PGCOMP - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Milton Santos, s/n - Ondina
Salvador, BA, Brasil, 40170-110

https://pgcomp.ufba.br
pgcomp@ufba.br

Refatorações são operações realizadas no código fonte que visam melhorar a capacidade de manutenção de um sistema de *software*. Apesar da literatura conter diversos estudos sobre refatorações, são poucos os trabalhos que investigam as motivações reportadas pelos desenvolvedores para realizar refatorações. Portanto, esta pesquisa tem o objetivo de investigar as motivações por trás das refatorações de *extract method* em sistemas reais. Para isso, foram conduzidos dois estudos experimentais, um preliminar envolvendo um único sistema de *software*, e um outro estudo onde foram considerados diversos sistemas. Os estudos foram baseados em mineração de *commits* onde ocorreram refatorações do tipo *extract method* em sistemas reais. Basicamente, buscou-se analisar as mensagens desses *commits* visando capturar as motivações dos desenvolvedores para realizar tais refatorações. Os resultados apontaram 16 motivações diferentes para a aplicação do *extract method*. Além disso, observou-se que apenas 16% das refatorações tiveram intenção explícita de melhorar a qualidade do código. Também foi identificado que as refatorações com intenção de melhorar a qualidade do código ocorreram com maior frequência em métodos com tamanho acima de 61 linhas de código e com menor frequência em métodos abaixo de 20 linhas de código.

Palavras-chave: Refatoração, *Extract Method*, Métricas de código, Mineração de repositório, Análise qualitativa.

MSC | 171 | 2023

Motivações para Aplicação da Refatoração Extract Method: Um Estudo Baseado em Mensagens de Commit

Jalisson dos Santos Henrique

UFBA

Motivações para Aplicação da Refatoração Extract Method: Um Estudo Baseado em Mensagens de Commit

Jalisson dos Santos Henrique

Dissertação de Mestrado

Universidade Federal da Bahia

Programa de Pós-Graduação em Ciência da Computação

Dezembro | 2023

Universidade Federal da Bahia
Instituto de Computação

Programa de Pós-Graduação em Ciência da Computação

MOTIVAÇÕES PARA APLICAÇÃO DA REFATORAÇÃO EXTRACT METHOD: UM ESTUDO BASEADO EM MENSAGENS DE COMMIT

Jalisson dos Santos Henrique

DISSERTAÇÃO DE MESTRADO

Salvador
11 de dezembro de 2023

JALISSON DOS SANTOS HENRIQUE

MOTIVAÇÕES PARA APLICAÇÃO DA REFATORAÇÃO EXTRACT METHOD: UM ESTUDO BASEADO EM MENSAGENS DE COMMIT

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Cláudio Nogueira Sant'Anna
Co-orientador: Marcos Barbosa Dósea

Salvador
11 de dezembro de 2023

Ficha catalográfica elaborada pela Biblioteca Universitária de Ciências e Tecnologias Prof. Omar Catunda, SIBI - UFBA.

H519 Henrique, Jalisson dos Santos

Motivações para aplicação da refatoração *extract method*: um estudo baseado em mensagens de *commit* / Jalisson dos Santos Henrique. – Salvador, 2023.

54 f.

Orientador: Prof. Dr. Cláudio Nogueira Sant'Anna.
Co-orientador: Prof. Dr. Marcos Barbosa Dósea.

Dissertação (Mestrado) – Universidade Federal da Bahia. Instituto de Computação, 2023.

1. Refatoração. 2. Extract Method. 3. Métricas de código. 4. Mineração de repositório. I. Sant'Anna, Cláudio Nogueira. II. II. Dósea, Marcos Barbosa. III. Universidade Federal da Bahia. IV. Título.

CDU 681.3

TERMO DE APROVAÇÃO

JALISSON DOS SANTOS HENRIQUE

MOTIVAÇÕES PARA APLICAÇÃO DA REFATORAÇÃO EXTRACT METHOD: UM ESTUDO BASEADO EM MENSAGENS DE COMMIT

Esta Dissertação de Mestrado foi julgada adequada à obtenção do título de Mestre em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação na Universidade Federal da Bahia.

