. Isso reforça a validade das razões identificadas na literatura para a desmotivação em contribuir com testes.

**Perfil do contribuidor de testes.**

Na análise realizada, identificamos cinco tipos distintos de perfis que desempenham papéis cruciais no processo de contribuição de testes: Contribuidor Movimentação, Atualizador de Cenários, Especialista em Bibliotecas, Analista Comentário e o estratégico Contribuidor de Evolução. Detalharemos a seguir as especificidades e atribuições de cada um deles.

O perfil do Contribuidor Movimentação se dedica a transposições de classes ou cenários de teste entre diferentes arquivos, uma tarefa que se torna necessária sobretudo em momentos de refatoração de código ou reestruturação. Este contribuidor se destaca por produzir PRs que apresentam um equilíbrio entre linhas removidas e adicionadas e uma prevalência de ajustes que não alteram substancialmente o código original.

Segue-se o Contribuidor Atualizador de Cenários, que dedica sua expertise na revitalização dos cenários de teste existentes, enfatizando a otimização dos códigos que fundamentam os *asserts*. Este profissional muitas vezes concentra suas modificações em *imports* e comentários, mantendo a inserção de novos *asserts* em um nível mínimo.

O Contribuidor de Bibliotecas, por sua vez, empreende uma missão de limpeza e otimização, focando na eliminação de bibliotecas e *imports* supérfluos para potencializar o desempenho e a clareza do código.

Identificamos ainda o perfil Analista Comentário, um colaborador meticuloso que, embora atue predominantemente na correção textual e adição de comentários, cumpre um papel vital para manter a qualidade e a coerência do código. Este perfil se destaca por realizar contribuições que incorporam comentários em mais de 90% das linhas modificadas, com uma quase ausência de alterações significativas no código e nos *asserts*.

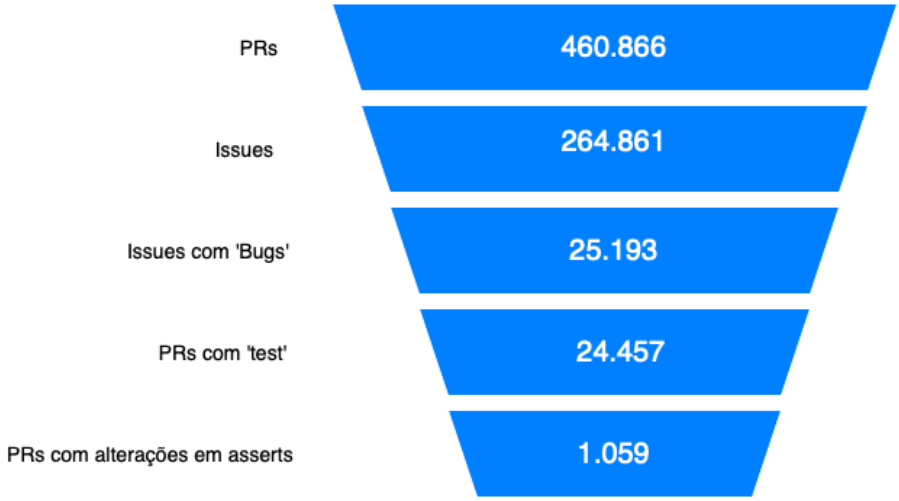
Por último, destacamos o perfil primordial do Contribuidor de Evolução. Este é o motor inovador nos projetos OSS, voltando sua energia para a concepção de novos cenários e *asserts* de teste. Este colaborador é identificado principalmente pela proporção significativamente positiva de *asserts* inseridos em relação aos removidos, marcando uma tendência de acréscimo e inovação contínua.

Dos 53 colaboradores analisados, 18 foram categorizados como Atualizadores de Cenários, o que corresponde a aproximadamente 34% do total. Além disso, 12 foram identificados como Contribuidores de Movimentação, representando cerca de 23% do grupo. Registramos 8 colaboradores com especialização em Bibliotecas, correspondendo a aproximadamente 15% do total. Cinco colaboradores foram classificados como Analistas de Comentários, equivalendo a aproximadamente 9% do conjunto. Por fim, 10 colaboradores se destacaram como Contribuidores Estratégicos para a Evolução, compreendendo cerca de 19% do total. Não foi feita nenhuma distinção entre os contribuidores com os repositórios ou linguagens.

**Contribuições de testes quando se corrige um bug.**

Após levantar todas as informações foi possível chegarmos a resposta da principais questão de pesquisa: quão frequente é a contribuição de teste em PRs resultantes de correções de bugs? Os números considerados para responder esta pergunta podem ser observados na figura Figura 3.4

**Figura 3.4.** Funil de informações.




Quando recuperamos as informações de um total de 460.866 Pull Requests (PRs) selecionados para o estudo, podemos observar que estão vinculados a esses PRs um total de 108.588 issues criadas. É importante considerar que, dentro do Github, uma issue pode ter escopos como melhoria, dúvida técnica, divida técnico ou defeito. Geralmente, essa classificação é feita usando tags dentro da issue.

