UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
Bacharelado em Ciência da Computação

Uma análise exploratória da influência de bad smells e refatorações no volume de comentários

Kleiton Ewerton de Oliveira

JUIZ DE FORA
MARÇO, 2025

Uma análise exploratória da influência de bad smells e refatorações no volume de comentários

Kleiton Ewerton de Oliveira

UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
Bacharelado em Ciência da Computação

Orientador: Gleiph Ghiotto Lima de Menezes
Coorientador: André Luiz de Oliveira

JUIZ DE FORA
MARÇO, 2025

Uma análise exploratória da influência de bad smells e refatorações no volume de comentários

Kleiton Ewerton de Oliveira

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Gleiph Ghiotto Lima de Menezes
Professor Doutor

André Luiz de Oliveira
Professor Doutor

Leonardo Vieira dos Santos Reis
Professor Doutor

Igor de Oliveira Knop
Professor Doutor

JUIZ DE FORA

11 DE MARÇO, 2025

Aos meus amigos e irmãos.

Aos pais, pelo apoio e sustento.

Resumo

Contexto: Ao desenvolver um software, é fundamental a adoção de boas práticas de programação para melhorar a qualidade e facilitar a manutenção dos sistemas. **Motivação**: Na literatura há vários estudos que relataram que excesso de comentários pode indicar a presença de *bad smells* no código, sugerindo que o código precisa ser refatorado para melhorar sua clareza. **Objetivo**: Este projeto de pesquisa teve como objetivo de investigar a relação entre refatorações e *bad smell* e o volume de comentários no código-fonte. **Metodologia**: Para alcançar este objetivo, foi conduzido uma revisão sistemática da literatura, seguido da extração de dados de projetos Java públicos e análise estatística. Foram selecionados projetos com base em critérios como popularidade e atividade. Para análise, foi desenvolvida uma ferramenta para identificar e salvar as refatorações, *bad smell* e comentários. **Resultados**: Os achados indicam que refatorações estão associadas ao aumento de comentários, especialmente de documentação. Além disso, *bad smells* como *God Class* e *Too Many Methods* também impactam o volume de comentários, sugerindo que a documentação é usada para mitigar a complexidade do código. **Conclusão**: O estudo contribui para entender a relação entre refatorações, *bad smells* e comentários, auxiliando no desenvolvimento de ferramentas e estratégias que equilibrem alterações no código e uso de comentários.

Palavras-chave: Refatoração, Comentários de Código, *Bad Smell*, Mineração de Repositórios, Qualidade de Software, Desenvolvimento de Software.

Agradecimentos

Para todos os meus familiares, agradeço pelo estímulo e suporte, especialmente ao meu pai e minha mãe, pois sem o apoio financeiro e emocional deles não teria conquistado o que conquistei.

Aos professores Gleiph e André pela orientação, amizade e principalmente, pela paciência, sem a qual este trabalho não se realizaria.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

Por fim, cabe destacar o papel fundamental da inteligência artificial no desenvolvimento deste trabalho. Com um agradecimento especial à OpenAI e à sua ferramenta, ChatGPT, que desempenhou um papel crucial ao auxiliar nas correções textuais e gramaticais, proporcionando um apoio valioso na elaboração e revisão desta pesquisa, sendo de minha total responsabilidade o seu uso.

"A simplicidade é o último grau de sofisticação'.

Leonardo da Vinci

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

DCC	Departamento de Ciência da Computação
UFJF	Universidade Federal de Juiz de Fora
SCV	Sistema de Controle de Versão
SATD	Self-Admitted Technical Debt (Dívida Técnica Autodeclarada)
GQM	Goals, Questions, and Metrics (Objetivos, Perguntas e Métricas)
IDE	Integrated Development Environment (Ambiente de Desenvolvimento Integrado)
API	Application Programming Interface (Interface de Programação de Aplicações)

1 Introdução

A codificação de um software, visando a qualidade e a manutenibilidade, exige a utilização de boas práticas de programação. No entanto, a literatura demonstra uma lacuna na investigação da conexão entre os fatores que alteram a qualidade do software e os comentários presentes no código. Este estudo busca suprir essa lacuna, examinando a interação entre refatorações e *bad smell* no software e a presença de comentários no código-fonte. A análise dessa relação visa aprimorar as práticas de desenvolvimento e, consequentemente, a qualidade do produto final

1.1 Apresentação do Tema

Comentários de código são elementos críticos na documentação do software, ao fornecerem contexto adicional, explicam decisões de design e auxiliam na compreensão de trechos complexos por outros desenvolvedores. No entanto, como observado por Ibrahim et al. (2012), a prática de atualização de comentários nem sempre acompanha adequadamente as mudanças da evolução do código, o que pode levar à introdução de *bugs* e a uma documentação desatualizada e inconsistente.

1.2 Contextualização

Ao desenvolver um software, é fundamental utiliza-se de boas práticas de programação para melhorar a qualidade e simplificar a manutenção dos sistemas, pro-

longando sua durabilidade. Contudo, a conexão entre os *bad smell*, refatorações e os comentários é pouco abordada na literatura, havendo uma lacuna a ser preenchida. Neste cenário, o objetivo foi examinar a conexão entre os diversos tipos de atributos e os comentários de código, a fim de compreender sua interação e influência na qualidade do software. Explorar tais relações visou aprimorar as práticas de desenvolvimento de software e a qualidade do produto final (GUO et al., 2014).

1.3 Justificativa

A documentação do código-fonte por meio de comentários é uma prática comum na Engenharia de Software, ao contribuir para a clareza, colaboração entre equipes e evolução contínua dos sistemas (PERERA et al., 2023). Contudo, sua negligência pode gerar inconsistências, dificultar a manutenção e comprometer a qualidade do software (CHAROENWET et al., 2024). Além disso, especialistas como Martin Fowler destacam que um excesso de comentários pode indicar a presença de *Code Smells*, sugerindo que o código precisa ser refatorado para melhorar sua clareza e design (FOWLER, 2009). Dessa forma, um código com muitos comentários pode ser um indicativo de problemas estruturais, tornando a base de código menos sustentável a longo prazo (KAUR; KAUR, 2016).

Apesar dos avanços na área, principalmente no que se tange à análise de dívidas técnicas (HALEPMOLLASI; TOSUN, 2024), a interação entre refatorações, *bad smells* e práticas negligenciadas de documentação permanece pouco explorada. Assim, este estudo buscou preencher essa lacuna ao analisar se diferentes atributos do código influenciam o volume e o tipo de comentários adicionados ou removidos.

Ao identificar padrões nessa interação, pretende-se contribuir com diretrizes para o desenvolvimento de ferramentas automatizadas de análise de comentários, além de promover boas práticas para tornar o código mais legível e sustentável. Essa abordagem não somente facilitará o trabalho dos desenvolvedores, mas também poderá influenciar a criação de novos modelos de documentação para códigos refatorados, aprimorando o processo de desenvolvimento de software.

1.4 Objetivos

A seguir, é apresenta as subseções que compõe essa seção de objetivos: subseção 1.4.1, na qual é apresentados os objetivos gerais e a subseção 1.4.2, no quais são apresentados os objetivos específicos.

1.4.1 Gerais

O objetivo principal deste estudo foi analisar os relacionamentos entre refatorações, *bad smells* e o volume de comentários do código-fonte. Principalmente, identificar os tipos de modificações mais comumente associadas à adição e/ou remoção de comentários, realizando uma análise dos possíveis impacto dessas mudanças na estrutura e legibilidade do código.

1.4.2 Objetivos Específicos

Para alcançar o objetivo principal foi exigido conduzir uma análise dos repositórios, para tal, foi necessário o desenvolvimento de um software dividido em duas partes: *i)* análise dos repositórios, no qual é extraído informações sobre os atributos

específicos que estão presentes em cada versão do código e identificar comentários adicionados ou removidos ao longo do tempo separados por seu tipo. *ii)* implementação do algoritmo que visa determinar a relevância estatística e as regras de associação entre a diferença da quantidade de cada tipo de comentário e os tipos de refatorações e *bad smell* ocorridos entre as versões dos *commits* pais e filhos.

A análise foi estruturada conforme o método *Goals, Questions and Metrics (GQM)* (SOLINGEN et al., 2002), detalhado na Tabela 5.1. Esse método fornece um passo a passo para estabelecer metas, formular perguntas e definir métricas para avaliar o sucesso de um projeto ou processo. A abordagem GQM auxilia na definição de objetivos claros, na identificação de perguntas-chave a serem respondidas e no estabelecimento de critérios mensuráveis para a avaliação de desempenho (OLSSON; RUNESON, 2001).

1.5 Metodologia

Este estudo emprega uma abordagem mista, combinando análise da literatura, análise exploratória e estatística, para investigar a correlação entre atributos e o volume de comentários no código-fonte. A metodologia está dividida em cinco etapas principais, cada uma abordada detalhadamente nas subseções a seguir:

- **Trabalhos Relacionados**, no qual é apresentada a revisão de literatura para identificar lacunas e contextualizar a pesquisa.
- **Estruturação da Pesquisa**, que explica como o método GQM foi utilizado para organizar as metas, perguntas e métricas da pesquisa.
- **Coleta e Preparação dos Dados**, que descreve o processo de seleção e

tratamento dos repositórios de código utilizados no estudo.

- **Análise Exploratória**, que visa identificar padrões e tendências entre refatorações e comentários mediante técnicas descritivas.
- **Avaliação dos Resultados**, no qual são aplicadas métricas de desempenho para avaliar os modelos de predição.

Cada uma dessas etapas contribui para uma compreensão mais profunda do impacto das refatorações e *bad smells* no volume de comentários, guiando o desenvolvimento de diretrizes para a documentação de código refatorado.

1.5.1 Trabalhos Relacionados

A revisão da literatura existente sobre comentários de código foi conduzida visando identificar lacunas no conhecimento, trabalhos correlatos e tendências na área. Esse levantamento é essencial para contextualizar a pesquisa, justificando sua relevância e destacando contribuições potenciais. Foram analisados estudos que investigam a relação entre práticas de desenvolvimento, métricas de código e a evolução dos comentários no software.

Os trabalhos selecionados foram examinados considerando suas metodologias, principais resultados e contribuições para o campo de estudo. Essa análise permitiu identificar abordagens relevantes e oportunidades de pesquisa ainda pouco exploradas. A forma detalhada de seleção dos trabalhos é apresentada na Seção 3.1, enquanto a discussão sobre os artigos analisados está na Seção 3.2.

1.5.2 Coleta e preparação dos dados

Para melhorar a extração de dados sobre a evolução de código Java, foi implementada uma metodologia que combina técnicas de mineração de repositórios de software e análise de código-fonte. Essa abordagem visa obter informações mais detalhadas e precisas sobre as refatorações e a evolução dos comentários em projetos Java de código aberto.

Para tanto, foi realizada a seleção de um conjunto representativo de 7 repositórios, considerando critério como popularidade, atividade e tamanho. A partir desses repositórios, são extraídos dados históricos sobre *commits*, incluindo informações detalhadas sobre as refatorações, sendo identificadas por meio da ferramenta *RefactoringMiner* [1], capaz de identificar a ocorrência de mais de 95 tipos distintos de refatorações (TSANTALIS; KETKAR; DIG, 2020), é extraído os *bad smells* utilizando a ferramenta do *PMD* [2], por fim, é extraído a evolução da quantidade de comentário ao longo do tempo utilizando a ferramenta do *JavaParser* [3].

1.5.3 Análise Exploratória

A análise exploratória teve como objetivo identificar padrões e tendências nas relações entre atributos e comentários. Foram calculadas estatísticas descritivas para quantificar a frequência de diferentes tipos de atributos e a quantidade média de comentários por *commit*. Além disso, foram gerados tabelas para auxiliar na interpretação dos dados.

[1]Disponível em: ⟨https://github.com/tsantalis/RefactoringMiner⟩
[2]Disponível em: ⟨https://pmd.github.io/⟩
[3]Disponível em: ⟨https://javaparser.org/⟩

1.5.4 Estruturação da pesquisa

Para garantir a sistematização da pesquisa, foi utilizado o método *Goals, Questions, and Metrics* (SOLINGEN et al., 2002), um modelo estruturado de definição e medição de objetivos em processos de engenharia de software, tendo sua base na organização e a análise de métricas de software em torno de objetivos específicos. Essa abordagem permitirá definir as metas da pesquisa, formular perguntas de pesquisa específicas e estabelecer métricas para avaliar os resultados obtidos, na pesquisa atual sua utilização encontra-se na Tabela 5.1.

