Análise do Impacto do Tamanho de Componentes React na Qualidade do Software em Repositorios no Github

Augusto Baldiotti Mendonça Alvares[1]

[1]Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais (PUCMG)
Belo Horizonte – MG – Brasil

abmalvares@sga.pucminas.br

Abstract. *React is widely used in web development, yet there remains a gap regarding how the size and structure of its components impact software quality. This study conducted a large-scale analysis of 30,585 public GitHub repositories, selecting 961 relevant projects. Structural metrics (LOC, cyclomatic complexity, and number of functions/props) and code smells were extracted using SonarQube and ReactSniffer. The results indicate that the number of props is the factor most strongly associated with higher code smell severity, followed by component size and complexity. Structural patterns signaling increased need for review and refactoring were also identified. Additionally, the analysis revealed low adoption of automated tests, highlighting weaknesses in testability. These findings support practical guidelines and point to directions for causal studies and improvements in test coverage.*

Resumo. *O React é amplamente utilizado no desenvolvimento web, mas ainda existe uma lacuna quanto ao impacto do tamanho e da estrutura de seus componentes na qualidade do software. Este estudo analisou, em larga escala, 30.585 repositórios públicos do GitHub, selecionando 961 projetos relevantes. Foram extraídas métricas estruturais (LOC, complexidade ciclomática e número de funções/props) e code smells com SonarQube e ReactSniffer. Os resultados indicam que o número de props é o fator mais associado à maior gravidade de smells, seguido pelo tamanho e pela complexidade dos componentes. Também foram identificados padrões estruturais que sinalizam maior necessidade de revisão e refatoração. A análise revelou ainda baixa adoção de testes automatizados, evidenciando fragilidades de testabilidade. Esses achados subsidiam diretrizes práticas e apontam caminhos para estudos causais e ampliação da cobertura de testes.*

Bacharelado em Engenharia de *Software* - PUC Minas
Trabalho de Conclusão de Curso (TCC)

Orientador de conteúdo (TCC I): João Pedro Batisteli - jp.batisteli@hotmail.com
Orientador de conteúdo (TCC I): Joana Souza - joanasouza@pucminas.br
Orientador de conteúdo (TCC I): Leonardo Vilela - leonardocardoso@pucminas.br
Orientador acadêmico (TCC I): Cleiton Tavares - cleitontavares@pucminas.br
Orientador do TCC II: Ramon Lacerda - ramonmarques@pucminas.br

Belo Horizonte, 12 de Dezembro de 2025.

1. Introdução

O *React* tem se consolidado como uma das principais bibliotecas para desenvolvimento *Front-end*, sendo amplamente utilizado por desenvolvedores para criar interfaces de usuário modernas e interativas [Stack Overflow 2024]. Por meio de sua arquitetura baseada em componentes independentes e reutilizáveis, o *React* facilita a modularização e organização de aplicações e impacta atributos fundamentais de qualidade de *software*, como manutenibilidade, confiabilidade e testabilidade. [Ferreira and Valente 2023].

Apesar de o *React* ser uma das tecnologias mais utilizadas no desenvolvimento web, com 41,6% dos desenvolvedores profissionais e 36,6% dos iniciantes relatando trabalho extensivo com ele no último ano e expressando interesse em continuar [Stack Overflow 2024], há poucos estudos na literatura que aprofundam sobre como o tamanho de componentes afeta aspectos como manutenibilidade, testabilidade e legibilidade [Ferreira and Valente 2023]. Práticas inadequadas de *design*, conhecidas como *code smells*, e *anti-patterns* são apontadas como fatores que prejudicam a qualidade do *software* [Saboury et al. 2017]. Embora pesquisas anteriores tenham explorado *code smells* em linguagens como *JavaScript* e *TypeScript* [Bogner and Merkel 2022], faltam estudos específicos sobre os impactos do tamanho dos componentes em projetos *React*. Dito isso essa pesquisa tem como problema central **a lacuna de evidências empíricas sobre como o tamanho e a estrutura de componentes *React* se relacionam com métricas de qualidade como manutenibilidade e testabilidade** Essa ausência de evidências pode dificultar que desenvolvedores tomem decisões informadas durante o *design* e refatoração de componentes, impactando potencialmente a criação de sistemas mais robustos e sustentáveis.

Compreender como o tamanho dos componentes *React* influencia a qualidade do *software* é essencial para fornecer diretrizes baseadas em evidências para decisões de *design* e refatoração de componentes. Estudos mostram que *code smells*, especialmente aqueles relacionados ao tamanho excessivo de componentes, podem comprometer a confiabilidade e dificultar a evolução do *software* [Saboury et al. 2017]. Além disso, o *React*, amplamente utilizado por desenvolvedores [Stack Overflow 2024], carece de estudos que relacionem características estruturais dos componentes com métricas de qualidade, como manutenibilidade e testabilidade [Ferreira and Valente 2023].

Dado o que foi exposto, o objetivo geral desta pesquisa é, **analisar como características estruturais de componentes *React* desenvolvidos em *TypeScript*, em projetos de código aberto no *GitHub*, estão conectadas a métricas de qualidade de *software***. Para alcançar esse objetivo, serão abordados os seguintes aspectos específicos: (1) identificar padrões de *code smells* relacionados ao tamanho e à estrutura dos componentes *React*, considerando métricas como LOC (*Lines of Code*), número de funções e *hooks* utilizados; (2) investigar o impacto da estrutura dos componentes na cobertura de testes, verificando se atributos como número de funções e *hooks* dificultam a criação de casos de teste; e (3) analisar a relação entre a complexidade ciclomática e as métricas de manutenibilidade como *Maintainability Rating* e *Technical Debt*, verificando se componentes mais complexos apresentam pior manutenibilidade.

Os resultados desta pesquisa têm como objetivo fornecer evidências empíricas que auxiliem desenvolvedores na criação de sistemas mais organizados, escaláveis e testáveis, baseados em *React*. Essas evidências serão fundamentadas na análise de métricas estrutu-

rais (como *LOC*, número de funções e *hooks*) e na identificação de *code smells* em projetos de código aberto no *GitHub*. Espera-se que as descobertas promovam recomendações para o *design* e refatoração de componentes, permitindo a redução de *code smells* e o aumento da aderência a boas práticas de desenvolvimento. Assim, promovendo uma maior conscientização sobre boas práticas de *design* e refatoração, além de fortalecer a adoção de padrões consistentes na criação de sistemas.

As demais seções deste trabalho organizam-se da seguinte maneira: a Seção 2 apresenta a fundamentação teórica, aprofundando conceitos sobre qualidade de *software*, princípios de *design* e características estruturais de componentes *React*. Em seguida, a Seção 3 discute trabalhos relacionados, analisando pesquisas relevantes que abordam *code smells* e métricas de qualidade. A Seção 4 detalha os materiais e métodos empregados neste estudo, destacando as etapas e ferramentas utilizadas na condução da pesquisa. Ademais, a Seção 5 apresenta os resultados obtidos, detalhando a coleta, o processamento dos dados e a geração de gráficos e tabelas. A Seção 6 discute esses resultados e aborda as ameaças à validade. Por fim, a Seção 7 apresenta as conclusões e perspectivas para trabalhos futuros.