Salvador, 11 de dezembro de 2023

gov.br Documento assinado digitalmente
CLAUDIO NOGUEIRA SANT ANNA
Data: 27/02/2024 12:42:49-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Cláudio Nogueira Sant'Anna
Universidade Federal da Bahia

gov.br Documento assinado digitalmente
CHRISTINA VON FLACH GARCIA CHAVEZ
Data: 19/04/2024 15:15:47-0300
Verifique em https://validar.iti.gov.br

Profa. Dra. Christina von Flach Garcia Chavez
Universidade Federal da Bahia

gov.br Documento assinado digitalmente
RAPHAEL PEREIRA DE OLIVEIRA
Data: 29/04/2024 09:19:16-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Raphael Pereira de Oliveira
Universidade Federal de Sergipe

À minha família.

AGRADECIMENTOS

Gostaria de expressar minha sincera gratidão a todas as pessoas que contribuíram para a realização deste trabalho de pesquisa.

À minha família, especialmente aos meus pais, Jailton Henrique e Edileuza Henrique, e à minha irmã Helen Verena, pelo apoio, paciência e suporte que me proporcionaram ao longo dessa jornada.

Ao meu orientador, Dr. Cláudio Sant'Anna, pela orientação e direcionamento ao longo desta pesquisa. Suas orientações foram fundamentais para o desenvolvimento deste trabalho.

Ao meu coorientador, Dr. Marcos Dósea, pela contribuição significativa na elaboração da pesquisa.

Aos meus amigos e colegas, pela colaboração, discussões e apoio ao longo desses anos. Um agradecimento especial para Diego Pereira, que me incentivou a fazer o curso e sempre esteve presente para auxiliar em todos os momentos.

À Universidade Federal da Bahia e a todos os seus professores, que me proporcionaram uma educação de qualidade.

Também expresso minha gratidão à Fapesb pelo apoio financeiro.

Obrigado a todos.

"Se a educação sozinha não transforma a sociedade, sem ela tampouco a sociedade muda."

—PAULO FREIRE

RESUMO

Refatorações são operações realizadas no código fonte que visam melhorar a capacidade de manutenção de um sistema de *software*. Apesar da literatura conter diversos estudos sobre refatorações, são poucos os trabalhos que investigam as motivações reportadas pelos desenvolvedores para realizar refatorações. Portanto, esta pesquisa tem o objetivo de investigar as motivações por trás das refatorações de *extract method* em sistemas reais. Para isso, foram conduzidos dois estudos experimentais, um preliminar envolvendo um único sistema de *software*, e um outro estudo onde foram considerados diversos sistemas. Os estudos foram baseados em mineração de *commits* onde ocorreram refatorações do tipo *extract method* em sistemas reais. Basicamente, buscou-se analisar as mensagens desses *commits* visando capturar as motivações dos desenvolvedores para realizar tais refatorações. Os resultados apontaram 16 motivações diferentes para a aplicação do *extract method*. Além disso, observou-se que apenas 16% das refatorações tiveram intenção explícita de melhorar a qualidade do código. Também foi identificado que as refatorações com intenção de melhorar a qualidade do código ocorreram com maior frequência em métodos com tamanho acima de 61 linhas de código e com menor frequência em métodos abaixo de 20 linhas de código.

Palavras-chave: Refatoração, *Extract Method*, Métricas de código, Mineração de repositório, Análise qualitativa.

ABSTRACT

Refactorings are operations performed on source code aimed at improving the maintainability of a software system. Although literature contains a high number of studies on refactorings, there are only few works that investigate the motivations reported by developers to perform refactorings. Therefore, this research aims to investigate the motivations behind extract method refactorings in real systems. To achieve this, we conducted two empirical studies: a preliminary study involving a single software system, and another study considering multiple real systems. The studies were based on mining software repository commits in which extract method refactorings occurred. Essentially, the goal was to analyze commit messages in order to capture developers' motivations for performing such refactorings. The results identified 16 different motivations for applying extract method. Additionally, it was observed that only 16% of the refactorings had an explicit intention of improving code quality. It was also identified that refactorings with the intention of improving code quality occurred more frequently in methods with size higher than 61 lines of code and less frequently in methods smaller than 20 lines of code.