Dentre essas 108.588 issues criadas, foi constatado que 25.193 (23.20%) são bugs existentes no projeto, ou seja, além de estarem vinculadas a um PR, também possuíam uma tag sinalizando ser um bug. Essas 25.193 issues de bug estão relacionadas a 83.054 PRs. Portanto, podemos notar que existem PRs que se propõem a corrigir o mesmo bug ou partes diferentes de um bug maior.

Ao aprofundarmos ainda mais a análise e considerarmos a classificação quanto a existência de testes buscando pelo termo *test* nos arquivos do PR, encontramos 24.457 registros que alteraram algum tipo de arquivo de teste. Após aplicarmos o PRAnalyzer e analisarmos os resultados, chegamos ao número de 1.983 PRs que contêm inserção ou exclusão de asserts.

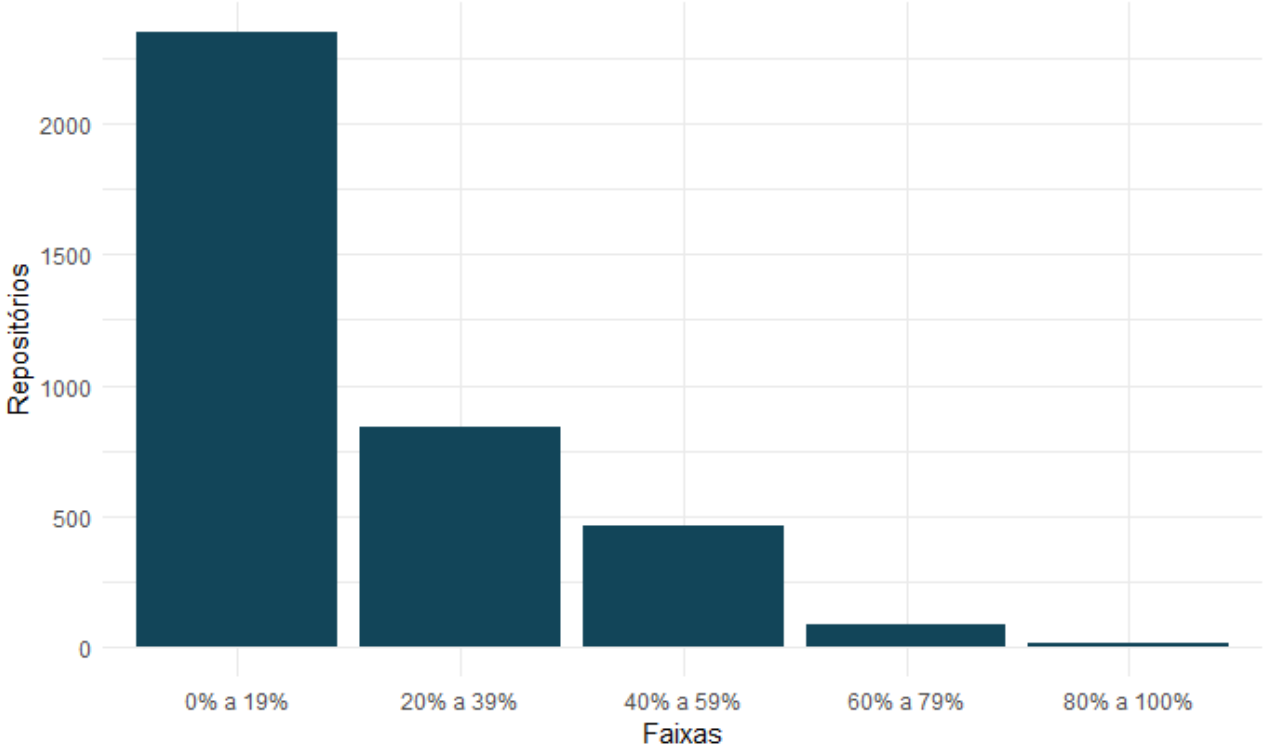
Dessa forma, podemos concluir que, dos 25.193 PRs que corrigem algum bug, há um total de 1.983 (7.87%) PRs que também evoluem os cenários de testes existentes.

### Aceite de PRs com testes

Para avaliar a validade da afirmação de (ALAMI et al., 2020) no contexto da amostra deste estudo, conduzimos duas análises: 1) Examinamos a proporção de PRs aceitos e mesclados que apresentam modificações em arquivos de teste; 2) avaliamos o número de PRs aceitos e mesclados que realizam alterações nos *asserts*. Em ambos os cenários, consideramos apenas os PRs que foram efetivamente mesclados. Isso nos permite inferir que a presença ou ausência desse fator específico não foi determinante na decisão de aceitar ou rejeitar o PR.

Nossos resultados mostram, conforme visto na Figura 3.5 que a vasta maioria dos projetos tem menos de 50% de seus PRs modificando arquivos de teste. Isso sugere que PRs que introduzem atualizações ou correções de bugs têm uma tendência maior a não apresentar alterações correspondentes nos testes.

**Figura 3.5.** Percentual de PRs aceitos e mesclados contendo alterações em arquivos de teste.




Ao aprofundarmos nossa análise focando exclusivamente em alterações de *asserts* (sejam adições ou exclusões), os resultados tornam-se mais alarmantes. De 3752 repositórios analisados, somente em dois deles a proporção de PRs com alterações em *asserts* varia entre 20% e 39%. Em outros dois repositórios, essa taxa eleva-se para uma faixa de 40% a 59%.