2. Fundamentação Teórica

Esta seção apresenta os conceitos e fundamentos que servem de base para o desenvolvimento deste estudo.

2.1. Qualidade de *Software*

A qualidade de *software* é um aspecto essencial no desenvolvimento de sistemas, pois impacta diretamente atributos como confiabilidade, manutenibilidade e eficiência. O *SWEBOK* fornece um referencial consolidado sobre a definição desses atributos e sobre categorias de métricas apropriadas para avaliá-los, servindo como suporte teórico para a seleção de indicadores adotados neste trabalho [Washizaki 2024]. De acordo com Bogner e Merkel (2022), a qualidade também está relacionada à capacidade do código de evitar *bugs* e facilitar sua resolução. Assim, métricas como densidade de *code smells* e complexidade cognitiva se apresentam como medidas coerentes para investigar a facilidade de compreensão e manutenção do código em componentes React [Tufano et al. 2015].

Essas métricas permitem uma análise objetiva da qualidade estrutural do *software*, auxiliando na identificação de problemas que podem comprometer atributos como confiabilidade e manutenibilidade. A Complexidade Ciclomática avalia a quantidade de caminhos independentes no fluxo de controle do código, enquanto as Linhas de Código (*LOC*) fornecem uma medida do tamanho do sistema, ambos servindo como indicadores de esforço necessário para manutenção e evolução [Kataieva and Nemec 2025].

No contexto deste trabalho, a análise de métricas como *LOC* e Complexidade Ciclômica será essencial para avaliar como características estruturais de componentes *React* influenciam atributos de qualidade, como manutenibilidade e testabilidade. A combinação dessas métricas permitirá identificar padrões problemáticos e propor diretrizes baseadas em evidências para melhorar a organização e escalabilidade dos sistemas.

2.2. *Code Smell*

Os *code smells* são indicadores de problemas potenciais no *design* ou na implementação de um *software*, que podem comprometer sua manutenibilidade, compreensibilidade e

evolução ao longo do tempo [Washizaki 2024]. Embora não sejam erros funcionais, esses problemas representam violações de boas práticas de *design* e podem aumentar os riscos de falhas futuras [Fowler et al. 1999, Tufano et al. 2015]. Em geral, *code smells* são caracterizados por estruturas de código que dificultam modificações, reduzem a coesão ou introduzem acoplamentos desnecessários.

No contexto de *React*, Ferreira e Valente [Ferreira and Valente 2023] catalogam *smells* como *Large Component* e *Too Many Props*, associados a acúmulo de responsabilidades e excesso de propriedades, que prejudicam compreensão, testes e modularidade. A detecção pode ser automatizada por ferramentas como o *ReactSniffer* [Ferreira et al. 2024], evidências indicam que arquivos com *smells* têm maior risco de falhas [Saboury et al. 2017, Jafari et al.], o que reforça a relevância de sua identificação.

Além disso, este estudo utiliza a métrica de gravidade (*Info*=1, *Minor*=2, *Major*=3, *Critical*=4, *Blocker*=5) atribuindo a gravidade de cada arquivo corresponde à média dos valores dos *smells* detectados, permitindo relacionar a criticidade dos *smells* às métricas estruturais analisadas.

2.3. Biblioteca *React*

A biblioteca *React* é declarativa e baseada em componentes, favorecendo composição, isolamento de estado e reutilização. Essa modularidade depende de escolhas de granularidade, passagem de *props* e particionamento de lógica, que influenciam acoplamento, compreensibilidade e testabilidade. Em aplicações componentizadas, métricas e visualizações estruturais ajudam a localizar pontos frágeis e oportunidades de refatoração [Tarner et al. 2021]. Além disso, decisões arquiteturais e de padrões introduzem *trade-offs* de desempenho em JavaScript, com impacto direto na qualidade [Selakovic and Pradel 2016, Gomes et al. 2023].

No ecossistema JavaScript, métricas estruturais (por exemplo, LOC, complexidade ciclomática, número de funções e de *props*) sinalizam facilidade de evolução e manutenção, mas *smells* emergem e persistem, degradando a qualidade quando não tratados [Tufano et al. 2015]. Em projetos JavaScript, sua presença associase a maior propensão a falhas [Saboury et al. 2017]; *dependency smells* e antipadrões assíncronos também prejudicam manutenibilidade e compreensão [Jafari et al. , Turcotte et al. 2022]. Estratégias de detecção para web apoiam inspeções e priorizam refatorações [Aniche 2015]. Em larga escala, TypeScript pode fortalecer robustez e manutenção, exigindo novas práticas de tipagem e *tooling* [Kushwah et al. 2024]. A interpretação cuidadosa de métricas evita vieses [Pantiuchina et al. 2018]; visualizações favorecem a compreensão estrutural [Scarsbrook et al. 2018], e análises estáticas contribuem para estimar esforço e manutenibilidade [Kataieva and Nemec 2025]. Esses elementos sustentam avaliar componentes *React* por métricas estruturais e definir critérios pragmáticos para revisão e refatoração.

3. Trabalhos Relacionados

Nesta seção, são apresentados estudos relacionados que investigam problemas de *design* e práticas inadequadas em aplicações *React*, com foco em características estruturais dos componentes e seu impacto na qualidade do *software*.

Ferreira and Valente (2023) apresentam um estudo empírico sobre refatorações em aplicações baseadas no *framework React*, com o objetivo de identificar operações específicas realizadas por desenvolvedores durante a manutenção e evolução de projetos de código aberto. A partir da análise manual de 320 *commits* em dez projetos populares do *GitHub*, os autores catalogaram 69 tipos distintos de refatoração, incluindo 25 específicos para *React*, 17 adaptações de refatorações tradicionais ao contexto do *React*, 22 refatorações tradicionais e seis específicas para *JavaScript* e *CSS*. Os resultados destacam a importância de práticas como modularização de componentes e *hooks*, remoção de manipulações diretas do *DOM* e migração de componentes de classe para funções, promovendo melhorias na manutenibilidade e desempenho das aplicações. Este trabalho é altamente relevante para a pesquisa proposta, pois fornece uma base sólida de refatorações específicas para *React* que podem ser usadas para mitigar *code smells* relacionados ao tamanho e à estrutura dos componentes, alinhando-se diretamente aos objetivos de investigar como características estruturais impactam métricas de qualidade, como manutenibilidade e testabilidade.