2 Fundamentação teórica

Nesta seção são apresentados os conceitos relacionados a sistemas de controle de versão (Seção 2.1), a definição de comentários (Seção 2.2), fatores de qualidade de software (Seção 2.3), refatoração (Seção 2.4) e *bad smells* (Seção 2.5) necessários à compreensão deste trabalho.

2.1 Sistema de controle de versão

Os sistemas de controle de versão (SCV) são ferramentas indispensáveis no desenvolvimento de software, permitindo o gerenciamento de alterações no código, facilitando a colaboração entre equipes e verificando suas alterações ao longo do tempo (SONG et al., 2020). Esses sistemas registram e mantêm o histórico completo de alterações, possibilitando a recuperação de versões anteriores, assim como anotações, explicações e problemas destinadas a eles (SONG et al., 2020).

O funcionamento de um SCV pode ser resumido no ciclo entre o repositório central (*remote*) e o ambiente local (*workspace*), como mostrado na Figura 2.1. Desenvolvedores criam uma cópia local do repositório central usando o comando *clone* e realizam alterações no *workspace*. Após preparar as mudanças com o comando *add*, elas são salvas localmente com o comando *commit*. Para sincronizar com o *remote*, utilizam-se os comandos *pull* (puxar do *remote* para o *workspace*) e *push* (empurrar do *workspace* para o *remote*), garantindo que todas as contribuições

sejam integradas corretamente.

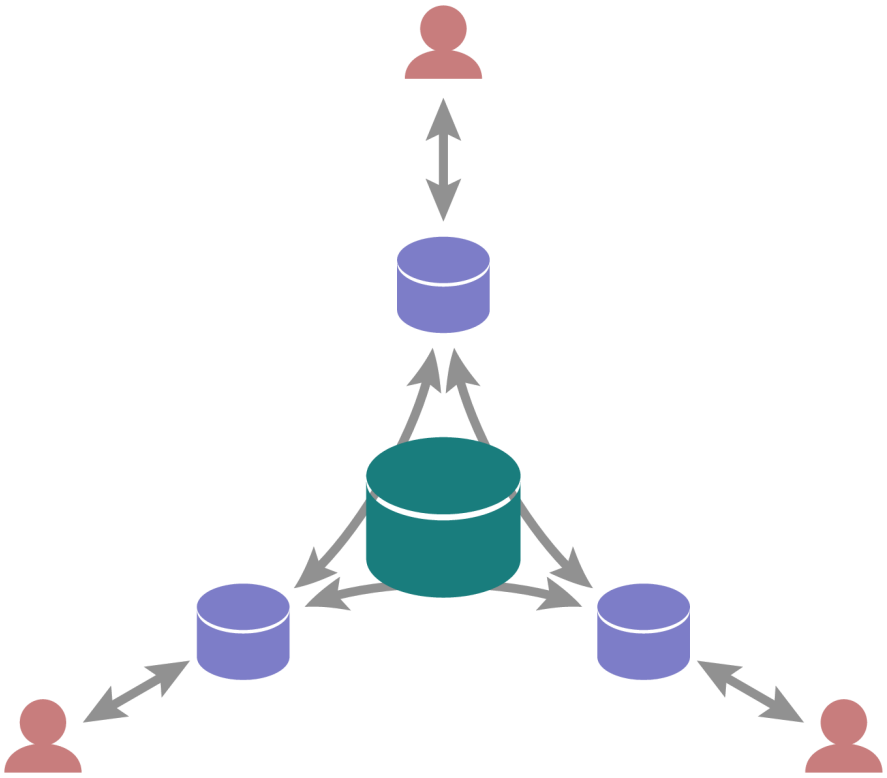

Figura 2.1: Diagrama representativo dos processos do sistema de controle de versões. Disponível em: ⟨https://gertingold.github.io/tools4scicomp/git.html⟩. Acesso em: 10 dez. 2024.

Além do controle de versão garantir a integridade das alterações ele permite realizar extrações de dados de cada versão. Essa mineração de repositórios permite a análise de dados provenientes desses sistemas e outras fontes relacionadas ao desenvolvimento de software (SILVA, 2005). Tal método contribui para identificar padrões e obter informações úteis para melhorar processos e produtos (ALOMAR et al., 2021; CHAROENWET et al., 2024). Por exemplo, a análise de históricos de *commits* pode revelar problemas recorrentes, enquanto logs de problemas ajudam a identificar tendências na ocorrência de *bugs* (ALOMAR et al., 2021; CHARO-

ENWET et al., 2024). Desta forma, os SCVs oferecem um meio estruturado e eficiente para o gerenciamento de código, promovendo colaboração, rastreabilidade e integridade nos projetos de software.

2.2 Comentários

Comentários no código são usados com intuito de melhorar a qualidade do desenvolvimento de software, visando contribuir para a legibilidade, manutenção e colaboração entre desenvolvedores (MARTIN, 2019). Eles documentam aspectos como lógica, entradas e saídas de funções, e facilitam a depuração ao isolar seções problemáticas ou desativar trechos temporariamente (PERERA et al., 2023). Nesta pesquisa, são considerados três tipos principais de comentários:

1. **Comentários de Linha**: Fornecem explicações rápidas sobre linhas específicas do código. Sendo recomendadas para notas breves (MARTIN, 2019). Na Figura 2.2 é demostrado sua utilização para descrever a inicialização de variáveis, método e classe.

```
1  // Represents a library system for managing books
2  public class Library {
3      // List of all books available in the library
4      private List<Book> books;
5
6      // Initializes the library with an empty book list
7      public Library() {
8          books = new ArrayList<>();
9      }
10 }
```

Figura 2.2: Exemplo de comentário de linha.

2. **Comentários de Bloco**: Explicam a funcionalidade de trechos maiores de código, sendo utilizada para oferecer uma visão geral de blocos complexos (MARTIN, 2019), conforme mostrado na Figura 2.3.

```
1 public void adicionarLivro(String titulo, String autor) {
2  /* Creates a new book and adds it to the book collection.
3       The title and author are provided as parameters
4       and used to initialize the data in the book. */
5      Book novoLivro = new Book(titulo, autor);
6      books.add(novoLivro);
7                                ...
8 }
```

Figura 2.3: Exemplo de comentário de bloco.

3. **Comentários de Documentação**: Utilizados para gerar documentação automática, detalhar funções, parâmetros e retornos, auxiliando na criação

de documentação técnica (MARTIN, 2019). Na Figura 2.4, é apresentado um exemplo do seu uso ao explicar o funcionamento do método "*buscarLivrosPorTitulo*".

```
1     /**
2     * Search for books by title.
3     * @param title The title of the book to be sought
4     * @return A list of books that match
5     * The title provided
6     */
7    public List<Book> buscarLivrosPorTitulo(String titulo) {
8        return books.stream()
9              .filter(book -> book.getTitulo()
10             .equalsIgnoreCase(titulo))
11             .collect(Collectors.toList());
12   }
```

Figura 2.4: Exemplo de comentário de documentação.

2.3 Fatores de Qualidade de Software

A qualidade do software é um conceito multifacetado, essencial para o sucesso de qualquer projeto. Para avaliá-la, consideram-se atributos como manutenibilidade, confiabilidade e custo-benefício ao longo do tempo (MORAES; JUNIOR, 2017). O padrão ISO/IEC 25010 define atributos essenciais, como usabilidade, confiabilidade, eficiência, manutenibilidade e portabilidade, que devem ser atendidos para garantir a qualidade do sistema (MORAES; JUNIOR, 2017).

Esses atributos são quantificados por meio de métricas, que transformam características do software em números, possibilitando uma avaliação objetiva (ABID et al., 2020). Exemplos comuns incluem métricas de tamanho, como Linhas de Código (LOC) e Pontos de Função (FP), e métricas de complexidade, como a Complexidade Ciclomática. A escolha das métricas depende dos objetivos da análise e

do tipo de software em questão, ajudando a identificar riscos e áreas de melhoria (ABID et al., 2020).

Diversos fatores impactam a qualidade do software, como os comentários no código, que, se bem elaborados, auxiliam na compreensão e manutenção, enquanto comentários inadequados podem aumentar custos futuros, ocasionando dividas técnicas (IBRAHIM et al., 2012). Essas dívidas técnicas, resultante de soluções rápidas que geram problemas a longo prazo, também compromete a manutenção do software (ABID et al., 2020). No entanto, ferramentas como *SonarQube* e *PMD* ajudam a identificar essas questões (GUO et al., 2014). Além disso, práticas de desenvolvimento, como revisões de código, refatorações e remoções de *bad smells* são cruciais para garantir a qualidade (ALOMAR et al., 2021).

A gestão da qualidade de software envolve o planejamento, garantia e controle da qualidade ao longo do ciclo de vida do software (PERERA et al., 2023). O planejamento da qualidade define metas, métricas e critérios de aceitação, enquanto a garantia de qualidade assegura que os testes e revisões sejam realizados em todas as etapas (IBRAHIM et al., 2020). O controle de qualidade monitora o desenvolvimento para garantir que os padrões sejam seguidos (IBRAHIM et al., 2020). A busca pela qualidade é contínua e iterativa, e o investimento em qualidade nas fases iniciais resulta em sistemas mais robustos, manuteníveis e com menor custo operacional a longo prazo (IAMMARINO et al., 2021).

2.4 Refatoração

Refatoração é a prática de modificar a estrutura interna de um código sem alterar seu comportamento externo, visando melhorar atributos como manutenibilidade, reusabilidade e extensibilidade (MENS; TOURWÉ, 2004). Essa prática envolve transformações controladas que otimizam a estrutura e a legibilidade do código, sendo essencial para a evolução contínua do software, permitindo adaptação a novas demandas sem comprometer sua integridade (MENS; TOURWÉ, 2004). Estudos indicam que a refatoração, quando bem executada, reduz *bad smells* e melhora a manutenção, como discutido por (HALEPMOLLASI; TOSUN, 2024).

A refatoração de renomeação de método, por exemplo, visa tornar os nomes de métodos mais descritivos, melhorando a legibilidade e reduz a necessidade de comentários adicionais. Essa técnica facilita a compreensão do código e diminui o risco de *bugs* associados a comentários desatualizados (IBRAHIM et al., 2012). A seguir, exemplifica-se a refatoração de um método cujo nome foi alterado de "*circum*" Figura 2.5 para "*circumference*" Figura 2.6, tornando-o mais alinhado com sua funcionalidade.

```
1 public static double circum(radius) {
2     return 2 * Math.PI * radius;
3 }
```

Figura 2.5: Código antes da refatoração: Nome de método pouco descritivo.

```
1 public static double circunference(radius) {
2     return 2 * Math.PI * radius;
3 }
```

Figura 2.6: Código após a refatoração: Nome de método mais descritivo.

Conforme destacado por estudos recentes, como o de Perera et al. (2023), a me-

lhoria na clareza do código, incluindo a prática de refatoração, pode ter impactos diretos na qualidade do software, facilitando a manutenção e reduzindo o tempo de resolução de problemas (SONG et al., 2020). Assim, a refatoração deve ser vista como uma prática contínua e estratégica, essencial para a evolução de sistemas de software robustos e sustentáveis (IBRAHIM et al., 2020).

2.5 *Bad Smells*

O termo *bad smells* foi introduzido por *Kent Beck* e popularizado por Martin Fowler em seus livros e pesquisas (MARTIN, 2019). Esses *bad smells* não são erros que impedem a compilação ou execução de um programa, mas sim indicadores de possíveis problemas estruturais no código, como design inadequado, complexidade excessiva ou baixa manutenibilidade (HALEPMOLLASI; TOSUN, 2024). A presença de *bad smells* pode impactar negativamente a qualidade do software a longo prazo, dificultando sua evolução e manutenção. Entre os tipos mais comuns de *bad smells* que afetam a qualidade do código, destacam-se:

1. **Complexidade Ciclomática**: Refere-se ao número de caminhos independentes em um método. Valores elevados indicam maior complexidade, dificultando a compreensão e os testes (MARTIN, 2019). A fragmentação de métodos complexos em componentes menores e mais específicos pode reduzir essa métrica e simplificar o código.

2. **God Class**: Classes que acumulam muitas responsabilidades e concentram uma quantidade desproporcional de funcionalidades. Essas classes tendem a ser extensas e difíceis de compreender (MARTIN, 2019). Dividir essas clas-

ses em componentes menores, cada um com responsabilidades específicas, é uma abordagem recomendada para melhorar a modularidade e a manutenibilidade.