Notamos que 184 repositórios não foram incluídos em ambas as análises. Ao investigar mais detalhadamente, constatamos que nesses repositórios não havia nenhum PRs, dentre os gerados pela pesquisa, que tivesse sido aceito. Realizamos uma avaliação manual para verificar possíveis erros na extração de dados, mas a própria interface gráfica do Github indicou que os PRs não foram mesclados. Suspeitamos que essas inconsistências podem ter surgido devido ao uso inadequado dos fluxos do GitHub ou de processos internos específicos dos repositórios.

# 4 CONCLUSÕES

Nesta seção, consolidamos os principais achados e reflexões emergentes, delineando não apenas o que aprendemos, mas também o que este conhecimento sugere para futuras investigações no campo.

## 4.1 Considerações Finais

Esta dissertação se propôs a investigar a frequência da inclusão de testes em PRs decorrentes de correções de *bugs* em projetos de software livre. Para alcançar esse objetivo, utilizamos uma análise quantitativa de diversos repositórios de código aberto, coletando informações relacionadas aos projetos, PRs e os seus respectivos contribuidores. Esta abordagem nos permitiu não apenas identificar a presença de contribuintes dedicados à inclusão de testes, mas também examinar as características específicas das contribuições de teste presentes nos PRs e determinar perfis padrão de contribuidores.

### 4.1.1 Contribuição 1: os resultados do estudo e resposta das questões de pesquisa

Conduzimos um estudo com 3.936 repositórios de projetos OSS e 7 diferentes e importantes linguagens de programação (C, C++, C#, Java, Javascript, Python e Ruby), analisando um total de 4.409.142 PRs, classificando membros contribuidores e suas contribuições. Usamos uma abordagem bastante conservadora para classificar os contribuidores e suas contribuições como apenas teste (quanto presente apenas código de teste), apenas código (quando presente apenas código de produção) e código e teste (quando presente código de produção e código de teste) em duas diferentes dimensões de tamanho: escopo de repositório e escopo da amostra. Desenvolvemos um extrator de informações do código contribuído e criamos perfis de contribuidores com base em seus históricos.

Nossos resultados mostram que os contribuidores que fazem PRs de apenas teste existem, independentemente da linguagem de programação. Eles estão concentrados em contribuir em apenas um projeto e apenas um PR. De maneira similar, os outros tipos de contribuição também estão concentrados em apenas um projeto e apenas um PR, mas de maneira menos acentuada.

Ao olharmos para a questão principal da pesquisa, de um total de 460.866 PRs haviam 25.193 que sinalizaram de forma clara e formal a correção de um BUG. Destes 25.193 PRs apenas 1.983 (7.87%) evoluíram algum cenário de teste existente no projeto. Identificamos também a existência de de cinco perfis de contribuidores: Contribuidor Movimentação, Contribuidor Atualizador de Cenários, Contribuidor de Bibliotecas, Contribuidor Comentário e, por fim, um dos mais importantes, o Contribuidor de Evolução.

Contrastando nossos resultados com a literatura, algumas questões podem ser levantadas: porque contribuições AT são ainda mais concentradas em apenas um projeto que contribuições CT? porque as barreiras são maiores? porque contribuições AT não possuem um nível maior de aceitação

uma vez que os projetos ainda precisam de testes e os PRs de teste são apenas aproximadamente 22% do total de PRs? porque que nas correções de bugs o percentual de teste é menor do que nos PRs que adicionam uma nova funcionalidade? Mesmo com tais questionamento, podemos concluir que ainda é muito baixa a criação de novos cenários nas correções de BUGs. independentemente do projeto ou de linguagem de programação. Novos estudos futuros, especialmente qualitativos, devem ser conduzidos para caracterizar as motivações de tais contribuidores.

### 4.1.2 Contribuição 2: o PRAnalyzer

O PRAnalyzer surgiu como uma ferramenta útil para ir além da análise convencional dos pull requests (PRs). Antes, nos concentrávamos principalmente em dados já disponíveis nos PRs, como tempo até sua aprovação, número de comentários, descrição e conexões com outros PRs e Issues. Com o PRAnalyzer, temos a oportunidade de olhar diretamente para o código contribuído, abrindo portas para uma compreensão mais detalhada do conteúdo inserido.

Além disso, a estrutura do PRAnalyzer foi pensada para ser flexível e adaptável. Isso pode ser útil para pesquisas que queiram investigar aspectos específicos, como contextos de métodos, situações de testes ou novas estruturas de assert, especialmente à medida que novos frameworks vão sendo introduzidos no campo.