Keywords: Refactoring, Extract Method, Code metrics, Repository mining, Qualitative analysis.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AST	*Abstract Syntax Tree*
CBO	*Coupling Between Object Classes*
DIT	*Arvore de Profundidade de Herança*
IBM	*International Business Machines*
LCOM	*Falta de Coesão dos Métodos*
LOC	*Número de Linhas de Código*
MRS	*Mineração de Repositórios de Software*
NOA	*Number of Attributes*
NOC	*Number of Children*
RFC	*Response for a Class*
SPSS	*Statistical Package for the Social Science*
TCC	*Tight Class Cohesion*
WMC	*Weighted Methods per Class*

Capítulo 1

INTRODUÇÃO

Uma característica comum em sistemas de *software* é a necessidade de evoluir e isso é impulsionado para atender às demandas dos usuários. No entanto, o processo de evolução do sistema geralmente leva a desvios do *design* original e a um aumento na complexidade. Este fenômeno é conhecido como envelhecimento do *software*, e afeta a capacidade de manutenção e evolução do sistema, tornando essas atividades mais custosas e difíceis de serem realizadas (PARNAS, 1994). Apesar de ser difícil evitar o envelhecimento do *software*, é possível lidar com esse fenômeno através da atividade de refatoração.

Refatoração é a atividade que busca melhorar a qualidade de um código existente, mas sem alterar seu comportamento (FOWLER et al., 1999). A refatoração pode ser utilizada para vários propósitos, como melhorar a capacidade de manuntenção, a compreensibilidade (BOIS; DEMEYER; VERELST, 2005), complexidade, modularidade e reusabilidade (MENS; TOURWE, 2004).

Fowler *et al.* (1999) propôs um catálogo com diversos tipos de refatorações. Dentre eles, o *extract method* é um dos mais populares e utilizado (MURPHY-HILL; PARNIN; BLACK, 2012; SILVA; TSANTALIS; VALENTE, 2016). O *extract method* tem como objetivo extrair um trecho ou fragmento de código fonte de um determinado método e levá-lo para um novo método (FOWLER et al., 1999).

Apesar da popularidade da refatoração de *extract method*, são poucos os trabalhos que investigaram as motivações por trás dessas refatorações. Silva *et al.* (2016) coletaram explicações de desenvolvedores a respeito de refatorações realizadas de modo a investigar as motivações para o uso de 12 tipos de refatorações, incluindo *extract method*. Já em Liu e Liu (2016) os autores entrevistaram desenvolvedores profissionais buscando compreender as principais motivações para as operações de *extract method*. Desse modo, a pequena quantidade de estudos desse tipo, reforça a importância de novas investigações para compreender melhor sobre as motivações para a aplicação do *extract method*. Além disso, os estudos anteriores não analisaram a relação entre as motivações para refatorar e o tamanho do método.

Compreender as motivações por trás das operações de *extract method* é importante para auxiliar os pesquisadores a entenderem melhor como desenvolver ferramentas de recomendações de refatorações com resultados mais precisos.

1.1 OBJETIVO E QUESTÕES DE PESQUISA

O objetivo dessa pesquisa é, portanto, investigar as motivações para realizar as operações de *extract method*. Além disso, uma vez que as operações de *extract method* diminuem o tamanho do método ao extrair parte dele para outro método, pretende-se também avaliar se o tamanho do método pode ter relação com a motivação do desenvolvedor para usar essa refatoração. Portanto, para atingir este objetivo, conduzimos dois estudos experimentais, onde buscamos responder as seguintes questões de pesquisa:

QP1: As refatorações de *extract method* ocorrem com maior frequência em métodos com altos valores de número de linhas de código (LOC)?

Essa questão busca analisar se, independentemente dos motivos reportados nas mensagens de *commit*, as refatorações de *extract method* ocorrem com maior frequência em métodos longos, o que pode indicar que o número de linhas de código tem influência na motivação para refatorar um método.

QP2: O que os desenvolvedores reportam como motivação para realizar o *extract method*?

Essa questão visa identificar e categorizar, com base nas mensagens de *commit*, as motivações por trás das operações de *extract method*.

QP3: Existe relação entre as motivações para refatoração de *extract method* extraídas das mensagens de *commits* e número de linhas de código do método refatorado?