3. **Métodos Longos**: Métodos extensos violam frequentemente o princípio da responsabilidade única, tornando o código confuso e difícil de manter (MARTIN, 2019). A refatoração desses métodos em funções menores e mais coesas promove maior clareza e facilita a manutenção.

A identificação e a correção de *bad smells* são essenciais para assegurar a qualidade do software (SANTANA; CRUZ; FIGUEIREDO, 2021). Ferramentas como *SonarQube*[4], *PMD*[5] e *JDeodorant*[6] desempenham um papel importante ao automatizar a detecção de *bad smells* e sugerir melhorias estruturais, contribuindo para a evolução contínua do código e a mitigação de problemas potenciais.

[4]Disponível em: ⟨https://www.sonarsource.com/products/sonarqube/⟩
[5]Disponível em: ⟨https://pmd.github.io⟩
[6]Disponível em: ⟨https://github.com/tsantalis/JDeodorant⟩

3 Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados a esta pesquisa, fornecendo um panorama das investigações já realizadas sobre o tema. São descritos aspectos como a natureza dos estudos analisados, suas metodologias, principais resultados e lacunas identificadas, que justificam a necessidade deste estudo.

Na Seção 3.1, detalha-se o processo de seleção dos trabalhos, incluindo os critérios adotados e a busca sistemática realizada. Já na Seção 3.2, são discutidos os artigos selecionados, destacando suas contribuições e sua relevância para a pesquisa.

3.1 Seleção dos Trabalhos

A seleção dos trabalhos para esta pesquisa foi conduzida por meio de uma busca sistemática na base de dados *Scopus*, seguindo uma abordagem baseada no método PICO. O objetivo dessa busca foi identificar estudos relevantes que analisam a relação entre refatoração de código, documentação e métricas de código, permitindo um levantamento atualizado sobre o tema. A *query* utilizada na busca é apresentada a seguir:

```
TITLE-ABS-KEY (
( "code comment*" OR "code document*" OR "line comment*"
OR "block comment*")
AND
( "attribute*" OR "development*" OR "lines of code"
OR "code block*" OR "metric*" OR "complexit*" )
AND
( "explorator*" OR "analyz*" OR "compar*" OR "check*"
OR "mining" OR "evaluation" OR "assessment" OR "inspection" ))
AND
PUBYEAR > 2015 AND PUBYEAR < 2026
AND
( LIMIT-TO ( DOCTYPE , "cp" ) OR LIMIT-TO ( DOCTYPE , "ar" ) )
AND
( LIMIT-TO ( SUBJAREA , "COMP" ) )
AND
( LIMIT-TO ( LANGUAGE ,"Portuguese" )
OR LIMIT-TO ( LANGUAGE , "English" ) )
```

A busca retornou um total de 164 trabalhos que atendiam aos critérios da *query*. Para refinar a seleção, foi realizada uma triagem inicial baseada na leitura dos títulos e resumos, resultando na exclusão de 136 trabalhos por não estarem alinhados com os objetivos desta pesquisa. Desses 136 trabalhos excluídos, oito não estavam disponíveis para download, impossibilitando uma análise mais aprofundada.

Os 28 trabalhos restantes foram lidos integralmente e avaliados com base na metodologia empregada, relevância dos resultados e aderência ao escopo desta pesquisa. Como resultado, 5 trabalhos foram selecionados para compor a análise detalhada desta inves-

tigação. As planilhas contendo as análises detalhadas desses estudos estão disponíveis no repositório da pesquisa[7].

3.2 Análise dos Trabalhos Relacionado

3.2.1 A First Look at Duplicate and Near-duplicate Self-admitted Technical Debt Comments

O artigo de Yasmin, Sheikhaei e Tian (2022) apresenta o conceito de comentários duplicados de dívida técnica autoadmitida (SATD) e realiza um estudo preliminar sobre sua existência e características em projetos de software de código aberto. O objetivo principal é abordar a falta de pesquisa sobre a presença e o impacto de comentários SATD duplicados, considerando que sua identificação e gestão são cruciais para a manutenção e evolução do software. O estudo, de caráter aplicado e experimental, propõe um método automatizado para identificar grupos de comentários SATD duplicados e realiza uma análise manual de amostras para determinar suas características e implicações.

A metodologia envolve a mineração do histórico de *commits* em cinco projetos Apache de código aberto, extraindo comentários SATD e utilizando modelos de Processamento de Linguagem Natural (PNL) para calcular similaridade semântica entre os comentários. O algoritmo DBSCAN é aplicado para agrupar comentários duplicados com base em suas pontuações de similaridade, e uma análise manual verifica se os comentários duplicados se referem à mesma causa raiz ou estão associados a clones de código. As principais ferramentas empregadas incluem o SmartSHARK para extração de dados, um modelo BERT pré-treinado para geração de embeddings, a biblioteca SentenceTransformer para cálculo de similaridade e o NiCad para detecção de clones de código. As principais contri-

[7]Disponível em: ⟨https://github.com/KleitonEwerton/project-analyzer-with-parsers/tree/main/trabalhosrelacionados⟩

buições do trabalho incluem a introdução do conceito de comentários SATD duplicados, o desenvolvimento de um método para identificação automatizada desses comentários e uma análise empírica de sua prevalência e características em projetos de código aberto.

Entre as descobertas, destaca-se que a maioria dos comentários SATD duplicados se relaciona à mesma causa raiz, mas nem sempre está presente em clones de código ou é introduzida e removida simultaneamente. O estudo, entretanto, apresenta limitações, como o foco em um número restrito de projetos e a dependência de tags predefinidas para identificar comentários SATD. O artigo conclui que comentários SATD duplicados são prevalentes e representam desafios para a gestão de dívida técnica, destacando a necessidade de pesquisas futuras para compreender melhor as práticas de desenvolvimento relacionadas a esses comentários e seus impactos na qualidade do software.

3.2.2 An empirical study on the co-occurrence between refactoring actions and Self-Admitted Technical Debt removal

O estudo de Iammarino et al. (2021) investiga a relação entre ações de refatoração e a remoção de Dívida Técnica Autoadmitida (SATD), visando compreender se a refatoração contribui diretamente para a remoção de SATD ou se ambas as atividades ocorrem juntas como parte de uma iniciativa geral de melhoria da qualidade do código. A abordagem metodológica combina análises quantitativas e qualitativas, utilizando dados de quatro projetos open-source Java, ferramentas como RMiner para detecção de refatorações, teste *Mann-Whitney U*, análise manual de coocorrências, e modelos de regressão logística.

Os resultados indicam que ações de refatoração são significativamente mais frequentes em *commits* que incluem remoções de SATD, embora somente uma pequena proporção dessas ações esteja explicitamente documentada nas mensagens de *commit*. Cerca de

42% das remoções apresentaram sobreposição entre o código refatorado e o SATD, mas apenas 8% estavam diretamente ligadas à melhoria da manutenibilidade. Apesar da contribuição da refatoração, o estudo conclui que a remoção de SATD é frequentemente parte de um esforço mais amplo de melhoria da qualidade, sendo influenciada também por métricas de qualidade do código, como CBO e LOC.

3.2.3 Analysis of LOC attributes using code analyzer and Correlation methods

O artigo de Srilatha et al. (2017) analisa atributos de linhas de código (LOC) utilizando uma ferramenta de análise de código e métodos de correlação. Embora não explicite um problema específico, o trabalho foca na análise de atributos de LOC para identificar *outliers* e correlações entre métricas de código. Trata-se de um estudo de caráter aplicado que emprega abordagem quantitativa, utilizando dados numéricos extraídos de código-fonte e ferramentas como o Code Analyzer para métricas como LOC, linhas de comentário e linhas em branco. As técnicas utilizadas incluem o método de Pearson para calcular correlações, análise de box plot para identificar *outliers* e gráficos de controle para delimitar valores fora do padrão. Os principais resultados destacam a identificação de *outliers* em atributos como linhas de comentário e uma baixa correlação entre o número de linhas de comentário e outros atributos de software. Embora o artigo não apresente inovações significativas, demonstra a aplicação de técnicas estatísticas conhecidas na análise de código. Entre as limitações, estão a ausência de um problema definido, a falta de generalização dos resultados para outros projetos de software e o foco restrito a um conjunto limitado de atributos de código. Apesar disso, a metodologia e as técnicas estatísticas empregadas são relevantes para projetos que buscam compreender a relação entre atributos de código e o volume de comentários. O artigo conclui que a complexidade do software aumenta com o número de funcionalidades, independentemente do

LOC, e que os métodos estatísticos empregados apresentaram resultados consistentes na identificação de *outliers*.

3.2.4 Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems

O artigo de Heričko e Šumak (2023) analisa variantes do Índice de Manutenibilidade (MI) para avaliar a consistência entre elas na medição da manutenibilidade de sistemas de software orientados a objetos. O problema observado está na falta de clareza sobre a consistência entre variantes do MI e como a escolha da variante pode influenciar a percepção da manutenibilidade. O estudo, de abordagem experimental e aplicada, utilizou 45 sistemas de software de código aberto em Java, com dados extraídos por meio da ferramenta JHawk. Foram empregadas análises descritivas, teste *Mann-Whitney U* (t-Test pareado e Wilcoxon Signed-Rank Test), correlação (coeficiente de Kendall t-b), escalonamento multidimensional e análise de tendência (Mann-Kendall). Entre as principais contribuições, destaca-se a comparação detalhada de sete variantes do MI, que revelou diferenças nos valores gerados, mas alta correlação entre elas, indicando tendências semelhantes de manutenibilidade ao longo do tempo. As variantes foram agrupadas em dois grupos: aquelas que consideram comentários de código e aquelas que não consideram. O estudo limita-se a sistemas Java de código aberto e a sete variantes do MI, sugerindo a inclusão de outras linguagens e variantes em pesquisas futuras. As conclusões enfatizam que a escolha da variante do MI impacta a percepção da manutenibilidade, mas tendências gerais permanecem consistentes. A relevância do estudo para projetos de pesquisa está na metodologia adotada e na análise da relação entre atributos de desenvolvimento, como comentários, e a manutenibilidade do código.

3.2.5 Keyword-labeled self-admitted technical debt and static code analysis have significant relationship but limited overlap

O artigo de Rantala, Mäntylä e Lenarduzzi (2024) investigou a relação entre a dívida técnica autodeclarada com rótulos de palavra-chave (KL-SATD) em comentários de código-fonte e os relatórios gerados pela ferramenta de análise estática SonarQube. O estudo abordou a questão de como essas abordagens se relacionam na identificação de problemas de manutenibilidade, utilizando uma metodologia quantitativa aplicada a 33 repositórios Java do Apache Software Foundation. Foram analisados comentários com as palavras-chave *TODO* e *FIXME*, além de métricas do SonarQube, como índice sqale, esforços de remediação e violações de regras. Os resultados indicaram que a KL-SATD está associada à redução da manutenibilidade do código e que sua remoção melhora a confiabilidade e a manutenibilidade, sendo predominantemente relacionada a *code smells*. Apesar da relação estatística identificada, houve sobreposição limitada entre os comentários KL-SATD e os problemas detectados pelo SonarQube, indicando que ambas as abordagens são complementares. As limitações incluem a generalização para outros tipos de projetos, o uso de configurações padrão do SonarQube e a presença de dados faltantes. O estudo conclui que a KL-SATD e as métricas do SonarQube oferecem perspectivas complementares sobre a qualidade do código, sugerindo a necessidade de investigações futuras para aprimorar a detecção e gestão da dívida técnica.

4 Abordagem

Neste capítulo é descrita a abordagem para investigar a influência entre as refatorações e os *bad smells* na variação do volume de comentários no código-fonte. Ela foi estruturada para automatizar o processo de coleta de dados de repositórios, possibilitando a reprodutibilidade dos resultados e a padronização na coleta dos dados. Para isso, foram adotadas diretrizes que incluem a padronização da coleta de dados e a utilização de ferramentas bem estabelecidas na comunidade acadêmica. Este capítulo está organizado em quatro seções. Na Seção 4.1 apresenta um panorama da pesquisa, destacando os objetivos gerais da abordagem utilizada. A Seção 4.2 discute os procedimentos adotados para garantir a consistência e uniformidade dos dados coletados. A Seção 4.3 descreve a implementação da abordagem, incluindo as ferramentas utilizadas. Por fim, a Seção 4.4 apresenta as considerações finais sobre os desafios e benefícios da abordagem adotada.