### 4.1.3 Contribuição 3: o conjunto de dados minerados

Além do PRAnalyzer, que é oferecido com licença de software livre, disponibilizamos um conjunto de dados abrangendo 610.028 PRs, detalhando suas características em relação à adição ou remoção de asserts, linhas em branco, comentários e código. Conforme discutido no capítulo sobre trabalhos relacionados, esse conjunto de dados não só enriquece futuras investigações permitindo uma análise mais profunda do que apenas comparar o caminho do arquivo para determinar se é uma contribuição de teste, mas também possibilita identificar se o código comitado apresenta modificações em asserts ou a inserção de linhas em branco. O dataset jutamente com os scripts criados para esta pesquisa podem ser acessados através do link <https://zenodo.org/records/10258052>

### 4.1.4 Contribuição 4: publicação de artigo

Um dos benefícios conquistados através dessa pesquisa foi a publicação de um artigo científico que abordou a análise de projetos de software livre em relação à presença de testes em seus repositórios, bem como informações sobre os contribuidores e suas contribuições. Esse artigo contribuiu para o corpo de conhecimento na área de engenharia de software, oferecendo informações valiosos sobre testes em projetos de software livre e também um dataset contendo as informações mineradas de mais de 5 mil repositórios. O artigo pode ser acessado através da plataforma ACM através do link <https://dl.acm.org/doi/10.1145/3555228.3555244>.

## 4.2 Limitações do Estudo

Uma limitação importante do estudo é a detecção de repositórios com teste e, posteriormente as PRs que contém algum arquivo com teste. Usamos, como uma decisão de diminuir o esforço, a estratégia de minerar os dados em dois passos por meio da API do Github. Isso porque queríamos analisar uma grande amostra, como foi o caso, de mais de 4 milhões de PRs. Identificar os arquivos com teste em diferentes linguagens de programação que podem usar diferentes *frameworks* de teste com uma amostra desse tamanho em um tempo razoável é um desafio considerável. Por isso reportamos a taxa de falsos positivos e verdadeiros negativos, o que torna o estudo mais transparente. Tal ameaça seria minimizada com técnicas mais custosas, como por exemplo, uma análise qualitativa dos PRs.

Na abordagem metodológica desta dissertação, optamos por explorar um amplo espectro de repositórios em detrimento da análise minuciosa de um número restrito deles. A escolha por essa estratégia visa proporcionar uma visão abrangente do cenário, permitindo uma compreensão mais holística das práticas relacionadas aos PRs que abordam correções de defeitos e a presença de testes associados.

Esta pesquisa reside na natureza focalizada da análise, que se concentrou exclusivamente nos PRs em cada repositório. Embora tenhamos identificado e categorizado cinco perfis distintos de contribuidores com base em suas interações por meio desses PRs, reconhecemos que essa abordagem não proporciona uma visão abrangente de todas as contribuições de um indivíduo em diferentes contextos. A análise limitada aos PRs pode obscurecer nuances e padrões que poderiam emergir ao considerar a totalidade das contribuições de um colaborador em vários repositórios.

Outra limitação do estudo é a detecção e classificação dos contribuidores. Como foi mencionado, existem usuários não humanos. A maioria desses usuários foram removidos em uma análise superficial para minimizar tal efeito e um deles permaneceu no estudo, o usuário “ghost”. Como ele possui muitos PRs (25.001) e representa usuários reais, optamos por mantê-lo juntamente com suas PRs. O efeito negativo de tal decisão é que no escopo da amostra ele e seus PRs foram classificamos como CT, o que representa 0,57% das PRs CT.

Também, podemos conjecturar que quanto mais contribuições um contribuidor fizer, e portanto por mais tempo ele permanecer contribuindo, maior é a tendência em ele contribuir não apenas com código de produção, mas também com código de teste. Aceitando tal conjectura, com o tempo, muitos MC ATs e MC AC se tornam contribuidores CT. Então, a proporção de PRs AT, PRs AC e PRs CT varia, fazendo com o tempo seja uma dimensão importante para a classificação dos contribuidores em AT, AC, e CT.

É possível que os contribuidores não tenham vinculado corretamente as issues às pull requests (PRs) de acordo com as diretrizes do GitHub. No entanto, neste estudo, assumimos que os contribuidores vincularam corretamente as PRs que corrigiram bugs atribuindo um dos termos recomendados pelo GitHub, conforme descrito na Seção 2.

Por fim, é crucial reconhecer que a abordagem adotada nesta pesquisa, centrada no número de PRs, não incluiu uma avaliação minuciosa da qualidade dos PRs, tanto no aspecto do código quanto na qualidade dos testes associados. A análise mais aprofundada da qualidade dos testes poderia revelar nuances importantes, destacando casos em que as alterações não se limitam a modificações de asserts, mas estendem-se a segmentos de código que exercem influência direta sobre esses asserts. A consideração da qualidade não apenas das alterações de código, mas também dos testes, poderia proporcionar uma compreensão mais refinada da eficácia das correções implementadas. Contudo, é imperativo reconhecer que a avaliação qualitativa envolve um ônus significativo em termos de recursos e esforços, sendo necessário ponderar cuidadosamente a relação custo-benefício para determinar a viabilidade de tal empreendimento em projetos de escala semelhante.