Essa questão pretende verificar as motivações para refatorar métodos em diversas faixas de valores de LOC, de modo a identificar em qual faixa de valores é mais comum ocorrerem refatorações com intenção de melhorar a qualidade do código.

1.2 METODOLOGIA

Para responder as questões de pesquisa, foram realizados dois estudos experimentais, seguindo as etapas apresentadas na Figura 1.1. A princípio, foram selecionados sistemas de *software* orientado a objetos, escrito na linguagem Java, *open source* e com repositório disponível no GitHub para compor o estudo. Além disso, devido as características do estudo, buscamos ainda por sistemas que apresentassem mensagens de *commit* que descrevessem as modificações que ocorreram durante a manutenção. Após ter os sistemas alvos selecionados, foi realizado a seleção da ferramenta de mineração de refatoração capaz de identificar refatorações de *extract method*. A ferramenta utilizada neste estudo foi o RefactoringMiner 2.0, devido aos bons resultados apresentados em Tsantalis *et al.* (2020) em relação a outras ferramentas com o mesmo propósito.

Para a coleta de dados, o RefactoringMiner 2.0 foi utilizado no histórico de *commits* dos sistemas alvos do estudo, de modo a identificar as refatorações de *extract method* que ocorreram ao longo da evolução dos sistemas. Depois disso, uma análise qualitativa foi realizada nas mensagens de *commits* onde ocorreram refatorações desse tipo, buscando identificar e classificar as motivações por trás das operações de *extract method*. Essa análise foi desenvolvida através da metodologia análise temática (CRUZES; DYBA, 2011). Em seguida, uma análise quantitativa foi realizada para verificar a relação entre

Figura 1.1 Resumo da metodologia

as motivações e a métrica de número de linhas de código.

Os dois estudos desenvolvidos seguiram os mesmos procedimentos. O primeiro estudo se tratou de um estudo preliminar onde o objetivo foi verificar a viabilidade de um estudo mais amplo. Sendo assim, o *estudo prelimi*nar foi executado considerando apenas um único sistema de *software*. Um artigo descrevendo o estudo preliminar também foi apresentado no *Workshop* de Visualização, Evolução e Manutenção de *Software* (VEM) (HENRIQUE; DÓSEA; SANT'ANNA, 2021). Em seguida, a partir da experiência obtida no primeiro estudo, onde foi possível identificar os desafios, limitações e aspectos que precisariam ser mais aprofundados no estudo final, foi conduzido um segundo estudo envolvendo uma quantidade muito maior de sistemas a serem analisados.

1.3 CONTRIBUIÇÕES

Este trabalho buscou investigar as motivações por trás das operações de refatoração de *extract method*. Avaliamos também a relação entre o tamanho do método e a motivação para refatorar. Para isso, foi realizado dois tipos de análises, uma qualitativa onde mensagens de *commit* referentes a refatorações de *extract method* foram analisadas, e uma quantitativa para verificar a relação entre a métrica LOC dos métodos refatorados e a motivação para refatorar. Os resultados do estudo final apontaram 16 diferentes motivações para aplicação do *extract method*. Além disso, foi observado que as refatorações de *extract method* ocorrem com maior frequência em métodos com valores de LOC entre 19 e 34, e que a maioria das refatorações, aproximadamente 79%, ocorrem motivadas por mudanças nos requisitos, sendo que apenas 16% das refatorações que apresentaram intenção de melhorar a qualidade do código. Desse modo, esse trabalho proporcionou mais evidências experimentais sobre as motivações por trás das operações de *extract method*. Essa melhor compreensão sobre tais motivações pode ser útil para auxiliar os pesquisadores a desenvolverem ferramentas de recomendações de refatorações que forneçam maior produtividade e assertividade aos desenvolvedores de *software*.

Este estudo também disponibiliza o conjunto de dados gerados pelas ferramentas de detecção de refatorações, com o objetivo de permitir que novas pesquisas repliquem o estudo ou analisem a relação de coocorrência a partir de uma perspectiva diferente do contexto aplicado neste trabalho.