Ferreira et al. (2024) investigam problemas de design em aplicações *web* baseadas no *React*, propondo um catálogo com 12 tipos de *code smells* específicos para esse *framework*. O estudo utilizou uma revisão de literatura cinza e entrevistas com desenvolvedores profissionais para identificar os *smells*, além de desenvolver uma ferramenta chamada *ReactSniffer* para detectá-los automaticamente em projetos *open-source* populares no *GitHub*. Entre os *smells* identificados, destacam-se *"Large Component"*, *"Too Many Props"* e *"Direct DOM Manipulation"*, que podem impactar negativamente atributos como manutenibilidade e testabilidade. A análise histórica realizada pelos autores revelou taxas de remoção variáveis entre os *smells*, sendo *"Large File"* o mais frequentemente corrigido (50,5%) e *"Inheritance Instead of Composition"* o menos (0,9%). Este trabalho contribui para a pesquisa proposta ao oferecer evidências concretas sobre como características estruturais dos componentes *React* influenciam atributos de qualidade, especialmente em aspectos como facilidade de teste e manutenibilidade.

Saboury et al. (2017) conduziram um estudo empírico sobre *code smells* em projetos *JavaScript*, com foco em sua relação com a propensão a falhas e a percepção dos desenvolvedores. O trabalho analisou 537 versões de cinco projetos populares de código aberto, identificando 12 tipos de *code smells* e utilizando análise de sobrevivência para comparar a ocorrência de falhas em arquivos com e sem esses problemas. Os resultados indicaram que arquivos com *code smells* apresentam uma taxa de risco de falhas 65% maior, sendo "*Variable Re-assign*"e "*Assignment in Conditional Statements*"os *smells* mais críticos. Além disso, uma pesquisa com 1.484 desenvolvedores revelou que *smells* como "*Nested Callbacks*"e "*Long Parameter List*"são percebidos como prejudiciais à manutenibilidade e confiabilidade do *software*. Este estudo é relevante para a presente pesquisa, pois destaca como práticas inadequadas de design podem impactar negativamente atributos de qualidade, fornecendo evidências que complementam a análise de características estruturais de componentes *React* no contexto de projetos *open-source*.

Kataieva et al. (2025) propõem uma abordagem para avaliar a qualidade do *software* por meio de métricas de análise de código, focando na identificação de *code smells* e na gestão da complexidade. O estudo introduz novas métricas, como o *Relative Complexity Index* (RCI) e o *Interaction Complexity* (IC), que complementam métricas tradicionais

como *Cyclomatic Complexity* (CC) e *Lines of Code* (LOC). Essas métricas foram implementadas em um *plugin Maven* para projetos *Spring Boot*, permitindo gerar relatórios detalhados sobre complexidade e *code smells*. Os resultados indicaram que as métricas propostas são mais eficazes na identificação de métodos complexos dentro de classes, oferecendo *insights* acionáveis para melhorar a manutenibilidade do código. Este trabalho se relaciona diretamente com o estudo proposto, pois explora a influência de características estruturais do código na qualidade do *software*. Enquanto o foco do artigo está em projetos *Spring Boot*, o presente estudo busca aplicar conceitos semelhantes ao contexto de componentes *React*, investigando como métricas estruturais, como LOC e número de funções, impactam atributos de qualidade como testabilidade e manutenibilidade.

4. Materiais e Métodos

Este trabalho é classificado como uma pesquisa quantitativa, com foco na análise de características estruturais de componentes *React* desenvolvidos em *TypeScript*. A abordagem se justifica pela necessidade de investigar evidências concretas sobre o impacto dessas características na qualidade de *software*, utilizando dados coletados diretamente de repositórios *open-source* no *GitHub* [Ferreira and Valente 2023]. Nesta seção, são apresentadas as etapas realizadas com o intuito de atingir os objetivos propostos para este trabalho.

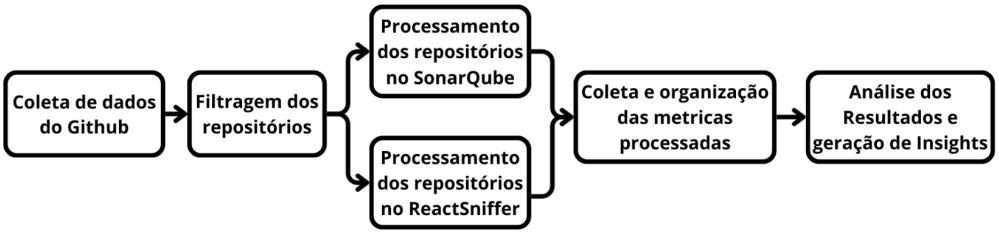

Figura 1. Fluxograma da metodologia

4.1. Ambiente e Ferramentas Consideradas

O ambiente no qual as análises foram realizadas é um computador com processador Intel Core i7 de décima segunda geração, 16 GB de RAM e SSD de 1 TB, rodando o sistema operacional Linux Ubuntu versão 24.04.2 *LTS* (*Long-term support*). Para extração das métricas de qualidade de código foram utilizados o *SonarQube* e o *ReactSniffer*; *scripts* em *JavaScript* facilitaram a coleta e filtragem de dados por meio da API GraphQL do GitHub.

O *ReactSniffer* (v 1.0.19) foi adotado por oferecer uma classificação dedicada de *smells* de componentes React e por possibilitar a extensão da taxonomia utilizada por Ferreira and Valente 2023, o que o torna particularmente adequado para análises centradas em características estruturais desses componentes. O *SonarQube* (edição gratuita) foi escolhido por oferecer métricas consolidadas de manutenibilidade e por permitir a extração automatizada das medidas de interesse. Outras ferramentas foram cogitadas, como o *ESLint* e o *React-Patterns*, porém optou-se pelo *ReactSniffer* e pelo *SonarQube* porque o primeiro oferece classificação específica para componentes React e extensibilidade para incorporar a taxonomia de Ferreira, enquanto o segundo fornece cobertura métrica consolidada que atendem aos objetivos do estudo.

4.2. Coleta e Filtro de Dados

Os dados foram coletados de repositórios *open-source* hospedados no *GitHub*, utilizando a *API GraphQL* implementada em *JavaScript*. Um *script* foi desenvolvido para buscar repositórios que contenham projetos *React* e filtrar aqueles que utilizam *TypeScript* como linguagem principal, além de filtrar por bibliotecas de teste de *Front-end*, como *Jest, Vitest, Testing Library*. Para garantir a relevância dos dados analisados, o *script* rodou em uma *pipeline* durante um prazo de 6 horas, limitado pelo *Github Actions*, para coletar os repositórios que foram filtrados posteriormente. Primeiramente, foi considerado apenas repositórios com histórico ativo de *commits* nos últimos 6 meses, assegurando que os projetos analisados estejam em manutenção ativa. Após isso, foi removido de forma manual, repositórios que utilizem *React* apenas em exemplos, tutoriais, documentação ou testes, como frequentemente ocorre em projetos de estudo ou aprendizado [Ferreira and Valente 2023]. Além disso, também foi removido repositórios que utilizem outras linguagens ou *frameworks* além do *React*, deixando apenas repositórios que utilizem exclusivamente a biblioteca, para evitar a poluição das métricas, o que pode se tornar uma ameaça à validade. Após a aplicação desses critérios, foram selecionados os repositórios restantes para análise mais aprofundada.