4.1 Visão Geral

A pesquisa visa analisar como as refatorações e *bad smells*, fatores que influenciam a qualidade de software, estão associados ao volume e aos tipos de comentários presentes no código-fonte. Para isso, adotou-se uma abordagem sistemática baseada em duas etapas principais:

1. **Identificação de *commits* relevantes**: São selecionados *commits* que contêm arquivos escritos em Java e que tenham sido modificados ao longo da evolução do projeto. Essa filtragem permite focar na análise de mudanças diretamente relacionadas à manutenção do código-fonte.

2. **Extração e categorização de informações**: Para cada *commit* identificado, extraímos automaticamente dados sobre: **Refatorações**: Identificação de transformações estruturais aplicadas no código. ***Bad smells***: Detecção de problemas recorrentes de *design* e código. **Comentários**: Extração e categorização dos comentários presentes no código com base na análise sintática.

3. **Armazenamento e organização dos dados**: As informações extraídas são estruturadas em uma base de dados, permitindo análises quantitativas e a aplicação de técnicas estatísticas e de aprendizado de máquina para identificar padrões entre refatorações, *bad smells* e comentários.

4.2 Especificação e Uniformização da Implementação

Esta seção apresenta a metodologia adotada para garantir a replicabilidade e a consistência dos procedimentos utilizados na extração e análise dos dados. A abordagem foi estruturada para padronizar a coleta de informações, minimizando vieses e garantindo que os resultados possam ser reproduzidos em estudos futuros. Para isso, seguimos uma sequência de passos que envolvem desde a extração do histórico de versões até a armazenagem dos dados. O fluxo metodológico adotado está ilustrado na Figura 4.1.

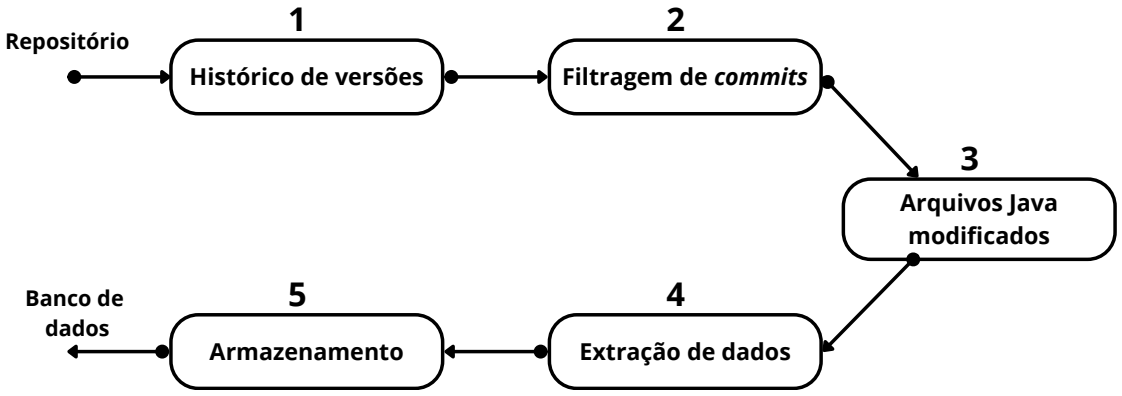

Figura 4.1: Passos da abordagem metodológica.

O processo de extração dos dados seguiu os seguintes passos:

1. **Acesso ao histórico de versões**: Recuperação dos *commits* do histórico do repositório Git.
2. **Filtragem de *commits* relevantes**: Exclusão de *commits* que não envolvem arquivos Java.
3. **Identificação de arquivos Java modificados**: Mapeamento das alterações realizadas em cada *commit*.
4. **Extração de dados**: Coleta de comentários (linha, bloco, documentação), detecção de *bad smells* e identificação das refatorações.
5. **Armazenamento padronizado**: Organização estruturada dos dados extraídos para análise posterior.

Cada um desses passos desempenha um papel fundamental na abordagem adotada. O acesso ao histórico de versões via Git permite recuperar todas as alterações registradas, possibilitando um panorama detalhado da evolução do projeto. Para garantir que

a análise se concentre em mudanças que efetivamente afetam a estrutura e o comportamento do código, aplicamos critérios específicos para a seleção dos *commits*. São consideradas modificações relevantes aquelas em que: **i)** ao menos um arquivo escrito em Java foi modificado no *commit*. **ii)**. o *commit* não contém exclusivamente deleções de arquivos, pois estas não contribuem para a análise da evolução do código e dos comentários. **iii)** as alterações incluem mudanças estruturais no código, como inserções e/ou remoções de trechos de código ou de trechos de comentários.

A identificação dos arquivos Java modificados possibilita mapear como o código evolui ao longo do tempo. Durante a extração de dados, utiliza-se a ferramenta *JavaParser* para capturar os comentários presentes no código e diferenciá-los por tipo. Paralelamente, a ferramenta *PMD* também foi utilizado, permitindo detectar os sete tipos de *bad smells*: *Data Class*, *Long Parameter List*, *Cyclomatic Complexity*, *Many Methods*, *God Class*, *Long Methods* e *Many Fields*. Além disso, a ferramenta *RefactoringMiner* foi adotada para identificar refatorações, extraindo informações sobre suas ocorrências.

4.3 Implementação da Abordagem

A abordagem foi implementada por meio de uma ferramenta desenvolvida na linguagem Java, estruturada utilizando componentes para integrar bibliotecas e ferramentas especializadas. O código-fonte e os dados obtidos estão disponíveis no repositório do Github *GitHub*[8]. A estrutura básica de componentes do sistema é apresentada na Figura 4.2, que ilustra os principais componentes e suas interações.

[8]Disponível em: ⟨https://github.com/KleitonEwerton/project-analyzer-with-parsers⟩

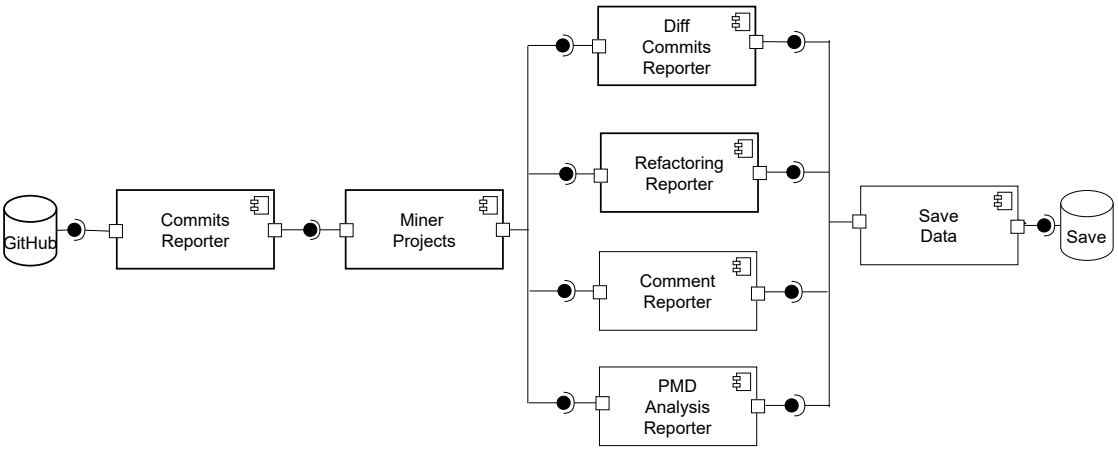

Figura 4.2: Diagrama simplificado de componentes da ferramenta para análise e extração de dados.

Os principais módulos da ferramenta são descritos a seguir:

- **CommitReporter**: Responsável por recuperar os *commits* do repositório utilizando o *Git*. Esse módulo coleta informações essenciais, como identificador do *commit*, arquivos modificados e arquivos apagados. Esses dados permitem rastrear quais arquivos sofreram alterações e servirão de base para a extração de refatorações, *bad smells* e comentários.

- **MinerProjects**: Coordena o fluxo de extração dos dados em cada projeto analisado, chamando os demais módulos conforme necessário.

- **DiffCommitsReporter**: Analisa as diferenças entre versões do código-fonte para cada *commit*, identificando as alterações feitas nos arquivos. As informações coletadas são armazenadas na memória e utilizadas para direcionar a análise dos módulos *RefactoringReporter*, *CommentReporter* e *PMDAnalyzerReporter*.

- **CommentReporter**: Utiliza o *JavaParser* para extrair e classificar os comentários do código. Ele identifica comentários de linha, bloco e de documentação, possibi-

litando sua análise detalhada.

- **RefactoringReporter**: Emprega o *RefactoringMiner* para detectar refatorações no código, categorizando-as de acordo com um dos 95 tipos identificáveis na versão 2.4 da ferramenta.
- **PMDAnalyzerReporter**: Utiliza a ferramenta PMD para identificar a presença dos sete tipos de *bad smells* utilizados na pesquisa, sendo eles: *Data Class*, *Long Parameter List*, *Cyclomatic Complexity*, *Many Methods*, *God Class*, *Long Methods* e *Many Fields*.
- **SaveData**: Estrutura e armazena os dados extraídos em um banco de dados relacional, garantindo organização e acessibilidade para análises posteriores.

O processamento se inicia com a obtenção dos *commits* e a análise dos arquivos modificados. Cada *commit* é examinado para identificar mudanças no código-fonte, refatorações realizadas e a presença de *bad smells*. A seguir, é apresentada na Tabela 4.1 a representação básica de como foi armazenado os dados pela ferramenta.

Tabela 4.1: Representação básica da estrutura dos dados coletados.

Coluna	**Descrição**
`hash do commit`	Identificador único do commit
`Refatoração`	Tipo de refatoração aplicada
`Tipo de comentário`	Classificação do comentário (linha, bloco, documentação)
`Tipo de bad smell`	Tipo de *bad smell* identificado

A estruturação dos dados coletados permite uma análise posterior da relação entre refatorações, *bad smells* e volume de comentários no código-fonte. Com essa abordagem, possibilita-se identificar padrões que podem auxiliar na compreensão do impacto dessas práticas na documentação do código.

4.4 Considerações finais

A abordagem apresentada neste capítulo tem o objetivo de investigar a relação entre refatorações, *bad smells* e a prática de comentar o código-fonte. A metodologia adotada compreende desde a coleta de dados por meio da mineração de repositórios até o salvamento dos dados.

Um dos diferenciais da abordagem é a integração de ferramentas especializadas, como o *JavaParser*, o *RefactoringMiner* e o *PMD*, garantindo a extração de informações relacionadas às mudanças no código, refatorações e identificação de *bad smells*. Apesar das vantagens, alguns desafios foram enfrentados, como a necessidade de filtrar *commits* irrelevantes e lidar com a complexidade computacional do processamento de grandes volumes de dados. Além disso, foi essencial garantir a execução da extração das informações assegurando que as métricas obtidas fossem consistentes e comparáveis entre si. O alinhamento entre os critérios de análise dos comentários, refatorações e *bad smells* foi uma preocupação constante para evitar distorções nos dados e fornecer um panorama fiel da evolução do código-fonte.

A abordagem desenvolvida tem o potencial de contribuir para a pesquisa acadêmica na área de qualidade de software, fornecendo um modelo estruturado de extração e análise de informações. Esse modelo pode ser expandido para diferentes contextos, permitindo a adaptação da metodologia a novas investigações sobre a influência de fatores internos do código, na prática de documentação. Além disso, a organização do software em componentes facilita a incorporação de novas técnicas de análise, permitindo ampliar o escopo de estudos na área.

5 Experimento

Neste capítulo são descritas as etapas do estudo experimental conduzidos para investigar a relação de refatorações e *bad smells* com o volume de comentários no código-fonte. O experimento foi estruturado conforme as diretrizes da Engenharia de Software Experimental, que inclui as fases de definição, planejamento, operação (execução), análise e interpretação de dados (WOHLIN et al., 2012). Na Seção 5.1, é apresentada a definição do estudo. Na Seção 5.2 é descrito o planejamento e o design do experimento.Na Seção 5.3 são apresentados os procedimentos adotados para a coleta dos dados. A análise dos dados obtidos é discutida na Seção 5.4. Na Seção 5.5 é apresentada a interpretação dos resultados e suas implicações. As ameaças à validade do estudo são descritas na Seção 5.6. Por fim, a Seção 5.7 apresenta as considerações finais deste capítulo.

5.1 Definição do Estudo

A qualidade do software está intrinsecamente relacionada à sua manutenibilidade, facilidade de compreensão e sustentabilidade ao longo do tempo. Dentre os fatores que impactam diretamente nesses aspectos, destacam-se as refatorações e a presença de *bad smells*, que podem exigir ou desencorajar a adição de comentários no código-fonte. No entanto, a relação entre esses fatores e o volume de comentários ainda não é amplamente compreendida. Neste contexto, o objetivo deste estudo é: analisar as refatorações e os *bad smells*, com o propósito de caracterizar, com respeito a sua efetividade para reduzir a quantidade e o tamanhos dos comentários e aumentar a qualidade do código-fonte, da perspectiva dos desenvolvedores de software no contexto de manutenção de software.