## 4.3 Trabalhos Futuros

Como propostas para trabalhos futuros, podem-se citar:

i) Evolução do PRAnalyzer para que ele vá além da captura de *asserts*. A ideia é que ele seja capaz de reconhecer e documentar cenários de teste completos. Este avanço não só proporcionará uma análise mais refinada e precisa da presença de testes, como também garantirá que a totalidade e a nuance do contexto do teste sejam integralmente consideradas.
ii) Compreensão das motivações que levam os contribuidores a introduzir novos cenários de teste em PRs, especialmente quando estes são originados de correções de defeitos. Entender esses impulsionadores pode nos fornecer dados valiosos sobre a cultura de teste e aprimoramento contínuo dentro da comunidade.
iii) Percebendo uma potencial lacuna no campo dos testes de regressão em projetos *de software livre*, propomos a elaboração de uma revisão sistemática atualizada. Esta revisão permitirá captar e sintetizar as mais recentes tendências, práticas e descobertas na área, garantindo que estejamos alinhados com o estado da arte em pesquisa e prática.
iv) Aprofundar a análise dos perfis dos contribuidores identificados nesta pesquisa pode proporcionar uma maior precisão na classificação, possibilitando a identificação de outras características relevantes.

# REFERÊNCIAS

ABDOU, Tamer; GROGONO, Peter; KAMTHAN, Pankaj. A conceptual framework for open source software test process. In: *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. [S.l.: s.n.], 2012. p. 458–463.

ABERDOUR, Mark. Achieving quality in open-source software. *IEEE Software*, v. 24, n. 1, p. 58–64, 2007.

ALAMI, Adam; COHN, Marisa Leavitt; WAISOWSKI, Andrzej. How do foss communities decide to accept pull requests? In: *Proceedings of the Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (EASE '20), p. 220–229. ISBN 9781450377317.

ALAMI, Adam; DITTRICH, Yvonne; WASOWSKI, Andrzej. Influencers of quality assurance in an open source community. In: *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018. (CHASE '18), p. 61–68. ISBN 9781450357258.

ALENEZI, Mamdouh; AKOUR, Mohammed; HUSSIEN, Alaa; AL-SAAD, Mohammad Z. Test suite effectiveness: an indicator for open source software quality. In: *2016 2nd International Conference on Open Source Software Computing (OSSCOM)*. [S.l.: s.n.], 2016. p. 1–5.

ALSHARA, Zakarea; SALMAN, Hamzeh Eyal; SHATNAWI, Anas; SERIAI, Abdelhak-Djamel. Ml-augmented automation for recovering links between pull-requests and issues on github. *IEEE Access*, v. 11, p. 5596–5608, 2023.

ANDAM, Berima; BURGER, Andreas; BERGER, Thorsten; CHAUDRON, Michel R. V. Florida: Feature location dashboard for extracting and visualizing feature traces. In: . New York, NY, USA: Association for Computing Machinery, 2017. (VaMoS '17), p. 100–107. ISBN 9781450348119. Disponível em: <https://doi.org/10.1145/3023956.3023967>.

ANVIK, John; HIEW, Lyndon; MURPHY, Gail C. Who should fix this bug? In: *Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2006. (ICSE '06), p. 361–370. ISBN 1595933751. Disponível em: <https://doi.org/10.1145/1134285.1134336>.

BARCOMB, Ann; STOL, Klaas-Jan; FITZGERALD, Brian; RIEHLE, Dirk. Managing episodic volunteers in free/libre/open source software communities. *IEEE Transactions on Software Engineering*, v. 48, n. 1, p. 260–277, 2022.

BARHAM, Adina. The emergence of quality assurance practices in free/libre open source software: A case study. In: PETRINJA, Etiel; SUCCI, Giancarlo; IOINI, Nabil El; SILLITTI, Alberto (Ed.). *Open Source Software: Quality Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 271–276.

BASHIR, Muhammad Farhan; BANURI, Syed Hammad Khalid. Automated model based software test data generation system. In: *2008 4th International Conference on Emerging Technologies*. [S.l.: s.n.], 2008. p. 275–279.

BEECHAM, Sarah; BADDOO, Nathan; HALL, Tracy; ROBINSON, Hugh; SHARP, Helen. Motivation in software engineering: A systematic literature review. *Information and Software Technology*, v. 50, n. 9, p. 860–878, 2008. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584907001097>.

BELLER, Moritz; GOUSIOS, Georgios; PANICHELLA, Annibale; ZAIDMAN, Andy. When, how, and why developers (do not) test in their IDEs. *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, p. 179–190, 2015.

BERTONCELLO, Marcus Vinicius; PINTO, Gustavo; WIESE, Igor Scaliante; STEINMACHER, Igor. Pull requests or commits? which method should we use to study contributors' behavior? In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2020. p. 592–601.

BIASI, Luciano; BECKER, Karin. Geração automatizada de drivers e stubs de teste para junit a partir de especificações u2tp. 01 2006.

BIRD, Christian; ZIMMERMANN, Thomas. Assessing the value of branches with what-if analysis. In: . New York, NY, USA: Association for Computing Machinery, 2012. (FSE '12). ISBN 9781450316149. Disponível em: <https://doi.org/10.1145/2393596.2393648>.

BONACCORSI, Andrea; ROSSI, Cristina. Why open source software can succeed. *Research Policy*, v. 32, n. 7, p. 1243–1258, 2003. ISSN 0048-7333. Open Source Software Development. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0048733303000519>.