4.3. Processamento dos Dados para Geração de Métricas

Com os repositórios selecionados, foi realizado o processamento dos projetos para a geração das métricas necessárias. Um *script* automatizado foi implementado para processar os projetos e coletar as métricas de interesse. O *script* realizou o clone de cada repositório localmente e, em seguida, executou análises utilizando as ferramentas *SonarQube* e *ReactSniffer* [Ferreira et al. 2024]. Essas ferramentas foram responsáveis por analisar o código-fonte e extrair informações relacionadas às métricas estabelecidas como *LOC*, complexidade ciclomática, número de funções e a presença de *code smells* previamente definidos, bem como métricas de manutenibilidade fornecidas pelo *SonarQube* (*Maintainability Rating* e *Technical Debt*). Os resultados dessas análises foram consolidados em uma planilha, mantendo os dados organizados de forma estruturada para facilitar a interpretação e análise estatística posterior.

4.4. Métricas de Avaliação

Para este estudo, as métricas analisadas abrangem diferentes aspectos estruturais dos componentes *React*, permitindo uma visão detalhada de suas características técnicas. A métrica *Lines of Code* (LOC) é utilizada para medir o tamanho dos componentes, contabilizando o número de linhas no código-fonte e refletindo o esforço necessário para compreensão e manutenção [Kataieva and Nemec 2025]. Complementarmente, a Complexidade Ciclomática (CC) mede o número de caminhos independentes no fluxo de controle do código, sendo amplamente reconhecida como indicador da dificuldade de compreensão e do risco de erros [Kataieva and Nemec 2025]. Outras métricas analisadas incluem o número de funções utilizadas nos componentes, importante para avaliar modularidade e reutilização [Ferreira et al. 2024], e o número de *props* recebidas por cada componente. As *props* correspondem às propriedades passadas a um componente *React*, funcionando como parâmetros que configuram sua renderização sem alterar seu estado interno. Adicionalmente, são coletadas *issues*, que funcionam como indicadores de débito técnico e orientam a avaliação do esforço necessário para manutenção.

Além disso, realiza-se a detecção de *code smells* específicos para componentes *React*. O *Large Component* caracteriza-se por componentes excessivamente extensos, seja pelo número elevado de linhas, pela quantidade de *props* recebidas ou ainda pelo acúmulo de atributos internos, que são elementos declarados no corpo do componente, como variáveis ou campos de classe. Já o *Too Many Props* indica componentes que recebem um número excessivo de propriedades, aumentando o acoplamento e dificultando a reutilização. Como indicadores complementares, o *SonarQube* fornece o *Maintainability Rating* (classificação de A a E) e a estimativa de *Dívida Técnica* (em minutos). O *Maintainability Rating* reflete a proporção do esforço de correção em relação ao custo de desenvolvimento (por exemplo, A = dívida até 5%, E = dívida $\geq$ 50%), enquanto a *Dívida Técnica* total corresponde à soma dos minutos estimados para a correção das *issues* reportadas. Juntas, essas métricas e detectores de *smells* suportam a avaliação de atributos como manutenibilidade e testabilidade dos componentes.

5. Apresentação dos Resultados

Esta seção apresenta os resultados alcançados por meio da execução das etapas metodológicas propostas, com ênfase na avaliação de métricas estruturais e na detecção de *code smells* em componentes *React*.

5.1. Coleta e Filtragem de Dados

A coleta de dados foi conduzida por meio de um *script* implementado em *JavaScript* executado no ambiente de integração contínua durante um prazo de 6 horas, limitado pelo *Github Actions*. Esse *script* analisou 30.585 repositórios públicos que utilizam *React* e *TypeScript*, aplicando filtros para excluir repositórios que utilizem *React* apenas em exemplos, tutoriais, documentação ou testes, além também de repositórios que utilizem outras linguagens ou *frameworks* além do *React*, deixando apenas repositórios que utilizem exclusivamente a biblioteca. Após essa etapa, foram selecionados 961 repositórios relevantes para análise mais aprofundada. A próxima fase consiste na execução de análises nas ferramentas *SonarQube* e *ReactSniffer*, ambas configuradas em ambiente local, com o objetivo de coletar métricas estruturais e *code smells*. Os dados obtidos são armazenados em duas planilhas separadas, correspondentes à cada ferramenta, permitindo posterior consolidação e comparação.

5.2. Coleta e Filtragem de Dados

A coleta de dados foi conduzida por meio de um *script* implementado em *JavaScript* executado no ambiente de integração contínua durante um prazo de 6 horas, limitado pelo *GitHub Actions*[1]. Para reduzir vieses de seleção, o script efetuou uma amostragem aleatória de repositórios públicos que declaram uso de *React* e *TypeScript*, totalizando 30.585 repositórios inicialmente analisados. Em seguida aplicaram-se critérios de filtragem para remover repositórios que utilizem *React* apenas em exemplos, tutoriais, documentação ou testes, bem como aqueles que empreguem outras linguagens ou *frameworks* além do *React*, resultando em 961 repositórios considerados relevantes para análise aprofundada. A etapa seguinte envolveu a execução de análises locais com *SonarQube* e *ReactSniffer* para extração de métricas estruturais e detecção de *code smells*.

[1]O script utilizado para a coleta encontra-se listado no Apêndice A.

Os resultados foram registrados em duas planilhas (uma por ferramenta) para posterior consolidação e comparação. Observa-se que, embora a amostragem aleatória busque aumentar representatividade, limitações temporais (janela de 6 horas) e critérios de filtro podem introduzir vieses — esses aspectos são discutidos na seção de Ameaças à Validade.

5.3. Processamento dos Dados para Geração de Métricas

Após a etapa de coleta e filtragem, os dados foram processados localmente por um *script* orquestrador[2] que, para cada repositório, executou o ciclo: clonagem, execução do *SonarQube* e do *ReactSniffer*, extração das métricas selecionadas de cada arquivo analisado, registro nos *CSVs* (*comma-separated values* ou Valores Separados por Vírgulas) específicos de cada ferramenta e, por fim, remoção do repositório clonado. O processamento ocorreu em paralelo, com cerca de 10 *workers* (*threads*) atuando simultaneamente para maximizar a eficiência. Ao todo, foram produzidas 76.731 linhas de dados, consolidadas em duas planilhas (uma por ferramenta). As métricas foram organizadas e comparadas, possibilitando avaliar a hipótese central. Com essa confirmação, iniciou-se a aplicação de métodos estatísticos e a elaboração de gráficos e painéis que sintetizam visualmente as correlações e padrões identificados, servindo como suporte para a interpretação final.