Com respeito ao escopo, este é um estudo de variação de múltiplo projetos que examina os tipos de refatorações e *bad smells* (objetos de estudo) em um conjunto de repositórios (sujeitos). O objetivo do estudo definido com base no template: motivação, objetos de estudo, propósito, enfoque de qualidade, perspectiva e contexto, especificados no framework da Engenharia de Software Experimental (WOHLIN et al., 2012). O modelo *Goals-Questions-Metrics* (GQM) (OLSSON; RUNESON, 2001) foi utilizado para derivar as questões de pesquisa. A partir do objetivo de investigar como características do código-fonte influenciam a prática de comentários, foram definidas as questões de pesquisa apresentadas na Tabela 5.1.

5.2 Planejamento do Estudo

Nesta fase, foram definidos o contexto no qual o experimento foi executado, as hipóteses, as variáveis dependentes e independentes, os sujeitos do estudo, objetos do estudo, o projeto e a instrumentação do experimento.

Este é um estudo *offline* (não é em ambiente industrial) no qual um desenvolvedor de software analisou diferentes repositórios para identificar relações entre refatorações, *bad smells* e quantidade de comentários no código-fonte. Foram formuladas uma hipótese nula e hipóteses alternativas com respeito a influência de refatorações e *bad smells* sob comentários de código-fonte. H0r: Refatorações não impactaram no tamanho dos comentários no código-fonte. H1r: Refatorações impactaram em redução da quantidade de comentários no código-fonte. H2r: Refatorações aumentaram a quantidade de comentários no código-fonte. H0b: *Bad smells* não impactaram no tamanho dos comentários no código-fonte. H1b: *Bad smells* impactaram em redução da quantidade de comentários no código-fonte. H2b: *Bad smells* aumentaram a quantidade de comentários no código-fonte.

Tabela 5.1: Definição do estudo utilizando o método Goals, Questions and Metrics.

G1: Analisar quais fatores que alteram a qualidade do software, introduzidos durante o desenvolvimento de um software, influenciam no volume de comentários de código.

Q1: Quais fatores apresentam diferenças estatisticamente significativas no volume de comentários de código?

Rationale: Esta questão tem objetivo de avaliar se a ocorrência de *bad smells* e a realização de refatorações impactam a quantidade de comentários adicionados ou removidos ao longo da evolução do código. Com isso, pretende-se identificar se há uma relação estatisticamente significativa entre esses fatores e a documentação do código-fonte.

M1: Quantidade de ocorrências de refatorações e *bad smells* de desenvolvimento por *commit*. Engloba refatorações e a presença de *bad smells*, incluindo: *Data Class*, *Long Parameter List*, *Cyclomatic Complexity*, *Many Methods*, *God Class*, *Long Methods* e *Many Fields*

M2: Quantidade de comentários por *commit* classificados por tipos específicos. Avalia a variação no número de comentários inseridos ou removidos em *commits* que contêm refatorações ou *bad smells*.

Q2: Quais regras de associação identificam que fatores implicam na alteração do volume de comentários?

Rationale: Esta questão tem o propósito de identificar padrões e dependências entre refatorações, *bad smells* e mudanças no volume de comentários. O objetivo é descobrir se determinadas refatorações ou más práticas estruturais frequentemente precedem, ou acompanham modificações na documentação do código.

M1: Quantidade de ocorrências de refatorações e *bad smells* de desenvolvimento por *commit*. Engloba refatorações e a presença de *bad smells*, incluindo: *Data Class*, *Long Parameter List*, *Cyclomatic Complexity*, *Many Methods*, *God Class*, *Long Methods* e *Many Fields*.

M2: Quantidade de comentários por *commit* classificados por tipos específicos. Avalia a variação no número de comentários inseridos ou removidos em *commits* que contêm refatorações ou *bad smells*.

5.2.1 Contexto da Pesquisa

A pesquisa foi conduzida com base na análise de projetos *open-source* hospedados no GitHub. Os critérios de seleção incluíram: *i)* Popularidade e relevância da base de código na comunidade de desenvolvimento. *ii)* Diversidade de aplicações para garantir uma análise mais abrangente. *iii)* Quantidade significativa de *commits* para viabilizar uma análise estatística robusta.

Na Tabela 5.2 são apresentados os projetos analisados, incluindo a quantidade de *commits* disponíveis e os *commits* efetivamente utilizados na análise.

Tabela 5.2: Projetos analisados na pesquisa.

Nome do Projeto	Commits Disponíveis	Commits Analisados	Link
controlsfx	3557	1382	⟨https://github.com/controlsfx/controlsfx⟩
httpcomponents-client	3740	2075	⟨https://github.com/apache/httpcomponents-client⟩
mondrian	4466	2719	⟨https://github.com/pentaho/mondrian⟩
morphia	4556	1710	⟨https://github.com/MorphiaOrg/morphia⟩
openpnp	4777	2231	⟨https://github.com/openpnp/openpnp⟩
pgjdbc	3683	1831	⟨https://github.com/pgjdbc/pgjdbc⟩
spring-data-mongodb	3953	2110	⟨https://github.com/spring-projects/spring-data-mongodb⟩

5.2.2 Design do Estudo

O experimento foi projetado com um único fator: o volume de comentários no código-fonte é alterado conforme os tratamentos correspondem às refatorações e à presença de *bad smells*.

Instrumentação: Avaliamos o impacto de cada tratamento no volume e na quantidade dos comentários em projetos Java de código aberto. Os repositórios selecionados são analisados antes e depois das refatorações e *bad smells* para medir as variações nos comentários.

Software: A análise é realizada utilizando ferramentas como o RefactoringMiner para identificar refatorações, PMD para detectar *bad smells* e JavaParser para extrair os

comentários. Essas ferramentas permitem uma avaliação automatizada e estruturada da relação entre refatorações de código, *bad smells* e a evolução dos comentários.

Fase Operacional: O estudo inclui a coleta de dados, definição do conjunto de dados, pré-processamento e análise estatística do impacto das refatorações e *bad smells* na quantidade de comentários. Os achados são avaliados por meio de teste estatístico e regras de associação para identificar padrões significativos.

5.2.3 Variáveis

Para realizar a análise foram definidas variáveis dependentes e independentes. As **variáveis dependentes** são as seguintes: quantidade de comentários em um *commit*, quantidade de refatorações em um *commit*, quantidade de *bad smells* em um *commit*. Por outro lado, as **variáveis independentes** são: repositórios analisados, linguagem de programação utilizada, número total de *commits* no repositório.

A decisão de analisar individualmente os *bad smells* baseia-se no fato de que diferentes tipos impactam a legibilidade e manutenibilidade do código de maneiras distintas. Trabalhos como Martin (2019) destacam que cada *bad smell* tem implicações específicas, tornando necessária sua avaliação separada. Por outro lado, as refatorações foram agrupadas por representarem um conjunto homogêneo de práticas voltadas à melhoria da estrutura interna do código sem alterar seu comportamento externo (FOWLER, 2009). No entanto, os tipos de refatoração foram coletados individualmente, permitindo uma análise mais detalhada em estudos futuros.

5.2.4 Metodologia

A metodologia utilizada para coletar e analisar os dados incluiu:

- **Extração de Dados**: Uso do *RefactoringMiner* para identificar refatorações,

JavaParser para extração e classificação de comentários, e *PMD* para detecção de *bad smells*.

- **Análise Estatística**: Cálculo de correlações e testes de significância para verificar relações entre refatorações, *bad smells* e comentários. Para avaliar e responder às questões da pesquisa, aplicou-se o teste de *Mann-Whitney U* (WOHLIN et al., 2012), selecionado por sua robustez na comparação de amostras independentes sem pressupor a normalidade dos dados. Este teste possibilitou verificar a significância estatística das diferenças no volume de comentários entre grupos de *commits* associados à presença de determinados *bad smells* e refatorações.

- **Regras de Associação**: Identificação de padrões que associem práticas de refatoração à variação no volume de comentários. Adicionalmente, foi realizada uma análise de regras de associação visando identificar padrões na ocorrência simultânea de refatorações, *bad smells* e alterações no volume de comentários.

5.3 Coleta de Dados

A coleta de dados foi realizada a partir da extração de informações de repositórios de código aberto hospedados no GitHub. O objetivo dessa etapa foi reunir informações sobre *commits* que envolvem modificações em arquivos Java, incluindo a incidência de refatorações, a presença de *bad smells* e as variações no volume de comentários.

5.3.1 Fonte e Critérios de Seleção

Foram analisados múltiplos projetos Java selecionados com base nos seguintes critérios:

- Projetos ativos com um histórico de mais de 3.000 *commits*.

- Disponibilidade do código-fonte e histórico de desenvolvimento no GitHub.

- Mais de 10.000 estrelas.

Tabela 5.3: Tabela de métricas por projeto.

Nome do projeto	**Refatorações**	**Bad Smells**	**Comentários de linha**	**Comentários de bloco**	**Comentários de doc**
controlsfx	14.512	7.938	121.234	3.006	90.730
httpcomponents-client	36.539	15.109	107.676	11.719	88.750
mondrian	24.874	50.164	892.906	16.272	293.315
morphia	32.824	15.314	45.914	1.367	84.200
openpnp	20.479	17.262	204.863	3.376	27.013
pgjdbc	13.660	28.822	404.851	61.889	106.297
spring-data-mongodb	21.300	19.492	278.157	54.574	209.420

A análise considerou somente *commits* que modificaram arquivos Java, excluindo aqueles que representam *merges* ou *commits* vazios.

5.3.2 Descrição dos Dados Coletados

Na Tabela 5.4 é apresentado um exemplo simplificado dos dados coletados para o projeto *spring-data-mongodb*, incluindo informações sobre refatorações, variação de comentários, complexidade ciclomática do código e a presença dos *bad smells*: God Class, Too Many Fields e Too Many Methods.

Tabela 5.4: Tabela dos dados coletados do projeto *spring-data-mongodb*.

id	*Hash* do *Commit*	Refatorações	Diferença nos comentarios de bloco	Diferença nos comentarios de documentação	Diferença nos comentarios de linha	Cyclomatic Complexity	Data Class	Excessive Parameter List	God Class	Too Many Fields	Too Many Methods
1	0000a8f	2	0	0	3	0	0	0	0	0	1
2	00034d8	13	0	0	0	0	1	0	0	0	0
3	0020499	1	0	0	0	9	0	0	3	0	4
4	004e8ba	0	0	0	1	2	0	0	1	1	2
...	...	...	...	...	...	...	...	...	...	...	...
2107	ffba352	8	9	45	19	2	0	0	1	0	7
2108	ffceed8	0	0	19	6	2	0	0	0	0	3
2109	ffe37a7	2	0	0	14	8	1	0	2	0	2
2110	fff69b9	1	1	1	2	0	0	0	0	0	1

Além do exemplo apresentado, os dados coletados abrangem um total de **15.901** ***com-***

mits analisados e que contia modificações em arquivos *Java*, eles compuseram um total de **164.188** refatorações e **154.101** *bad smells*.

5.4 Análise dos dados

Nesta seção, são apresentados os principais achados obtidos a partir dos teste *Mann-Whitney U* e das regras de associação, com foco nas diferenças significativas no volume de comentários e nos padrões observados.

5.4.1 Diferenças Estatísticas no Volume de Comentários

Os testes *Mann-Whitney U* realizados evidenciaram diferenças significativas no volume de comentários em diversos projetos. De maneira geral, as refatorações se destacaram como um fator relevante para o aumento da documentação. No projeto Mondrian, por exemplo, a presença de refatorações resultou em um aumento médio de 4,50 comentários de documentação, enquanto nos commits sem refatorações essa média foi de somente 0,63 (p-valor: 3,73E-55). Um padrão similar foi observado no projeto pgjdbc, no qual as refatorações levaram a uma média de 4,29 comentários, comparado a 0,56 na ausência dessas modificações (p-valor: 3,74E-26).

Tabela 5.5: Resultados dos teste *Mann-Whitney U* dos projetos analisados.