1.4 ESTRUTURA DA DISSERTAÇÃO

O restante desta dissertação está estruturado da seguinte forma. O capítulo 2 apresenta o referencial teórico, que envolve os principais conceitos referente a refatoração: *extract method*, *code smells*, métricas de *software* e mineração de repositório de *software*. Além disso, também são apresentados trabalhos relacionados a este estudo.

O capítulo 3 apresenta o estudo experimental preliminar, descrevendo o planejamento do estudo, a discussão dos resultados obtidos no estudo e as ameaças à validade.

Já o capítulo 4 apresenta o estudo experimental final com a descrição de cada etapa realizada e discute seus resultados e conclusões.

Por fim, o capítulo 5 apresenta as conclusões, contribuições da pesquisa, e os possíveis trabalhos futuros.

Capítulo 2

REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta conceitos nos quais esse estudo se baseia e que são importantes para o entendimento dessa pesquisa científica. Desse modo, serão apresentadas informações gerais sobre o tema da pesquisa, bem como abordagens e trabalhos relacionados ao campo de estudo proposto.

2.1 REFATORAÇÃO

Refatoração é a atividade que busca melhorar a qualidade de um código existente, sem alterar seu comportamento (FOWLER et al., 1999). Especificamente, são modificações que ocorrem no código fonte buscando tornar o sistema mais fácil de ser compreendido e modificado, contudo, as refatorações não modificam o comportamento externo do sistema, apenas a estrutura do código (VALENTE, 2020).

O termo refatoração foi utilizado pela primeira vez por Opdyke (1992), quando em sua tese de doutorado ele estabeleceu que refatoração representa modificações que ocorrem no código, de modo a melhorar aspectos como a legibilidade, reusabilidade e a manutenção de um sistema, sem alterar o comportamento do sistema. Sistemas de *software* evoluem continuamente, adaptando-se a novos requisitos. Conforme os desenvolvedores realizam manutenções no sistema, seja para adicionar novos recursos ou corrigir *bugs*, essas modificações, ao longo do tempo, gradativamente reduzem a qualidade do código. Um código de baixa qualidade é mais difícil de ser compreendido e modificado, impactando diretamente o desempenho da equipe de desenvolvimento. Contudo, a refatoração proporciona melhoria da estrutura interna do *software*, elevando a qualidade do código. A refatoração pode aprimorar a modularidade, a testabilidade e tornar o código mais legível e menos complexo (VALENTE, 2020).

Apesar de as refatorações possibilitarem melhorar a qualidade do código, é importante ressaltar que a atividade de refatoração também apresenta certos custos e riscos. Quando um código que já está funcionando é modificado, é possível que novos problemas surjam no sistema. Sendo assim, é necessário atenção e cuidados ao refatorar um código, de modo

a evitar o surgimento de falhas e *bugs* no sistema (FOWLER et al., 1999; YAMASHITA; MOONEN, 2013).

Devido aos seus benefícios, as refatorações se tornaram populares e importantes para garantia da qualidade do código de um *software*. Segundo Soares (2011), as refatorações representam cerca de 27% das modificações de um sistema durante seu desenvolvimento. Hoje em dia, muitos dos ambientes de desenvolvimento integrado (IDE), como Visual Studio, Eclipse e NetBeans, já contém suporte para refatoração automatizadas.

Fowler *et al.* (1999) desenvolveram um catálogo contendo vários tipos diferentes de refatorações, incluindo *Extract Method, Extract Class, Extract Variable, Inline Method, Inline Temp, Pull Up Field* e muitos outros. Em particular, esta pesquisa aborda o *extract method*, que definiremos na próxima subseção.