5.4. Geração dos gráficos e tabelas

Nesta subseção são apresentadas as tabelas e gráficos gerados ao longo da análise. A organização dos subtópicos segue diretamente os objetivos específicos do estudo, de modo que cada conjunto de visualizações apoia a investigação de um aspecto particular da qualidade dos componentes. As figuras foram construídas com agregação por faixas, estratégia adotada devido à alta dispersão das variáveis, e a seleção das métricas destacadas baseou-se nos coeficientes de *Pearson* obtidos. Essa estrutura permite apresentar os resultados de forma progressiva e comparável, facilitando a identificação dos padrões discutidos posteriormente.

5.4.1. Identificar padrões de *code smells* relacionados ao tamanho e à estrutura dos componentes *React*

Antes da análise, vale registrar que a Tabela 2 (*LOC* × Métodos x Média de Gravidade de *Smells*) e a Figura 2 (*props* × gravidade) foram construídas a partir de agregações por faixas. Essa estratégia foi adotada porque as variáveis analisadas apresentam amplitudes elevadas e distribuição bastante dispersa, resultado do grande volume de arquivos processados. Assim, o uso de faixas permite agrupar valores semelhantes e reduzir ruído visual, facilitando a identificação de tendências gerais. A seleção das variáveis baseou-se nos coeficientes de *Pearson* obtidos: *props* apresentou a maior correlação com a gravidade (r = 0,6298), seguida por *LOC* (r = 0,4056) e pelo número de métodos (r = 0,3014). Já o número de atributos mostrou relação fraca (r = 0,1771). Por isso, demos destaque a *props* em um gráfico dedicado e concentramos a análise principal em tamanho (*LOC*) e granularidade interna (métodos), mantendo atributos apenas como referência complementar.

[2]Os scripts utilizados para geração e extração de métricas encontram-se listados no Apêndice A.

Tabela 1. Correlação de Pearson entre gravidade e métricas estruturais

	LOC	Props	Atributos	Métodos
Gravidade	0.4056	0.6298	0.1771	0.3014

Média de Gravidade de Smells	*Quantidade de Métodos*							
LOC	a) 0	b) 1	c) 2-3	d) 4-6	e) 7-10	f) 11-14	g) 15+	Total geral
a) 0-49	1	4	4	4	4	4	5	1
b) 50-99	3	3	3	1	4		4	3
c) 100-199	2	2	3	4	4	3	4	3
d) 200-399	4	4	4	4	4	4	4	4
e) 400-699	5	4,5	5	5	5	5	5	5
f) 700-999	5		5	5	5	5	5	5
g) 1000+	5					5	5	5
Total geral	**3**	**3**	**3**	**4**	**4**	**4**	**5**	**3**

Tabela 2. *LOC* × Métodos × Média de Gravidade de *Smells*

Na Tabela 2 (*LOC* × Métodos x Média de Gravidade de *Smells*), observa-se que a média da gravidade dos *smells* aumenta conforme o componente cresce e ganha mais métodos, alinhado às correlações r(gravidade, *LOC*) = 0,41 e r(gravidade, métodos) = 0,30. Componentes pequenos e sem funções quase sempre apresentam baixa gravidade, mas a inclusão de muitas funções faz a gravidade subir rapidamente (chegando a 5 mesmo em arquivos curtos) o que sugere acúmulo de responsabilidades. Entre 200–399 *LOC*, a gravidade fica próxima de 4 independentemente do número de métodos, acima de 400 *LOC*, estabiliza entre 4,5 e 5, e, a partir de 700+, mantém-se em 5, padrão típico do *smell Large Component*. Pelo eixo dos métodos, valores de 4 funções já levam a gravidade ao patamar 4, e 15+ funções a elevam para 5. Pequenas variações por faixa não alteram essa tendência geral.

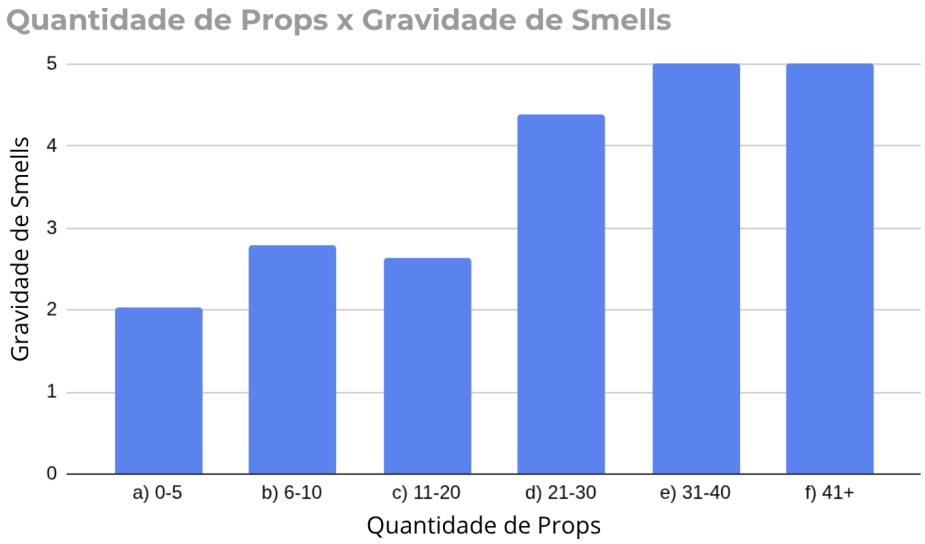

Figura 2. Gráfico Quantidade de *Props* × Gravidade de Smells

Na Figura 2 (Quantidade de *props* × Gravidade de Smells), observa-se uma relação

mais sensível entre as variáveis. A gravidade começa próxima de 2 para componentes com 0–5 *props*, cresce para em torno de 2,8 em 6–10 e retorna a aproximadamente 2,63 em 11–20. A partir daí, há um salto expressivo: 4,37 para 21–30 *props* e gravidade máxima (5) já no intervalo 31–40, permanecendo nesse valor em 41+. Com correlação de *Pearson*, r = 0,63, esse comportamento em "degraus" descreve o *smell Too Many Props* e indicam limiares práticos: revisão a partir de 21–30 *props* e criticidade em 31+ (especialmente 41+). Esses padrões contribuem diretamente para o objetivo de identificar *smells* relacionados ao tamanho e à estrutura dos componentes.

5.4.2. Investigação o impacto da estrutura dos componentes na cobertura de testes

Apesar da relevância do objetivo de investigar a relação entre a estrutura dos componentes e a cobertura de testes, esta análise não pôde ser realizada de forma consistente. Durante a execução das análises, observou-se que, entre os 961 repositórios coletados, apenas 32 arquivos, de 3 repositórios diferentes, apresentavam cobertura de testes superior a 0%. Essa baixa incidência impossibilitou a obtenção de dados estatisticamente significativos, inviabilizando correlações entre métricas estruturais, como número de funções, e a criação de casos de teste. Tal resultado evidencia a escassez de práticas de teste automatizado em projetos *React open-source*, reforçando a necessidade de maior atenção da comunidade a esse aspecto.