Projeto	Fator	Tipo de Comentário	Média (Com Fator)	Média (Sem Fator)	Desvio Padrão (Com Fator)	Desvio Padrão (Sem Fator)	Commits (Com Fator)	Commits (Sem Fator)	p-valor
controlsfx	Refatorações	Documentação	2,4829	4,8254	8,5445	50,3225	642	739	9,63E-09
controlsfx	GodClass	Documentação	5,0747	1,6394	47,3717	6,7095	843	538	0,0046
controlsfx	*Too Many Methods*	Documentação	4,1798	1,1485	40,2906	4,2844	1179	202	0,0049
httpcomponents-client	Refatorações	Documentação	1,8592	1,3034	7,3641	6,1598	1129	946	8,15E-08
httpcomponents-client	Data Class	Documentação	4,2068	1,1245	13,2853	4,6340	324	1751	1,56E-06
httpcomponents-client	GodClass	Documentação	2,0000	1,2104	8,2363	5,0551	1039	1036	0,0337
mondrian	Refatorações	Documentação	4,5036	0,6317	17,4589	2,5078	1261	1458	3,73E-55
mondrian	Refatorações	Linha	11,4734	2,1420	51,0810	15,7733	1261	1458	3,40E-29
mondrian	Data Class	Documentação	8,8798	1,2618	28,2143	4,7782	416	2303	6,46E-25
openpnp	Refatorações	Documentação	1,1333	0,9163	8,7995	22,4314	953	1278	1,22E-16
openpnp	*Too Many Methods*	Documentação	1,5000	0,1391	22,3958	0,8569	1426	805	9,18E-14
openpnp	Refatorações	Linha	9,4932	3,5086	68,4279	26,8918	953	1278	7,38E-11
pgjdbc	Refatorações	Documentação	4,2933	0,5619	22,7990	37,9225	733	1098	3,74E-26
pgjdbc	Refatorações	Linha	7,0218	2,5319	33,2497	29,0780	733	1098	1,30E-12
pgjdbc	*Too Many Fields*	Documentação	1,8336	2,1658	46,4094	23,2122	607	1224	0,0002
spring-data-mongodb	Refatorações	Bloco	1,0952	-1,3008	6,2059	46,1646	1219	891	1,22E-10
spring-data-mongodb	Refatorações	Documentação	6,0025	0,1605	29,6181	60,7908	1219	891	1,52E-09
spring-data-mongodb	*Too Many Methods*	Linha	4,7062	1,1288	44,7787	7,7461	1947	163	1,15E-05
morphia	Refatorações	Documentação	4,6892	0,7767	49,4997	3,9472	1065	645	0,0001
morphia	*Too Many Methods*	Documentação	3,4119	0,1165	40,4056	1,3380	1607	103	0,0002
morphia	Data Class	Linha	0,7289	0,9683	13,5223	10,3915	450	1260	0,0006

A Tabela 5.5 apresenta os resultados detalhados para cada projeto, evidenciando que, além das refatorações, alguns *bad smells* também influenciam a adição de comentários. No projeto controlsfx, a ocorrência de *God Class* esteve associada a um aumento expressivo na documentação (5,07 contra 1,64; p-valor: 0,0046). Da mesma forma, no projeto openpnp, a presença do *Too Many Methods* resultou em uma média de 1,50 comentários, enquanto na ausência desse *bad smell* a média foi de apenas 0,13 (p-valor: 9,18E-14).

Tabela 5.6: Resultados do teste *Mann-Whitney U* com todos os dados dos projetos analisados.

Fator	Tipo de Comentário	Média (Com Fator)	Média (Sem Fator)	Desvio Padrão (Com Fator)	Desvio Padrão (Sem Fator)	*Commits* (Com Fator)	*Commits* (Sem Fator)	p-valor
Refatoração	Documentação	3,7004	1,1555	25,7480	32,4872	7002	7055	4,81E-107
Too Many Methods	Documentação	2,7405	0,7788	31,9783	4,8417	11783	2274	5,51E-38
Refatoração	Linha	5,3990	3,0007	36,8934	35,9819	7002	7055	4,91E-20
Cyclomatic Complexity	Bloco	0,1352	0,1736	16,8545	1,7358	11038	3019	0,1996
Cyclomatic Complexity	Linha	4,8660	1,7433	40,5528	12,9795	11038	3019	4,09E-16
God Class	Linha	5,2416	2,2812	43,2646	18,1870	9089	4968	1,15E-11
Data Class	Bloco	-0,2951	0,2453	27,5430	9,9719	2650	11407	0,0981
Data Class	Documentação	5,4355	1,7233	58,4859	16,2671	2650	11407	1,44E-08
God Class	Bloco	0,1349	0,1590	18,5623	1,6212	9089	4968	0,2347
Too Many Methods	Linha	4,6697	1,7375	39,0895	17,0799	11783	2274	3,41E-07
God Class	Documentação	2,8062	1,7222	35,1271	13,3972	9089	4968	1,24E-06
Cyclomatic Complexity	Documentação	2,6958	1,4263	32,9382	6,5861	11038	3019	1,94E-06
Too Many Fields	Bloco	0,0500	0,1663	33,1080	3,2462	2761	11296	0,5228
Refatoração	Bloco	0,2559	0,0318	5,7875	20,3099	7002	7055	1,09E-05
Data Class	Linha	10,1049	2,8225	69,4653	22,5220	2650	11407	4,04E-05
Too Many Fields	Linha	8,6335	3,1106	59,4619	28,0011	2761	11296	6,24E-05
Too Many Methods	Bloco	0,1562	0,0774	16,3290	1,1241	11783	2274	0,0086
Too Many Fields	Documentação	3,8439	2,0759	45,4003	23,8285	2761	11296	0,0178

A Tabela 5.6 consolida os resultados dos projetos analisados, confirmando que refatorações e certos *bad smells*, como *Too Many Methods* e *God Class*, têm um impacto significativo no volume de comentários, especialmente os de documentação. Em projetos como Mondrian e OpenPnP, a presença de refatorações levou a um aumento expressivo no número médio de comentários, reforçando a hipótese de que modificações estruturais incentivam a adição de explicações no código.

Dentre os *bad smells*, os resultados mostram que *God Class* e *Too Many Methods* estão fortemente associados ao aumento da documentação. No projeto controlsfx, por exemplo, a presença de *God Class* elevou a média de comentários de documentação de 1,64 para

5,07 (p-valor: 0,0046). Da mesma forma, no projeto openpnp, *Too Many Methods* resultou em uma média de 1,50 comentários, contra apenas 0,13 na sua ausência (p-valor: 9,18E-14). Esses achados sugerem que a complexidade introduzida por esses *bad smells* pode levar os desenvolvedores a adicionar mais explicações para mitigar dificuldades de manutenção.

Esse comportamento pode ser explicado pelo fato de que muitas refatorações, como *Extract Method* e *Push Down*, fragmentam trechos de código maiores em unidades menores e mais compreensíveis. Essa subdivisão exige frequentemente a adição de novos comentários para documentar as mudanças, garantindo que a evolução do código seja bem documentada e compreensível para outros desenvolvedores.

Assim, com base nos resultados apresentados na Tabela 5.5, observa-se que as refatorações não somente alteram a estrutura do código, mas também podem resultar em um aumento no volume de documentação. Projetos como Mondrian e OpenPnP ilustram essa tendência, destacando que a documentação pode ser usada como uma estratégia compensatória para explicar mudanças significativas na arquitetura do código. Esse fenômeno alerta para o risco de que a utilização errônea de refatoração, possa levar a um uso excessivo de comentários, o que pode prejudicar a clareza e a manutenibilidade do código (MARTIN, 2019). Isso reforça a necessidade de boas práticas de documentação, para evitar que o código se torne excessivamente dependente de anotações que tentam compensar falhas estruturais ou de clareza.

5.4.2 Regras de Associação

Os resultados das regras de associação reforçaram os achados estatísticos, evidenciando padrões consistentes entre refatorações, *bad smells* e o volume de comentários no código. O padrão mais notável foi a associação entre refatorações e aumento de comentários,

especialmente no projeto *mondrian*, em que essa relação apresentou confiança de 0,5527 e *lift* de 1,4046 (Tabela 5.7). Esse padrão sugere que as refatorações frequentemente geram novos métodos e estruturas que exigem documentação adicional para esclarecimento das mudanças.

No que se refere aos *bad smells*, a presença do *Too Many Methods* no projeto *openpnp* apresentou confiança de 0,2138 (*lift*: 1,3002), indicando que esse *smell* frequentemente leva ao aumento da documentação. Além disso, no *spring-data-mongodb*, a complexidade ciclomática mostrou uma relação significativa com o crescimento da documentação, apresentando confiança de 0,4933.

A Tabela 5.8 consolida esses achados ao unir os dados dos diferentes projetos analisados. Observa-se que a presença de refatorações possui um *lift* elevado, sugerindo um impacto relevante na quantidade de comentários adicionados ao código. Esse achado indica que os desenvolvedores tendem a documentar mais o código após refatorações, seja para registrar as mudanças realizadas ou para tornar a nova estrutura mais compreensível para outros membros da equipe.

Tabela 5.7: Resultados das regras de associação por projetos analisados.

Projeto	**Regra**	**Suporte**	**Confiança**	**Lift**
mondria	Refatorações → Aumento de Comentários de Linha	0,2795	0,6027	1,2519
mondria	Refatorações → Aumento de Comentários de Documentação	0,2563	0,5527	1,4046
spring-data-mongodb	Refatorações → Aumento de Comentários de Documentação	0,3081	0,5332	1,1107
openpnp	Refatorações → Aumento de Comentários de Linha	0,2134	0,4995	1,2868
mondria	*God Class*→ Aumento de Comentários de Linha	0,4027	0,4991	1,0367
mondria	Cyclomatic Complexity → Aumento de Comentários de Linha	0,4494	0,4988	1,0360
spring-data-mongodb	Cyclomatic Complexity → Aumento de Comentários de Documentação	0,3498	0,4933	1,0275
mondria	*Too Many Methods* → Aumento de Comentários de Linha	0,4395	0,4924	1,0227
spring-data-mongodb	*God Class*→ Aumento de Comentários de Documentação	0,2550	0,4904	1,0215
spring-data-mongodb	*Too Many Methods* → Aumento de Comentários de Documentação	0,4488	0,4864	1,0131

Ao analisar os *bad smells*, os dados da Tabela 5.7 mostram que *Too Many Methods* e *God Class* são fatores que frequentemente influenciam o volume de comentários. No caso do projeto *ControlsFX*, a presença de *God Class* foi associada a um aumento expressivo na documentação (confiança de 0,4991), conforme a Tabela 5.8. Isso pode indicar que os desenvolvedores adicionam mais comentários como uma estratégia para compensar a complexidade excessiva dessas classes, tornando o código mais compreensível sem necessariamente resolver o problema estrutural subjacente.

Tabela 5.8: Resultados das regras de associação geradas com todos os dados dos projetos analisados.

Regra	**Suporte**	**Confiança**	**Lift**
Refatoração → Aumento de Comentários de Documento	0,2081	0,4177	1,3306
Refatoração →Aumento de Comentários de Linha	0,2014	0,4043	1,1743
God Class Positiva →Aumento de Comentários de Linha	0,2408	0,3724	1,0817
Complexidade Ciclomática Positiva →Aumento de Comentários de Linha	0,2900	0,3694	1,0727
Muitos Métodos Positivos →Aumento de Comentários de Linha	0,2979	0,3554	1,0323
Muitos Métodos Positivos →Aumento de Comentários de Documento	0,2853	0,3404	1,0843
God Class Positiva →Aumento de Comentários de Documento	0,2161	0,3343	1,0647
Complexidade Ciclomática Positiva →Aumento de Comentários de Documento	0,2562	0,3262	1,0392

A relação entre complexidade ciclomática e volume de comentários, evidenciada na Tabela 5.8, sugere que trechos de código mais complexos geram um aumento nos comentários de linha. Esse comportamento pode ser explicado pela necessidade de explicitar fluxos de execução intricados, nas quais múltiplas condições e laços de repetição podem dificultar a interpretação direta do código.

5.4.3 Q1: Quais fatores apresentam diferenças estatisticamente significativas no volume de comentários de código?

Esta questão buscou avaliar se a ocorrência de *bad smells* e a realização de refatorações impactam no volume de comentários adicionados ou removidos ao longo da evolução do código. Com isso, pretende-se identificar se há uma relação estatisticamente significativa entre esses fatores e a documentação do código-fonte.

Os resultados do teste de *Mann-Whitney U* com a união dos dados de todos os projetos (Tabela 5.5) demonstram que as refatorações têm um impacto estatisticamente significativo no volume de comentários, com p-valores muito baixos (menor que 0,001) em diversos projetos. Os valores médios indicam um aumento considerável no número de comentários em *commits* que contêm refatorações, especialmente comentários de documentação. Assim, os resultados confirmam **H2r** e rejeitam **H0r** e **H1r**, indicando que refatorações tendem a aumentar a quantidade de comentários no código-fonte.