2.2 EXTRACT METHOD

Dentre as refatorações propostas por Fowler *et al.* (1999), o *extract method* é uma das mais populares (MURPHY-HILL; PARNIN; BLACK, 2012) e utilizadas (SILVA; TSANTALIS; VALENTE, 2016). O *extract method* tem como objetivo extrair um trecho ou fragmento de código fonte de um determinado método e levá-lo para um novo método (FOWLER et al., 1999). As Figuras 2.1 e 2.2 apresentam um exemplo de código fonte ilustrando a operação de *extract method*. A Figura 2.1 mostra o código antes dele sofrer a refatoração, e a Figura 2.2 mostra o código depois. Na Figura 2.1, os códigos entre as linhas 9 a 19 foram extraídos para um novo método nomeado de apresentaResultadoImc, apresentado na Figura 2.2.

```
1  package Model;
2
3  public class IMC {
4
5      public void calculadoraImc(double altura, double peso) {
6
7          double imc = peso / (altura * altura);
8
9          if(imc < 18.5) {
10             System.out.println("Classificação: Magreza - IMC: " + imc);
11         }else if(imc >= 18.5 && imc <= 24.9) {
12             System.out.println("Classificação: Normal - IMC: " + imc);
13         }else if(imc >= 25.0 && imc <= 29.9) {
14             System.out.println("Classificação: Sobrepeso - IMC: " + imc);
15         }else if(imc >= 30 && imc <= 39.9) {
16             System.out.println("Classificação: Obesidade - IMC: " + imc);
17         }else if(imc >= 40) {
18             System.out.println("Classificação: Obesidade Grave - IMC: " + imc);
19         }
20     }
21 }
```

Figura 2.1 Código antes do *Extract Method*

Uma das principais motivações para utilização do *extract method* é decompor métodos muito longos e difíceis de entender e manter. Métodos curtos com nomes significativos melhoram a legibilidade e também favorecem a reutilização (SILVA, 2014).

```java
package Model;

public class IMC {

    public void calculadoraImc(double altura, double peso) {

        double imc = peso / (altura * altura);

        apresentaResultadoImc(imc);
    }

    private void apresentaResultadoImc(double imc) {
        if(imc < 18.5) {
            System.out.println("Classificação: Magreza - IMC: " + imc);
        }else if(imc >= 18.5 && imc <= 24.9) {
            System.out.println("Classificação: Normal - IMC: " + imc);
        }else if(imc >= 25.0 && imc <= 29.9) {
            System.out.println("Classificação: Sobrepeso - IMC: " + imc);
        }else if(imc >= 30 && imc <= 39.9) {
            System.out.println("Classificação: Obesidade - IMC: " + imc);
        }else if(imc >= 40) {
            System.out.println("Classificação: Obesidade Grave - IMC: " + imc);
        }
    }
}
```

Figura 2.2 Código depois do *Extract Method*

A refatoração de *extract method* também pode ser utilizada para corrigir *code smells*, como Código Duplicado, onde a mesma estrutura de código em mais de um local é extraído em um único método, ou como o *code smell Feature Envy*, que ocorre quando parte de um método que utiliza vários dados de outra classe é inicialmente extraído para um novo método e, em seguida, é movido para a classe que inveja (FOWLER et al., 1999).

Além disso, estudos apontam que o *extract method* é uma das refatorações mais versáteis, utilizado para diversas finalidades distintas como facilitar a extensão, melhorar a testabilidade, habilitar recursão, entre outros (SILVA; TSANTALIS; VALENTE, 2016). Sendo assim, o *extract method* é uma refatoração muito eficaz para o desenvolvimento de *software*.

2.3 CODE SMELLS

Quando Fowler introduziu o conceito de refatoração ao final da década de 90, elas foram postas como soluções para problemas de qualidade de *design* de *software*, os *code smells*. Segundo o autor, os *smells* representam um possível problema estrutural no código-fonte e que pode ser melhorado com a refatoração (FOWLER et al., 1999).

Os *code smells* que também são conhecidos como *bad smells* (FOWLER et al., 1999), *design flawns* (MARINESCU, 2004) e anomalias de código (OIZUMI et al., 2016) são decisões ruins de *design* que refletem a baixa qualidade no código. Um sistema que apresenta uma quantidade excessiva de *code smells*, acaba tornando-se difícil de manter e evoluir (SHARMA; SPINELLIS, 2018).

Um projeto de desenvolvimento de *software*, deve produzir um sistema simples, fácil

de ser compreendido e modificado. Contudo, essas características nem sempre são alcançadas. Por conta das pressões relacionadas aos prazos de entrega, nem sempre os desenvolvedores conseguem produzir um código de qualidade adequada. Como resultado, durante o processo de desenvolvimento, pode ocorrer a inserção de fragmentos de código que comprometam os princípios da orientação a objetos. Esses fragmentos de códigos são conhecidos como *code smells*, e são capazes de degradar o *design* do projeto, resultando no aumento da complexidade do sistema (FOWLER et al., 1999).