5.4.3. Analisar a relação entre a complexidade ciclomática e as métricas de manutenibilidade *Maintainability Rating* e *Technical Debt*

Antes da análise, a escolha de focar a Complexidade Ciclomática (CC) versus débito técnico (minutos e por 100 *LOC*), *Maintainability Rating* e *issues* de manutenibilidade foi guiada pelos coeficientes de *Pearson* como mostrado na Tabela 3.

Tabela 3. Correlação entre métricas e gravidade dos componentes

	Débito Técnico (à cada 100 LOC)	**Débito Técnico (em Minutos)**	**Maintainability Rating (1–5)**	**Open Issues**	**Total Issues**
CC	0,005082	0,442596	-0,008755	0,499820	0,487290

No conjunto completo: *CC* correlaciona moderadamente com Débito Técnico (em Minutos) ($r \approx 0,44$) e com *issues* (*open issues* $r \approx 0,50$ e *total issues* $r \approx 0,49$), enquanto mostra correlação praticamente nula com *Maintainability Rating* ($r \approx -0,009$) e com Débito Técnico (à cada 100 *LOC*) ($r \approx 0,005$). Por isso, priorizamos leituras em termos de esforço absoluto (minutos) e contagem de *issues* como indicadores mais sensíveis de manutenibilidade ao nível de componente.

A análise por faixas de *CC* revela um comportamento por etapas. Até *CC*=10, o débito técnico médio é praticamente nulo. Na faixa 11–30 surge um débito inicial (5 min), que cresce para 19 min em 31–100, sobe para 84 min em 101–300 e chega a 373 min em 301–1000. Normalizando por tamanho (min/100 *LOC*), essa escalada também aparece após *CC*>10, com valores de 4,1, 7,85, 13,17 e 24,89. A queda acima de *CC*>1000

decorre do tamanho muito reduzido da amostra (1 arquivo). Já o *Maintainability Rating* permanece em 1 em todas as faixas, mostrando que essa métrica pouco distingue arquivos mais complexos enquanto o débito relativo continua abaixo dos limites do *SonarQube*. No geral, a tabela indica um ponto de inflexão próximo de *CC*=11 e um aumento progressivo de esforço conforme a complexidade cresce.

CC	Média de Débito Técnico (em Minutos)	Média de Débito Técnico (à cada 100 LOC)	Média de Avaliação de Manutenabilidade (1-5)	Número de Arquivos
a) 0	0	0	1	18882
b) 1–3	0	0	1	24194
c) 4–10	0	0	1	16228
d) 11–30	5	4,098360656	1	10127
e) 31–100	19	7,853403141	1	3406
f) 101–300	84	13,16758542	1	330
g) 301–1000	373	24,89270386	1	19
h) >1000	925	5,419816019	1	1
i) n/a	0	0	1	5
Total geral	**0**	**0**	**1**	**73192**

Tabela 4. Complexidade Ciclomatica por Métricas de Manutenibilidade

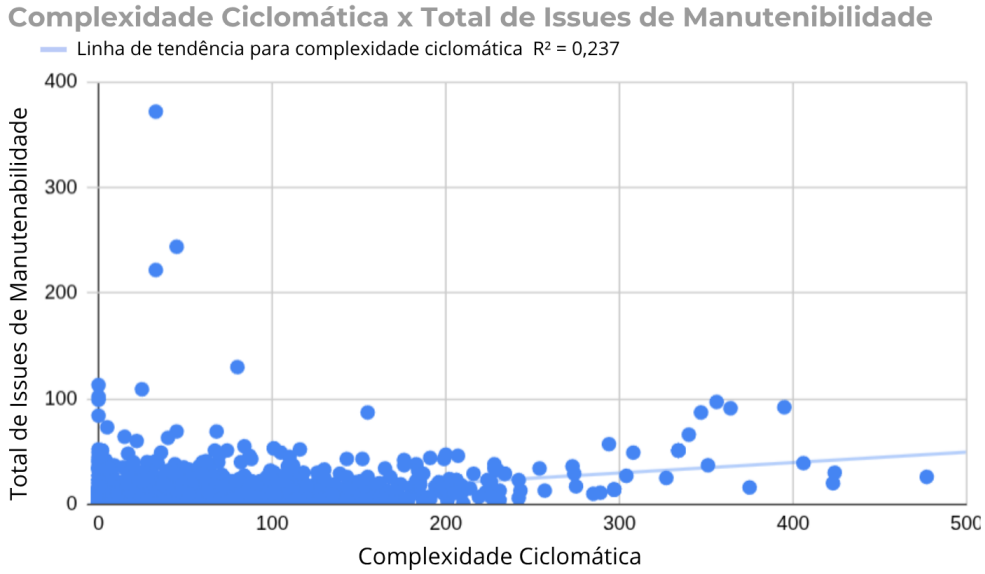

Figura 3. Gráfico de dispersão: *CC* x Total de *Issues* de Manutenabilidade

Os gráficos de dispersão mostram que a relação entre a Figura 4 (CC × Débito Técnico em Minutos) exibe tendência positiva com a linha de tendência R²=0,196 (coerente com a correlação de *Pearson* $r \approx 0,44$), e a Figura 3 (CC × Total de *Issues*) apresenta linha de tendência de R²=0,237 ($r \approx 0,49$), isto é, a complexidade explica cerca de 20–24% da variância nessas métricas. A correlação praticamente nula entre CC e Débito Técnico à (cada 100 LOC) sinaliza que, globalmente, a relação é não linear e sofre efeito-piso (muitos arquivos com *CC* baixo e dívida zero), o que dilui a associação linear, nos agregados por faixa, porém, o crescimento por degraus após $CC > 10$ é nítido.