BöHMER, Hans J. Stetter Klaus. *Defect Correction Methods*. [S.l.]: Springer Vienna, 2012.

CAPRETZ, Luiz Fernando; VARONA, Daniel; RAZA, Arif. Influence of personality types in software tasks choices. *Computers in Human Behavior*, v. 52, p. 373–378, 2015. ISSN 0747-5632.

CAPRETZ, Luiz Fernando; WAYCHAL, Pradeep; JIA, Jingdong; VARONA, Daniel; LIZAMA, Yadira. Studies on the software testing profession. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. [S.l.: s.n.], 2019. p. 262–263.

CAPRETZ, Luiz Fernando; WAYCHAL, Pradeep Kashinath. Why a testing career is not the first choice of engineers. *ASEE Annual Conference and Exposition, Conference Proceedings*, v. 2016-June, n. June, 2016.

CHATURVEDI, K.K.; SING, V.B.; SINGH, Prashast. Tools in mining software repositories. In: *2013 13th International Conference on Computational Science and Its Applications*. [S.l.: s.n.], 2013. p. 89–98.

CHEN, Tse-Hsun; THOMAS, Stephen W.; NAGAPPAN, Meiyappan; HASSAN, Ahmed E. Explaining software defects using topic models. In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2012. p. 189–198.

COHEN, Jacob. *Statistical power analysis for the behavioral sciences*. 2nd edition. ed. [S.l.]: Routledge, 1988. 567 p. ISBN 9780805802832.

DAKA, Ermira; FRASER, Gordon. A survey on unit testing practices and problems. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. [S.l.: s.n.], 2014. p. 201–211.

DALLAL, Jehad Al. Automation of object-oriented framework application testing. In: *2009 5th IEEE GCC Conference & Exhibition*. [S.l.: s.n.], 2009. p. 1–5.

DEAK, Anca. A comparative study of testers' motivation in traditional and agile software development. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 8892, p. 1–16, 2014. ISSN 16113349.

DEAK, Anca; STÅLHANE, Tor; SINDRE, Guttorm. Challenges and strategies for motivating software testing personnel. *Information and Software Technology*, v. 73, p. 1–15, 2016. ISSN 09505849.

Del Bianco, Vieri; LAVAZZA, Luigi; MORASCA, Sandro; TAIBI, Davide; TOSI, Davide. An investigation of the users' perception of OSS quality. *IFIP Advances in Information and Communication Technology*, v. 319 AICT, p. 15–28, 2010. ISSN 18684238.

EICKELMANN, Nancy S.; RICHARDSON, Debra J. An evaluation of software test environment architectures. In: *Proceedings of the 18th International Conference on Software Engineering*. USA: IEEE Computer Society, 1996. (ICSE '96), p. 353–364. ISBN 0818672463.

FRANÇA, A. César C.; Da Silva, Fabio Q.B. An empirical study on software engineers motivational factors. *2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, p. 405–409, 2009.

FRANÇA, César; SHARP, Helen; SILVA, Fabio. Motivated software engineers are engaged and focused, while satisfied ones are happy. *International Symposium on Empirical Software Engineering and Measurement*, 09 2014.

GAROUSI, Vahid; MäNTYLä, Mika V. When and what to automate in software testing? a multi-vocal literature review. *Information and Software Technology*, v. 76, p. 92–117, 2016. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584916300702>.

GELLER, D.P. Software reliability: Principles and practices. *Proceedings of the IEEE*, v. 66, n. 3, p. 363–364, 1978.

GITHUB. *GitHub About*. 2023. Acessado em 2 de julho de 2023. Disponível em: <https://github.com/about>.

GOUSIOS, Georgios; PINZGER, Martin; DEURSEN, Arie van. An exploratory study of the pull-based software development model. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 345–355. ISBN 9781450327565.

HAMILL, Paul. *Unit Test Frameworks*. [S.l.]: First Edition : O'Reilly Media, 2004.

HARS, A.; OU, S. Working for free? - Motivations of participating in open source projects. *Proceedings of the Hawaii International Conference on System Sciences*, v. 00, n. c, p. 163, 2001. ISSN 10603425.

HASSAN, Ahmed E. The road ahead for mining software repositories. In: *2008 Frontiers of Software Maintenance*. [S.l.: s.n.], 2008. p. 48–57.

HAUGSET, Børge; HANSSEN, Geir Kjetil. Automated acceptance testing: A literature review and an industrial case study. In: *Agile 2008 Conference*. [S.l.: s.n.], 2008. p. 27–38.

HERZBERG, Frederick; MAUSNER, Bernard; SNYDERMAN, Barbara. *The motivation to work, 2nd ed.* Oxford, England: John Wiley, 1959. xv, 157–xv, 157 p.

HERZIG, Kim; GREILER, Michaela; CZERWONKA, Jacek; MURPHY, Brendan. The art of testing less without sacrificing quality. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.: s.n.], 2015. v. 1, p. 483–493.

HERZIG, Kim; JUST, Sascha; ZELLER, Andreas. It's not a bug, it's a feature: How misclassification impacts bug prediction. In: *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2013. p. 392–401.