Os testes de *Mann-Whitney U* também indicam uma relação estatisticamente significativa entre a presença de alguns *bad smells* e o aumento dos comentários, como evidenciado pelos casos de *God Class* e *Too Many Methods*. Esses resultados confirmam **H0b** e rejeitam **H1b** e **H2b**, sugerindo que a presença de *bad smells* pode levar os desenvolvedores a adicionar mais comentários para esclarecer trechos de código considerados complexos ou problemáticos.

5.4.4 Q2: Quais regras de associação identificam que fatores implicam na alteração do volume de comentários?

Além de analisar diferenças estatísticas, esta questão busca identificar padrões e dependências entre refatorações, *bad smells* e mudanças no volume de comentários. O objetivo é descobrir se determinadas refatorações ou más práticas estruturais frequentemente precedem ou acompanham modificações na documentação do código.

Os principais achados incluem:

- **Refatorações** estão associadas a um aumento de comentários de documentação, com lift de 1,4046 no projeto *Mondrian* e 1,1107 no *Spring Data MongoDB* (Tabela 5.7).
- **Too Many Methods** impacta significativamente o volume de comentários, com confiança de 0,4924 no *Mondrian* e lift de 1,0843 na análise consolidada (Tabela 5.8).
- **God Class** também está fortemente associada a um aumento no volume de comentários, com confiança de 0,4991 no *ControlsFX* e lift de 1,0647 na análise unificada.

Esses achados indicam que refatorações e *bad smells* influenciam diretamente o comportamento dos desenvolvedores em relação à documentação do código, evidenciando um padrão de aumento de comentários como resposta às mudanças estruturais no software.

5.5 Discussão

Os resultados obtidos neste estudo revelam padrões significativos na relação entre refatorações, *bad smells* e o volume de comentários no código-fonte. Essa análise foi con-

duzida com base em dados extraídos de múltiplos projetos Java, utilizando técnicas estatísticas e regras de associação. A seguir, discutiremos os principais achados, correlacionando-os às figuras e tabelas apresentadas.

As refatorações emergiram como o fator mais fortemente associado ao aumento de comentários no código-fonte, conforme evidenciado nas Tabelas 5.8 e 5.7. Especificamente, a Tabela 5.7 destaca que no projeto Mondrian, as refatorações foram responsáveis por um aumento expressivo nos comentários de documentação (confiança: 0,5527; lift: 1,4046). Esse padrão também foi observado no projeto Spring Data MongoDB (confiança: 0,5332; lift: 1,1107) e no OpenPnP (confiança: 0,4995; lift: 1,2868).

Esse comportamento pode ser explicado pelo fato de que muitas refatorações, como *extract method* e *push down*, tendem a subdividir o código, introduzindo novas estruturas que exigem explicações adicionais. Um exemplo específico é ilustrado na Figura 5.1, que mostra uma refatoração de *extract operation* no *commit* '26d7d974' do projeto Mondrian. Nesse caso, a criação de três novos métodos no arquivo 'RolaAggregationManager.java' resultou em um aumento significativo nos comentários de documentação. Da mesma forma, a Figura 5.2 exemplifica uma refatoração de *push down* no *commit* 'db75e4e8', no qual a redistribuição de responsabilidades entre classes demandou a inserção de novos comentários para esclarecer as mudanças realizadas.

RolapAggregationManager.java

```
228 -   // Go through the compound members/tuples once and separate them
229 -   // into groups.
230 -   List<List<RolapMember>> rolapAggregationList =
231 -   new ArrayList<List<RolapMember>>( );
232 -   for (List<Member> members : aggregationList) {
233 -       // REVIEW: do we need to copy?
234 -       List<RolapMember> rolapMembers = Util.cast(members);
235 -       rolapAggregationList. add(rolapMembers);
205 +   /**
206 +    * Retrieves CompoundPredicateInfo for each tuple in the evaluator's
207 +    * aggregationLists. Uses this to add the predicate (and its string
208 +    * representation) to the cell request.
209 +    * Returns false if any predicate results in an unsatisfiable request.
210 +   */
211 +   private static boolean applyCompoundPredicates(
212 +       RolapEvaluator evaluator,  CellRequest request)
213 +   {
214 +       final Member[] currentMembers = evaluator. getNonAllMembers( );
215 +       final RolapstoredMeasure measure =
216 +           (RolapstoredMeasure)currentMembers[e];
217 +       for (List<List<Member>> aggregationList
218 +           : evaluator. getAggregationLists ( ) )
219 +      {
```

Figura 5.1: Ocorrência de refatoração (*Extract Operration*) no arquivo 'RolaAggregationManager.java' no *commit* '26d7d974' do projeto Modrian aumento de documentação.

JdbcDialectImp.java

```
751 -   public boolean supportsOrderByNullsLast( ) {
752 -       return false;
754 +   protected String generateOrderByNullsLast(
755 +       String expr,
756 +           boolean ascending)
757 +
758 +       // default implementation makes no attempt to force nulls to
759 +       // collate last
760 +       return expr + (ascending ? " ASC" : " DESC" );
761 +    }
```

Figura 5.2: Ocorrência de refatoração (*PUSH DOWN*) no arquivo 'JdbcDialectImp.java' do *commit* 'db75e4e8' do projeto Modrian aumento de documentação.

Entre os *bad smells* analisados, *God Class* e *Too Many Methods* foram os mais frequentemente associados a mudanças no volume de comentários. No projeto Mondrian, a presença de *God Class* foi correlacionada com um aumento de comentários de linha (confiança: 0,4991; lift: 1,0367), conforme mostrado na Tabela 5.7. Essa relação é exemplificada na Figura 5.3, que ilustra o arquivo 'Util.java', uma *God Class* com mais de 1.400 linhas de código utilizada para múltiplos propósitos. A complexidade dessa classe exige explicações adicionais, especialmente durante sua refatoração.

Util.java

```
1391 + /**
1392 +  * Returns whether a collection contains precisely one distinct element.
1393 +  * Returns false if the collection is empty, or if it contains elements
1394 +  * that are not the same as each other.
1395 +  *
1396 +  * @param collection Collection
1397 +  * @return boolean true if all values are same
1398 +  */
1399 + public static <T> boolean areOccurrencesEqual(
1400 +     Collection<T> collection)
1401 + {
1402 +     Iterator<T> it = collection.iterator();
1403 +     if (!it.hasNext()) {
1404 +         // Collection is empty
1405 +         return false;
1406 +     }
1407 + }
```

Figura 5.3: Ocorrência de *God Class* do arquivo 'Util.java' no *commit* '3b50bf13' do projeto Mondrian.

De maneira semelhante, no projeto OpenPnP, a presença de *Too Many Methods* mostrou uma forte relação com o aumento de comentários de documentação (confiança: 0,4864; lift: 1,0131). Isso pode ser atribuído à necessidade de esclarecer a funcionalidade de múltiplos métodos, como destacado na Tabela 5.8. A Figura 5.4 ilustra outro exemplo relevante, mostrando o arquivo ReplicationSlotInfo.java no projeto pgjdbc, uma *Data Class* cuja simplicidade estrutural ainda assim demanda documentação para explicar seu

propósito.

ReplicationSlotInfo.java

```
27 +    public ReplicationSlotInfo(String slotName, ReplicationType replicationType,
28 +        LogSequenceNumber consistentPoint, String snapshotName, String outputPlugin) {
29 +        this.slotName = slotName;
30 +        this.replicationType = replicationType;
31 +        this.consistentPoint = consistentPoint;
32 +        this.snapshotName = snapshotName;
33 +        this.outputPlugin = outputPlugin;
34 +    }
35 +
36 +    /**
37 +     * Replication slot name.
38 +     *
39 +     * @return the slot name
40 +     */
41 +    public String getSlotName() {
42 +        return slotName;
43 +    }
44 +
45 +    /**
46 +     * Replication type of the slot created, might be PHYSICAL or LOGICAL.
47 +     *
48 +     * @return ReplicationType, PHYSICAL or LOGICAL
49 +     */
50 +    public ReplicationType getReplicationType() {
51 +        return replicationType;
52 + }
```

Figura 5.4: Ocorrência de *Data Class* no arquivo ReplicationSlotInfo.java' do *commit* '84e8d90b' do projeto pgjdbc.

A complexidade ciclomática elevada também se mostrou um fator relevante para o aumento de comentários, especialmente no projeto Spring Data MongoDB. A Tabela 5.7 indica que a complexidade ciclomática foi associada ao aumento de comentários de linha (confiança: 0,4933; lift: 1,0275). A Figura 5.5 exemplifica essa relação, mostrando trechos de código altamente complexos no projeto httpcomponents-client que exigem explicações adicionais para melhorar a compreensão.

DigestScheme.java

```
693 + /**
694 +   * Maps a string representation of an algorithm to the corresponding 695 enum constant.
695 +
696 +   * @param algorithm the algorithm name, e.g., "SHA-256" or "SHA-512-256-sess"
697 +   * @return the corresponding {@code DigestAlgorithm} constant
698 +   * @throws UnsupportedDigestAlgorithmException if the algorithm is unsupported
699 + */
700 + private static DigestAlgorithm fromString(final String algorithm) {
701 +    switch (algorithm.toUpperCase(Locale.ROOT)) {
702 +        case "MD5":
703 +            return MD5;
704 +        case "MD5-SESS":
705 +            return MD5_SESS;
706 +        case "SHA-256":
707 +            return SHA_256;
708 +        case "SHA-256-SESS":
709 +            return SHA_256_SESS;
710 +        case "SHA-512/256":
711 +        case "SHA-512-256":
712 +            return SHA_512_256;
713 +        case "SHA-512-256-SESS":
714 +            return SHA_512_256_SESS;
715 +        default:
716 +            throw new UnsupportedDigestAlgorithmException("Unsupported digest algorithm: " + algorithm);
```

Figura 5.5: Ocorrência de *Cyclomatic Complexity* no arquivo 'DigestScheme.java' do *commit* '4b2a365c' do projeto httpcomponents-client.

As regras de associação geradas pela união dos dados analisados (Tabela 5.8) reforçam esses padrões. Por exemplo, a regra "Refatoração → Aumento de Comentários de Documentação"apresenta um lift de 1,3306, indicando que as refatorações têm impacto relevante na quantidade de documentação gerada. Da mesma forma, a presença de *God Class* e *Too Many Methods* está associada ao aumento de comentários de linha e documentação, com lifts de 1,0647 e 1,0843, respectivamente.

Esses achados sugerem que práticas de refatoração planejadas e revisões regulares de *bad smells* podem contribuir significativamente para melhorar a documentação do código, aumentando sua clareza e manutenibilidade a longo prazo.