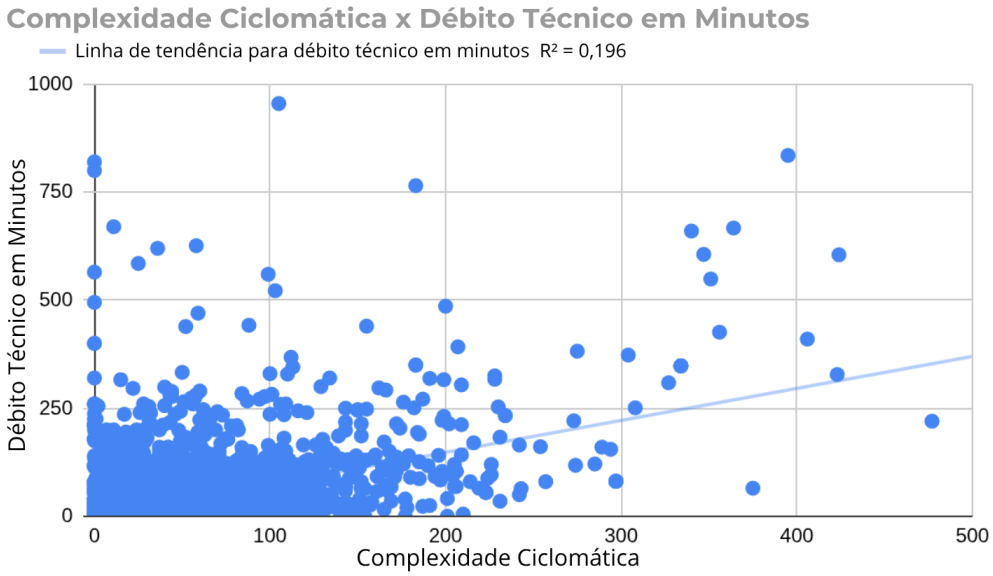

Figura 4. Gráfico de dispersão: *CC* x Débito Técnico em Minutos

6. Discussão dos Resultados e Ameaça à Validade

6.1. Discussão dos Resultados

Embora o efeito do tamanho seja conhecido, nossos resultados revelam pontos de ruptura específicos do ecossistema *React* que não se reduzem a *LOC*. A gravidade cresce em degraus quando o número de *props* atinge 21–30 e torna-se crítica em 31+, caracterizando com precisão o *smell Too Many Props*. Esse comportamento aparece também em faixas intermediárias de *LOC* e número de funções, sugerindo que a pressão cognitiva da passagem de dados e do *prop drilling* atua como fator autônomo no acúmulo de *smells*. Em outras palavras, estruturas idiomáticas de componentes, como composição via *props* e *children*, amplificam riscos qualitativos de forma distinta de módulos tradicionais, em concordância e ampliação de evidências sobre *smells* em aplicações *React* [Ferreira and Valente 2023].

Na manutenibilidade, a escalada observada de esforço a partir de $CC \geq 11$ e o regime crítico em $CC \geq 101$ são consistentes com caminhos de renderização que concentram condicionais aninhadas, expressões ternárias encadeadas e efeitos colaterais em *hooks*, padrões típicos de JSX que elevam *CC* sem crescimento proporcional de *LOC*. Isso ajuda a explicar por que a simples microextração de *helpers* (maior granularidade interna) não reverte a tendência: o acoplamento por fluxo de controle e dados da árvore de renderização permanece. Esse perfil difere do código de domínio *back-end* estudado em trabalhos clássicos e reforça a necessidade de análises sensíveis a componentes baseados em UI [Tarner et al. 2021].

Surpreendentemente, o *Maintainability Rating* manteve-se em A mesmo quando débito técnico e *issues* cresceram com *CC*, evidenciando desalinhamento entre métricas agregadas e sinais locais de degradação em arquivos *React*. Esse descolamento ecoa alertas sobre a percepção equivocada de métricas e pode induzir decisões otimistas sem fundamento [Pantiuchina et al. 2018]. Ao quantificar limiares operacionais (≥ 21 *props*, ≥ 400 *LOC* e, sobretudo, ≥ 101 de *CC*), os dados sugerem priorizar inspeções e refatorações

guiadas por sinais específicos de componentes, em vez de depender de *ratings* globais.

Finalmente, a quase ausência de cobertura de testes — apenas 32 arquivos com cobertura $> 0\%$ em 961 repositórios — indica um gargalo processual que potencializa riscos estruturais. À luz de recomendações consolidadas de verificação e validação [Washizaki 2024] e de catálogos de refatorações aplicáveis a aplicações *React* [Fowler et al. 1999, Ferreira et al. 2024], os limiares identificados funcionam como gatilhos práticos: reduzir *props* via *Context*/*compound components*, quebrar caminhos de renderização complexos e instituir testes de unidade/integração focados nos componentes de maior *CC*. Assim, o estudo não apenas confirma tendências, mas delimita onde e como agir no ecossistema *React*.

6.2. Ameaça à Validade

Nesta seção, discutimos os principais riscos à validade do estudo e as medidas adotadas para mitigá-los. Quanto à validade de construção, há risco de que os *proxies* adotados para tamanho e granularidade (*LOC*, número de funções e *props*) e para manutenibilidade (*Maintainability Rating* e Dívida Técnica do *SonarQube*) não representem integralmente os construtos pretendidos. Em *React*, a relação arquivo–componente é imperfeita (arquivos com múltiplos componentes, *HOCs*, *Server/Client Components*, *CSS-in-JS*), o que pode distorcer contagens e a gravidade de *smells*. Além disso, a estabilidade do *Rating* A mesmo em faixas de alta complexidade sugere baixa sensibilidade do perfil de regras ao nível de arquivo [Pantiuchina et al. 2018]. Para mitigar, padronizamos ambiente e versões, registramos perfis e triangulamos com indicadores de esforço (dívida técnica e *issues*), ainda assim, validações manuais por amostragem e regras específicas (*lint/AST*) fortaleceriam o construto.

No que tange à validade interna, as associações observadas entre estrutura (*LOC*, funções, *props*) e gravidade de *smells*, bem como entre complexidade ciclomática e esforço (dívida e *issues*), são correlacionais e sujeitas a confundidores, como maturidade do repositório, domínio, versão de *React/TypeScript*, práticas de revisão, *monorepos*, *tooling* e políticas de qualidade. A baixa cobertura registrada pode refletir limitação de instrumentação (ausência de execução/relatórios) e não ausência de testes. Mitigamos evitando afirmações causais, tratando cobertura como inconclusiva e adotando interpretações conservadoras como controles adicionais, bloqueio por domínio, *matching* por tamanho/idade e estratificação por *tooling*, que reduziriam vieses.

Quanto à validade externa, a coleta automatizada em janela de 6 horas, com filtros restritivos (projetos ativos, *TypeScript* e uso exclusivo de *React*), aumenta a reprodutibilidade, mas reduz a representatividade do ecossistema ao excluir *monorepos*, integrações comuns e bases privadas corporativas. Há viés temporal (limites de taxa e disponibilidade do *GitHub*) e de plataforma (apenas *GitHub*). Mitigamos documentando integralmente o processo e os critérios, recomendamos ampliar escopo (incluir *JavaScript*, *monorepos* e outras plataformas), considerar projetos corporativos quando possível e conduzir estudos longitudinais.

Por fim, há ameaças à validade de conclusão. O uso predominante de correlação de *Pearson* em dados com *zero-inflation*, heterocedasticidade e possíveis não linearidades pode subestimar ou distorcer associações, a agregação por faixas facilita a leitura, mas pode induzir degraus artificiais. Não houve controle de múltiplas comparações nem inter-

valos de confiança por reamostragem. Como mitigação, reforçamos o caráter associativo dos achados e acompanhamos leituras agregadas. Para trabalhos futuros, sugerimos *Spearman*, regressões robustas/quantílicas, modelos zero-inflados e ajuste por confundidores.