HINDLE, Abram; GERMAN, Daniel M.; HOLT, Ric. What do large commits tell us? a taxonomical study of large commits. In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2008. (MSR '08), p. 99–108. ISBN 9781605580241. Disponível em: <https://doi.org/10.1145/1370750.1370773>.

HOODA, Itti; CHHILLAR, Rajender Singh. Software test process, testing types and techniques. *International Journal of Computer Applications*, Citeseer, v. 111, n. 13, 2015.

I, Samoladas; G, Gousios; D, Spinellis; I., Stamelos. The SQO-OSS quality model: measurement based open source software evaluation. Open source development, communities and quality. *The International Federation for Information Processing IFIP*, v. 275, p. 237–48, 2008.

IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, p. 1–84, 1990.

KALLIS, Rafael; Di Sorbo, Andrea; CANFORA, Gerardo; PANICHELLA, Sebastiano. Predicting issue types on github. *Science of Computer Programming*, v. 205, p. 102598, 2021. ISSN 0167-6423. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642320302069>.

KARG, Lars M; GROTTKE, Michael; BECKHAUS, Arne. A systematic literature review of software quality cost research. *Journal of Systems and Software*, Elsevier, v. 84, n. 3, p. 415–427, 2011.

KARHU, Katja; REPO, Tiina; TAIPALE, Ossi; SMOLANDER, Kari. Empirical observations on software testing automation. In: *2009 International Conference on Software Testing Verification and Validation*. [S.l.: s.n.], 2009. p. 201–209.

KHANJANI, Atieh; SULAIMAN, Riza. The process of quality assurance under open source software development. In: *2011 IEEE Symposium on Computers Informatics*. [S.l.: s.n.], 2011. p. 548–552.

KOCHHAR, Pavneet Singh; BISSYANDé, Tegawendé F.; LO, David; JIANG, Lingxiao. Adoption of software testing in open source projects - A preliminary study on 50,000 projects. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, p. 353–356, 2013. ISSN 15345351.

KOCHHAR, Pavneet Singh; BISSYANDé, Tegawendé F.; LO, David; JIANG, Lingxiao. An empirical study of adoption of software testing in open source projects. In: *2013 13th International Conference on Quality Software*. [S.l.: s.n.], 2013. p. 103–112.

KRASNER, Herb. *From problem to solutions*. [S.l.: s.n.], 2022.

KUMAR, Divya; MISHRA, K.K. The impacts of test automation on software's cost, quality and time to market. *Procedia Computer Science*, v. 79, p. 8–15, 2016. ISSN 1877-0509. Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050916001277>.

LAMKANFI, Ahmed; DEMEYER, Serge; GIGER, Emanuel; GOETHALS, Bart. Predicting the severity of a reported bug. In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. [S.l.: s.n.], 2010. p. 1–10.

LEE, Amanda; CARVER, Jeffrey C.; BOSU, Amiangshu. Understanding the Impressions, Motivations, and Barriers of One Time Code Contributors to FLOSS Projects: A Survey. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, IEEE, p. 187–197, 2017.

LEE, Sang-Yong Tom; KIM, Hee-Woong; GUPTA, Sumeet. Measuring open source software success. *Omega*, v. 37, n. 2, p. 426–438, 2009. ISSN 0305-0483. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0305048307000898>.

LEUNG, H.K.N.; WHITE, L. Insights into regression testing (software testing). In: *Proceedings. Conference on Software Maintenance - 1989*. [S.l.: s.n.], 1989. p. 60–69.

LEUNG, H.K.N.; WHITE, L. A study of integration testing and software regression at the integration level. In: *Proceedings. Conference on Software Maintenance 1990*. [S.l.: s.n.], 1990. p. 290–301.

LEUNG, Hareton K.N.; WONG, Peter W.L. A study of user acceptance tests. *Software Quality Journal*, v. 6, n. 2, p. 137–149, Jun 1997. ISSN 1573-1367. Disponível em: <https://doi.org/10.1023/A:1018503800709>.

MAKADY, Soha; WALKER, Robert J. Test code reuse from oss: Current and future challenges. In: *Proceedings of the 3rd Africa and Middle East Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (AMECSE '17), p. 31–36. ISBN 9781450355124. Disponível em: <https://doi.org/10.1145/3178298.3178305>.

MALEKZADEH, Mehdi; AINON, Raja Noor. An automatic test case generator for testing safety-critical software systems. In: *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*. [S.l.: s.n.], 2010. v. 1, p. 163–167.

MICHLMAYR, Martin; HUNT, Francis; PROBERT, David. Quality practices and problems in free software projects. In: *Proceedings of the first international conference on open source systems*. [S.l.: s.n.], 2005. p. 24–28.

MORASCA, Sandro; TAIBI, Davide; TOSI, Davide. Oss-tmm: Guidelines for improving the testing process of open source software. *Int. J. Open Source Softw. Process.*, IGI Global, USA, v. 3, n. 2, p. 1–22, abr. 2011. ISSN 1942-3926.

MORI, Allan. Anomaly analyses to guide software testing activity. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. [S.l.: s.n.], 2020. p. 427–429.

MYERS, Glenford J. *The Art of Software Testing*. [S.l.]: Wiley, 1979.