Yamashita e Moonen (2013) colocam os *smells* como possíveis problemas de *design* estrutural em sistemas orientado a objetos que podem dificultar a atividade de manutenção do *software*. Neste sentido, os *smells* são postos como código potencialmente problemáticos, pois nem todos os *smells* são igualmente prejudiciais, e alguns deles podem nem ser prejudiciais em determinados contextos. No entanto, são vistos como indicativos de que algo está em desacordo com as boas práticas e princípios de *design*.

Apesar dos *code smells* não quebrarem o fluxo de execução do sistema, seus impactos podem trazer problemas para o produto, processo e pessoas envolvidas no projeto. Sharma e Spinellis (2018) apontaram alguns impactos que os *code smells* geram. Segundo eles, os *smells* podem afetar a manutenibilidade, a confiabilidade, a testabilidade e o desempenho do *software*. Além disso, os *smells* aumentam o esforço necessário para produzir um *software*, pois afeta a compreensibilidade, o que contribui também para o aumento do custo do produto (FOWLER et al., 1999).

Por outro lado, Tom *et al.* (2013) afirmam que uma quantidade elevada de *code smells* aumenta consideravelmente a dívida técnica do sistema, o que afeta negativamente a moral e motivação da equipe de desenvolvimento, levando a um alto desgaste. Além disso, estudos apontam que códigos com maior quantidade de *smells* estão mais propensos a apresentarem *bugs* (NASCIMENTO; SANT'ANNA, 2017).

A seguir, serão apresentados exemplos de *code smells*, com o intuito de compreender melhor suas características e particularidades. Esta pesquisa tem foco em refatorações de *extract method*, que, por sua vez, ocorrem somente em métodos. Por isso, serão apresentados exemplos de alguns *smells* que ocorrem em métodos e que podem ser eliminados ao aplicar o *extract method*.

2.3.1 Duplicate Code

Duplicate code (código duplicado) representa trechos de códigos que se repetem em dois ou mais locais em um sistema de *software* . Esse *smell* pode levar a vários problemas, como aumento desnecessário de linhas de código, aumento da complexidade, introdução de defeitos e dificuldade de manutenção.

Segundo Fowler *et al.* (1999) quando a mesma estrutura de código é visualizada em mais de um local no sistema, é um sinal de que esse código deve ser colocado em um novo procedimento. Assim, o código ficará mais organizado e fácil de manter.

A Figura 2.3 apresenta um exemplo de código duplicado. Nela é possível verificar que trechos de códigos se repetem nos métodos calcularPagamento e calcularPagamentoDesconto. Esses códigos repetidos são referentes aos cálculos para comissão e imposto, o que difere os métodos é o cálculo de desconto.

```
5     public double calcularPagamento(double totalVendas){
6         // Calcular a comissão do vendedor
7         double comissao = totalVendas * 0.10;
8         // calcular o imposto sobre a venda
9         double imposto = totalVendas * 0.05;
10
11        // calcular o pagamento final
12        double pagamento = totalVendas - imposto - comissao;
13
14        return pagamento;
15    }
16
17    public double calcularPagamentoDesconto(double totalVendas){
18        // Calcular a comissão do vendedor
19        double comissao = totalVendas * 0.10;
20        // calcular o imposto sobre a venda
21        double imposto = totalVendas * 0.05;
22        // calcular o desconto sobre a venda
23        double desconto = totalVendas * 0.02;
24
25        // calcular o pagamento final
26        double pagamento = totalVendas - imposto - comissao - desconto;
27
28        return pagamento;
29    }
```

Figura 2.3 Exemplo de código duplicado

Uma forma de resolver esse problema de código duplicado apresentado no exemplo, é remover a parte comum do código do método calcularPagamentoDesconto e chamar o método calcularPagamento dentro do método anterior, utilizando o código já existente, e apenas deixar o cálculo do desconto, como pode ser visto na Figura 2.4.

```
17    public double calcularPagamentoDesconto(double totalVendas){
18
19        double pagamento = calcularPagamento(totalVendas);
20
21        // calcular o desconto sobre a venda
22        double desconto = totalVendas * 0.02;
23
24        // calcular o pagamento final
25        pagamento -= desconto;
26
27        return pagamento;
28    }
29
30
```