7. Conclusão

Os resultados consolidam o entendimento de que atributos de qualidade como manutenibilidade, testabilidade e confiabilidade são fortemente influenciados por decisões estruturais no nível do componente. Em particular, o aumento do número de *props* e do tamanho (*LOC*) acentua a gravidade de *smells* e sugere acúmulo de responsabilidades, por sua vez, a complexidade ciclomática se reflete em esforço de manutenção e maior incidência de *issues*. A constatação de que *ratings* genéricos podem permanecer estáveis em situações de deterioração local reforça a importância de indicadores mais finos e específicos por arquivo. Somado a isso, a escassez de cobertura de testes observada torna ainda mais necessário disciplinar a estrutura dos componentes para mitigar riscos.

Como encaminhamento, o trabalho recomenda institucionalizar limiares para *props*, *LOC* e *CC* como critérios de qualidade nas esteiras de *CI/CD* (Integração Contínua / Implantação Contínua), estabelecer gates de revisão com foco em modularização e composição, e priorizar métricas de esforço e telemetria de *issues* para guiar a gestão de dívida técnica [Aniche 2015]. A adoção de ferramentas como *SonarQube* e *ReactSniffer*, com as tabelas e gráficos que evidenciem faixas de risco, facilita intervenções pontuais e aumenta a transparência sobre o impacto das decisões de design. Complementarmente, os resultados sustentam o uso de limites operacionais específicos para orientar revisão e refatoração: revisão sistemática a partir de 21–30 *props* (crítica em 31+), atenção redobrada para componentes acima de 400 *LOC* e revisão de complexidade a partir de $CC \geq 11$, com níveis de alerta elevados em $CC \geq 31$ e $CC \geq 101$. Com essas medidas, incorporando esses indicadores às *pipelines* e tratando minutos de dívida e contagem de *issues* como sinais primários de saúde ao nível de componente, além de fortalecer a cultura de testes automatizados, as equipes podem transformar evidências em prática, promovendo bases de código mais sustentáveis, previsíveis e seguras em projetos *React*.

Como trabalho futuro, abre-se a possibilidade de investigar o efeito causal de refatorações que reduzam atributos estruturais como número de *props*, *LOC* ou *CC*. Estudos posteriores podem, por exemplo, explorar estratégias que tratem *pull requests* que promovem reduções nessas métricas como potenciais intervenções, comparando seu impacto com arquivos de controle no mesmo repositório e período. Linhas de pesquisa dessa natureza poderiam examinar, sob diferentes configurações de janelas temporais, em que medida tais mudanças estruturais se relacionam com indicadores como dívida técnica, incidência de *bugs* ou estabilidade de *PRs*. Esses caminhos permitiriam avançar na compreensão sobre quando e em quais contextos refatorações estruturais geram benefícios mensuráveis.

Pacote de Replicação

O pacote de replicação deste trabalho encontra-se disponível em:
`https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2025-1-tcci-0393100-pes-augusto-baldiotti`

Referências

Aniche, M. F. (2015). Detection strategies of smells in web software development. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 598–601.

Bogner, J. and Merkel, M. (2022). To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 658–669, New York, NY, USA. Association for Computing Machinery.

Ferreira, F., Borges, H. S., and Valente, M. T. (2024). Refactoring react-based web apps. *Journal of Systems and Software*, 215:112105.

Ferreira, F. and Valente, M. T. (2023). Detecting code smells in react-based web apps. *Information and Software Technology*, 155:107111.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Gomes, F., Soares, P., and Araújo, A. A. (2023). Examining the performance implications of design patterns in front-end web development: A preliminar comparative study with react and vue.js. In *Proceedings of the 29th Brazilian Symposium on Multimedia and the Web*, WebMedia '23, page 255–259, New York, NY, USA. Association for Computing Machinery.

Jafari, A. J., Costa, D. E., Abdalkareem, R., Shihab, E., and Tsantalis, N. Dependency smells in javascript projects. *IEEE Transactions on Software Engineering*, 48, number = 10, pages = 3790–3807, year = 2022, doi = 10.1109/TSE.2021.3106247,.

Kataieva, Y. and Nemec, M. (2025). Determining software quality using code analysis metrics. In *2025 29th International Conference on Information Technology (IT)*, pages 1–4.

Kushwah, S., Bansal, C., Rathore, S., Kate, V., Bargal, D., and Vishwakarma, D. (2024). Typescript: An open-source programming language with options for robust development and large-scale applications. In *2024 International Conference on Advances in Computing Research on Science Engineering and Technology (ACROSET)*, pages 1–5.

Pantiuchina, J., Lanza, M., and Bavota, G. (2018). Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91.

Saboury, A., Musavi, P., Khomh, F., and Antoniol, G. (2017). An empirical study of code smells in javascript projects. In *24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 294–305.

Scarsbrook, J. D., Ko, R. K. L., Rogers, B., and Bainbridge, D. (2018). Metropoljs: Visualizing and debugging large-scale javascript program structure with treemaps. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 389–393.

Selakovic, M. and Pradel, M. (2016). Performance issues and optimizations in javascript: An empirical study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 61–72.

Stack Overflow (2024). Stack overflow developer survey 2024. Accessed: 2025-03-28.

Tarner, H., van den Bongard, D., and Beck, F. (2021). Visually analyzing the structure and code quality of component-based web applications. In *2021 Working Conference on Software Visualization (VISSOFT)*, pages 160–164.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M. D., Lucia, A. D., and Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, pages 403–414.

Turcotte, A., Shah, M. D., Aldrich, M. W., and Tip, F. (2022). Drasync: Identifying and visualizing anti-patterns in asynchronous javascript. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 774–785.

Washizaki, H., editor (2024). *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0*. IEEE Computer Society. Latest edition (SWEBOK v4.0, updated as v4.0a in 2025).

Apêndice

A. Scripts Utilizados

Os scripts utilizados na coleta, processamento e extração de métricas encontram-se disponíveis no repositório da pesquisa. Seguem os links diretos:

Coleta de Dados

- **Script de mineração de repositórios:**
  `https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2025-1-tcci-0393100-pes-augusto-baldiotti/blob/main/Instrumentos/Codigos/1_mine_ts_react_jest.js`

Geração de Métricas

- **Execução do SonarQube:**
  `https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2025-1-tcci-0393100-pes-augusto-baldiotti/blob/main/Instrumentos/Codigos/2a_analyze_with_sonarqube.js`
- **Extração das métricas do SonarQube:**
  `https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2025-1-tcci-0393100-pes-augusto-baldiotti/blob/main/Instrumentos/Codigos/2b_extract_sonar_ts_metrics.js`
- **Execução em lote do ReactSniffer e extração:**
  `https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2025-1-tcci-0393100-pes-augusto-baldiotti/blob/main/Instrumentos/Codigos/3_extract_react_sniffer_batch.js`