NGUYEN-DUC, Anh; CRUZES, Daniela S.; CONRADI, Reidar. The impact of global dispersion on coordination, team performance and software quality-A systematic literature review. *Information and Software Technology*, Elsevier B.V., v. 57, n. 1, p. 277–294, 2015. ISSN 09505849.

PFLEEGER, Shari Lawrence. *Engenharia de Software: Teoria e Prática*. [S.l.]: Second Edition, Pearson, 2004.

PHAM, Raphael; SINGER, Leif; LISKIN, Olga; FILHO, Fernando Figueira; SCHNEIDER, Kurt. Creating a shared understanding of testing culture on a social coding site. In: *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2013. p. 112–121.

PINTO, Gustavo; STEINMACHER, Igor; GEROSA, Marco Aurélio. More common than you think: An in-depth study of casual contributors. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.: s.n.], 2016. v. 1, p. 112–123.

PRESSMAN, Roger S. *Engenharia de software*. [S.l.]: AMGH, 2021.

RAFI, Dudekula Mohammad; MOSES, Katam Reddy Kiran; PETERSEN, Kai; MäNTYLä, Mika V. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: *2012 7th International Workshop on Automation of Software Test (AST)*. [S.l.: s.n.], 2012. p. 36–42.

RIANSYAH, Muh. What motivate software tester ? A Systematic Literature Review. n. November, p. 1–5, 2015.

RIGBY, Peter; CLEARY, Brendan; PAINCHAUD, Frederic; STOREY, Margaret-Anne; GERMAN, Daniel. Contemporary peer review in action: Lessons from open source development. *IEEE Softw.*, IEEE Computer Society Press, Washington, DC, USA, v. 29, n. 6, p. 56–61, nov. 2012. ISSN 0740-7459.

ROTHERMEL, G.; HARROLD, M.J. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, v. 22, n. 8, p. 529–551, 1996.

ROZINAT, A.; JONG, I. S. M. de; GÜNTHER, C. W.; AALST, W. M. P. van der. Process mining applied to the test process of wafer scanners in asml. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, v. 39, n. 4, p. 474–479, 2009.

RUNESON, P. A survey of unit testing practices. *IEEE Software*, v. 23, n. 4, p. 22–29, 2006.

RYAN, Richard M.; DECI, Edward L. Intrinsic and Extrinsic Motivations: Classic Definitions and New Directions. *Contemporary Educational Psychology*, v. 25, n. 1, p. 54–67, 2000. ISSN 0361476X.

SANTOS, Ronnie Edson De Souza; MAGALHAES, Cleyton Vanut Cordeiro De; CORREIA-NETO, Jorge Da Silva; SILVA, Fabio Queda Bueno Da; CAPRETZ, Luiz Fernando; SOUZA, Rodrigo. Would You Like to Motivate Software Testers? Ask Them How. *International Symposium on Empirical Software Engineering and Measurement*, v. 2017-Novem, p. 95–104, 2017. ISSN 19493789.

SCACCHI, Walt; JENSEN, Chris. *Open Source Software Development*. [S.l.]: Institute for Software Research, 2012.

SOMMERVILLE, Ian. *Engenharia de Software*. [S.l.]: 9. ed. — São Paulo : Pearson Prentice Hall, 2011.

STEINMACHER, Igor; CONTE, Tayana Uchôa; GEROSA, Marco Aurélio; REDMILES, David F. Social barriers faced by newcomers placing their first contribution in open source software projects. *CSCW 2015 - Proceedings of the 2015 ACM International Conference on Computer-Supported Cooperative Work and Social Computing*, n. October 2014, p. 1379–1392, 2015.

TERRAGNI, Valerio; SALZA, Pasquale; PEZZè, Mauro. Measuring software testability modulo test quality. In: *Proceedings of the 28th International Conference on Program Comprehension*. New York, NY, USA: Association for Computing Machinery, 2020. (ICPC '20), p. 241–251. ISBN 9781450379588.

THOMPSON, Christopher. Large-Scale Analysis of Modern Code Review Practices and Software Security in Open Source Software. *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, p. 83–92, 2017.

van der Aalst, W.M.P.; REIJERS, H.A.; WEIJTERS, A.J.M.M.; van Dongen, B.F.; Alves de Medeiros, A.K.; SONG, M.; VERBEEK, H.M.W. Business process mining: An industrial application. *Information Systems*, v. 32, n. 5, p. 713–732, 2007. ISSN 0306-4379. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0306437906000305>.

ZAMPETTI, Fiorella; BAVOTA, Gabriele; CANFORA, Gerardo; PENTA, Massimiliano Di. A study on the interplay between pull request review and continuous integration builds. In: *2019 IEEE 26th*

*International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2019. p. 38–48.

ZHANG, Xunhui; YU, Yue; GOUSIOS, Georgios; RASTOGI, Ayushi. Pull request decisions explained: An empirical overview. *IEEE Transactions on Software Engineering*, v. 49, n. 2, p. 849–871, 2023.

ZHAO, Luyin; ELBAUM, Sebastian. Quality assurance under the open source development model. *Journal of Systems and Software*, v. 66, n. 1, p. 65–75, 2003. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S016412120200064X>.