5.6 Ameaças à Validade

Apesar das contribuições desta pesquisa, algumas limitações devem ser consideradas ao interpretar os resultados. Essas limitações decorrem de decisões metodológicas e de fatores externos que podem ter influenciado os achados. A seguir, discutimos as principais ameaças à validade, categorizadas conforme Wohlin et al. (2012).

i) Validade de Conclusão: A validade de conclusão diz respeito à capacidade de estabelecer corretamente relações entre as refatorações, *bad smells* e o volume de comentários. **Baixo poder estatístico**: A análise foi conduzida em um conjunto específico de repositórios, o que pode limitar a robustez estatística dos testes aplicados. Embora os teste *Mann-Whitney U* tenham sido conduzidos para identificar diferenças significativas, um número maior de projetos poderia fortalecer a confiabilidade dos achados. **Assunções estatísticas**: Algumas análises pressupõem distribuições específicas dos dados, como normalidade e independência das observações. Violações dessas suposições podem comprometer a validade dos resultados estatísticos. Técnicas como testes não paramétricos foram utilizadas para mitigar essa ameaça.

ii) Validade Interna: A validade interna refere-se à existência de fatores que possam ter influenciado os resultados sem o conhecimento do pesquisador. Histórico e maturação: Como os projetos analisados possuem diferentes tempos de desenvolvimento e manutenção, mudanças estruturais nos repositórios ao longo do tempo podem ter influenciado os padrões de comentários independentemente das refatorações e *bad smells*. **Instrumentação**: O uso de ferramentas automáticas para detecção de refatorações, *bad smells* e comentários pode introduzir vieses. Ferramentas como RefactoringMiner e PMD possuem limitações na identificação exata de modificações e podem gerar falsos positivos ou negativos. No entanto, foram escolhidas por sua ampla utilização na literatura e validadas em estudos anteriores. **Viés de seleção**: A escolha dos repositórios

foi baseada em critérios como popularidade e atividade, o que pode ter introduzido viés. Projetos bem mantidos podem apresentar padrões de documentação diferentes daqueles em projetos com menor manutenção, afetando as associações observadas.

iii) Validade de Construção: A validade de construção está relacionada à precisão com que os conceitos teóricos foram operacionalizados na pesquisa. **Definição dos construtos**: A medição do volume de comentários pode não capturar completamente sua qualidade ou utilidade. Além disso, diferentes tipos de refatoração podem ter impactos distintos na documentação, mas foram agrupados para análise estatística. **Viés de mono-operação**: O estudo se baseia em uma única fonte de dados – os repositórios de código-fonte – sem triangulação com outras abordagens, como entrevistas com desenvolvedores. Isso pode limitar a interpretação dos resultados, já que não há informações diretas sobre a intenção dos desenvolvedores ao adicionar ou remover comentários.

iv) Validade Externa: A validade externa trata da possibilidade de generalizar os resultados para outros contextos além dos analisados. **Generalização para outras linguagens**: A análise foi conduzida exclusivamente em código Java, o que pode limitar a aplicabilidade dos resultados a outras linguagens de programação. Paradigmas diferentes podem influenciar como refatorações e *bad smells* impactam a documentação do código. **Interação entre configuração experimental e tratamento**: Os projetos analisados podem não representar práticas comuns em toda a indústria de software. Organizações com diretrizes específicas para documentação e refatoração podem apresentar padrões diferentes dos observados neste estudo. **Interação entre tempo e tratamento**: O estudo analisou projetos em um período específico de tempo, mas padrões de documentação podem evoluir ao longo dos anos. Estudos longitudinais poderiam examinar tendências de longo prazo na relação entre refatoração e comentários.

Essas limitações não invalidam os achados do estudo, mas destacam oportunidades para

pesquisas futuras que ampliem a análise, explorando diferentes contextos, abordagens metodológicas e perspectivas qualitativas.

5.7 Considerações Finais

Os resultados da pesquisa indicam que a presença de refatorações está estatisticamente associada ao aumento do volume de comentários, especialmente os voltados à documentação. Refatorações como *Extract Method* e *Push Down* demonstraram forte tendência de introdução de novos comentários, sugerindo que a reestruturação do código frequentemente demanda explicações adicionais para garantir a compreensibilidade do sistema. Esse padrão foi particularmente evidente nos projetos Mondrian, Spring Data MongoDB e OpenPnP.

Em relação aos *bad smells*, padrões como *God Class* e *Too Many Methods* mostraram uma relação significativa com o acréscimo de comentários. No projeto Mondrian, a presença de *God Classes* resultou em um aumento expressivo de comentários de linha, enquanto no OpenPnP, *Too Many Methods* esteve associado a um crescimento nos comentários de documentação. Esses achados sugerem que desenvolvedores utilizam a documentação como um mecanismo paliativo para lidar com dificuldades de compreensão geradas por estruturas problemáticas no código.

A análise de regras de associação reforçou a influência das refatorações no aumento da documentação, com um *lift* de 1,3306, indicando que mudanças estruturais frequentemente exigem explicações adicionais. Os *bad smells*, embora também associados ao crescimento do volume de comentários, apresentaram um impacto menos pronunciado. Isso sugere que, enquanto refatorações motivam a inserção de novas explicações para dar suporte à compreensão do código alterado, a presença de *bad smells* leva a um aumento de comentários mais como uma necessidade de compensação da complexidade existente.

Por fim, os achados desta pesquisa podem contribuir para o aprimoramento de ferramentas automatizadas de suporte ao desenvolvimento, auxiliando na detecção da necessidade de documentação em processos de refatoração. Além disso, futuras investigações podem expandir essa análise para outras linguagens de programação e incluir abordagens qualitativas, como entrevistas com desenvolvedores, a fim de obter uma compreensão mais aprofundada sobre as motivações e desafios da documentação do código.

6 Conclusão

Este estudo explorou a relação entre refatorações, *bad smells* e a variação do volume de comentários no código-fonte, fornecendo evidências quantitativas e qualitativas sobre como a qualidade estrutural do software influencia a documentação produzida pelos desenvolvedores. Os resultados obtidos reforçam as hipóteses de que tanto refatorações quanto a presença de *bad smells* impactam significativamente o volume de comentários, ainda que de formas distintas.

No que tange à primeira questão de pesquisa (Q1), foi identificado que refatorações como *Extract Method* e *Push Down* são estatisticamente associados ao aumento de comentários de documentação. Esse fenômeno sugere que a reestruturação do código frequentemente demanda uma explicação mais detalhada para contextualizar as mudanças, garantindo a compreensibilidade do sistema. Da mesma forma, a presença de *bad smells*, como *God Class* e *Too Many Methods*, também demonstrou relação significativa com um aumento no volume de comentários, indicando que os desenvolvedores utilizam a documentação como um mecanismo de mitigação para o aumento da complexidade estrutural do código.

Quanto à segunda questão de pesquisa (Q2), a análise baseada em regras de associação revelou que refatoração possuem um impacto mais direto e consistente na documentação do código quando comparados aos *bad smells*. O *lift* calculado para as regras geradas indica que refatoração, em particular, estão frequentemente correlacionados com a introdução de novos comentários explicativos. Por outro lado, embora a presença de *bad smells* também impulsione a inserção de comentários, seu efeito é menos pronunciado e mais dependente do contexto específico do projeto analisado.

Diante desses achados, o estudo contribui para a compreensão da interação entre qualidade do código e práticas de documentação, destacando a importância da integração de processos de análise de código e documentação no desenvolvimento de software. A partir dessas evidências, sugere-se que ferramentas automatizadas de análise de código incorporem métricas que considerem não somente a presença de refatoração e *bad smells*, mas também a evolução dos comentários, permitindo um monitoramento mais preciso da compreensibilidade do software ao longo do tempo.

Como limitação, destaca-se que o estudo se concentrou em projetos de software open-source escritos em Java, o que pode impactar a generalização dos resultados para outros contextos e linguagens de programação. Além disso, a classificação das refatorações e *bad smells* foram realizadas com base em ferramentas que utilizam heurísticas, podendo haver erros e incertezas na coleta.

Para trabalhos futuros, sugere-se a ampliação da pesquisa para diferentes linguagens de programação e ambientes industriais, além da investigação de como diferentes estilos de documentação afetam a compreensão e a manutenção do código. Também seria valioso explorar como a interação entre desenvolvimento colaborativo e refatoramento influencia padrões de comentação, permitindo a formulação de diretrizes ainda mais precisas para a melhoria da documentação de software. Dessa forma, este estudo não apenas amplia o entendimento sobre a relação entre refatoramento,*bad smells* e documentação, mas também abre caminho para novas abordagens que integrem essas dimensões como parte de um ciclo contínuo de melhoria na engenharia de software.

Bibliografia

ABID, C.; KESSENTINI, M.; ALIZADEH, V.; DHAOUADI, M.; KAZMAN, R. How does refactoring impact security when improving quality? a security-aware refactoring approach. *IEEE Transactions on Software Engineering*, IEEE, v. 48, n. 3, p. 864–878, 2020.

ALOMAR, E. A.; ALRUBAYE, H.; MKAOUER, M. W.; OUNI, A.; KESSENTINI, M. Refactoring practices in the context of modern code review: An industrial case study at xerox. In: IEEE. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. [S.l.], 2021. p. 348–357.

CHAROENWET, W.; THONGTANUNAM, P.; PHAM, V.-T.; TREUDE, C. Toward effective secure code reviews: an empirical study of security-related coding weaknesses. *Empirical Software Engineering*, Springer, v. 29, n. 4, p. 88, 2024.

FOWLER, M. *Refatoração: Aperfeiçoamento e Projeto*. [S.l.]: Bookman Editora, 2009.

GUO, J.-J.; HSUEH, N.-L.; LEE, W.-T.; HWANG, S.-C. Improving software maintenance for pattern-based software development: A comment refactoring approach. In: IEEE. *2014 International Conference on Trustworthy Systems and their Applications*. [S.l.], 2014. p. 75–79.

HALEPMOLLASI, R.; TOSUN, A. Exploring the relationship between refactoring and code debt indicators. *Journal of Software: Evolution and Process*, Wiley Online Library, v. 36, n. 1, p. e2447, 2024.

HERIČKO, T.; ŠUMAK, B. Exploring maintainability index variants for software maintainability measurement in object-oriented systems. *Applied Sciences*, MDPI, v. 13, n. 5, p. 2972, 2023.

IAMMARINO, M.; ZAMPETTI, F.; AVERSANO, L.; PENTA, M. D. An empirical study on the co-occurrence between refactoring actions and self-admitted technical debt removal. *Journal of Systems and Software*, Elsevier, v. 178, p. 110976, 2021.

IBRAHIM, R.; AHMED, M.; NAYAK, R.; JAMEL, S. Reducing redundancy of test cases generation using code smell detection and refactoring. *Journal of King Saud University-Computer and Information Sciences*, Elsevier, v. 32, n. 3, p. 367–374, 2020.

IBRAHIM, W. M.; BETTENBURG, N.; ADAMS, B.; HASSAN, A. E. On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, Elsevier, v. 85, n. 10, p. 2293–2304, 2012.

KAUR, A.; KAUR, M. Analysis of code refactoring impact on software quality. In: EDP SCIENCES. *MATEC Web of Conferences*. [S.l.], 2016. v. 57, p. 02012.

MARTIN, R. *Código Limpo: Habilidades Práticas do Agile Software*. Alta Books, 2019. ISBN 9788550811482. Disponível em: ⟨https://books.google.com.br/books?id=GXWkDwAAQBAJ⟩.

MENS, T.; TOURWÉ, T. A survey of software refactoring. *IEEE Transactions on software engineering*, IEEE, v. 30, n. 2, p. 126–139, 2004.

MORAES, M. H. B. M.; JUNIOR, F. R. L. Proposição e aplicação de uma metodologia baseada no ahp e na iso/iec 25000 para apoiar a avaliação da qualidade de softwares de gestão de projetos. *Revista Gestão da Produção Operações e Sistemas*, v. 12, n. 2, p. 239–239, 2017.

OLSSON, T.; RUNESON, P. V-gqm: A feed-back approach to validation of a gqm study. In: IEEE. *Proceedings Seventh International Software Metrics Symposium*. [S.l.], 2001. p. 236–245.

PERERA, D.; PREMATHILAKE, H.; THATHSARANI, K.; NETHMINI, R.; SILVA, D. D.; SAMARASEKARA, H. Analyzing the impact of code commenting on software quality. In: IEEE. *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. [S.l.], 2023. p. 1–6.

RANTALA, L.; MÄNTYLÄ, M.; LENARDUZZI, V. Keyword-labeled self-admitted technical debt and static code analysis have significant relationship but limited overlap. *Software Quality Journal*, Springer, v. 32, n. 2, p. 391–429, 2024.

SANTANA, A.; CRUZ, D.; FIGUEIREDO, E. An exploratory study on the identification and evaluation of bad smell agglomerations. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. [S.l.: s.n.], 2021. p. 1289–1297.

SILVA, S. R. Q. M. d. *Controle de versões-um apoio à edição colaborativa na Web*. Tese (Doutorado) — Universidade de São Paulo, 2005.

SOLINGEN, R. V.; BASILI, V.; CALDIERA, G.; ROMBACH, H. D. Goal question metric (gqm) approach. *Encyclopedia of software engineering*, Wiley Online Library, 2002.

SONG, Q.; KONG, X.; WANG, L.; LI, B. An empirical investigation into the effects of code comments on issue resolution. In: IEEE. *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. [S.l.], 2020. p. 921–930.

SRILATHA, G.; MADHUMATHI, R.; SRESHTA, P.; THIRUMALAI, C. Analysis of loc attributes using code analyzer and correlation methods. In: IEEE. *2017 International Conference on Trends in Electronics and Informatics (ICEI)*. [S.l.], 2017. p. 1147–1150.

TSANTALIS, N.; KETKAR, A.; DIG, D. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, IEEE, v. 48, n. 3, p. 930–950, 2020.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. et al. *Experimentation in software engineering*. [S.l.]: Springer, 2012. v. 236.

YASMIN, J.; SHEIKHAEI, M. S.; TIAN, Y. A first look at duplicate and near-duplicate self-admitted technical debt comments. In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. [S.l.: s.n.], 2022. p. 614–618.