Figura 2.4 Exemplo de código duplicado removido

Apesar das Figuras 2.3 e 2.4 trazerem um exemplo muito simples, ainda assim é possível verificar que ao evitar esse tipo de *code smell*, o código fica mais modular e mais fácil de modificar, pois se for necessária uma alteração nos cálculos de comissão e de imposto, essa modificação irá ocorrer apenas eu um único local, o que evita mais

trabalho e também diminui um pouco a possibilidade de introdução de defeitos durante essa manutenção.

2.3.2 Long Method

O *smell Long method* (Método longo) representa métodos que apresentam um número excessivo de linhas de código. Para Fowler *et al.* (1999) quanto maior for o método, mais difícil será entendê-lo e mantê-lo. Martin (2009) sugere que um método muito grande pode ser dividido em métodos menores e mais específicos.

A Figura 2.5 apresenta um exemplo do *long method*. O método calcularPrecoFinal possui várias responsabilidades, como o cálculo do preço final, aplicação de descontos, cálculo de frete e até a impressão do resultado do preço final. Isso faz com que o método se torne complexo e difícil de ser compreendido, principalmente se outras pessoas precisarem ler ou alterar o código.

```
5⊖    public void calcularPrecoFinal(float precoBase, int quantidade, boolean desconto,
6             int[] cupons, boolean freteGratis) {
7
8         float precoFinal = precoBase * quantidade;
9
10        if (desconto) {
11            precoFinal = precoFinal * 0.9f;
12        }
13
14        for(int cupom : cupons){
15            if(cupom == 1) {
16                precoFinal *= 0.9;
17            }
18            else if(cupom == 2) {
19                precoFinal *= 0.8;
20            }
21            else if(cupom == 3) {
22                precoFinal *= 0.7;
23            }
24            else if(cupom == 4) {
25                precoFinal *= 0.6;
26            }
27            // mais opções de cupom
28        }
29
30        if (!freteGratis) {
31            precoFinal += 10.0f;
32        }
33
34        System.out.println("O preço final é: " + precoFinal);
35
36    }
37 }
```

Figura 2.5 Exemplo do *code smell Long Method*

Esse *code smell* pode ser solucionado através da refatoração de *extract method*. O *extract method* é capaz de extrair partes do código para outros métodos menores. Isso torna o código mais fácil de ser compreendido, mantido e testado. Com a aplicação do *extract method*, é possível separar as diferentes responsabilidades do método calcularPrecoFinal em outros métodos menores e mais específicos, o que torna o código mais modular

e legível.

2.3.3 Long Parameter List

Long Parameter List (Lista Longa de Parâmetros) é um *code smell* que pode ser identificado em um método que apresenta uma quantidade excessiva de parâmetros de entrada. Essa característica pode tornar o código mais difícil de ser entendido e mantido. Isso ocorre porque, quando o número de parâmetros aumenta, a complexidade do método também cresce. Esse comportamento torna o código mais propenso a erros e dificulta a sua compreensão.

Segundo Fowler (1999) a *Long Parameter List* é um dos sinais mais comuns de que um método precisa ser refatorado. Fowler sugere que, quando um método tem mais de três ou quatro parâmetros, é hora de considerar a possibilidade de reorganizá-los de alguma forma.

De modo semelhante, Martin (2009) também destaca a importância de evitar a Lista Longa de Parâmetros em códigos. Para ele, os métodos devem ter no máximo quatro parâmetros e que, se for necessário mais que isso, é melhor reorganizá-lo em uma classe ou estrutura de dados.

```
5⊖    public void calcularTotalCompra(String nomeCliente, String enderecoCliente,
6             String telefoneCliente, String emailCliente, String nomeProduto,
7             double valorProduto, int quantidadeProduto, double valorFrete,
8             boolean desconto, String metodoPagamento) {
9
10        //código para cálcular o total da compra
11    }
12
```

Figura 2.6 Exemplo do *code smell Long Parameter List*

A Figura 2.6 apresenta um exemplo deste *code smell*. No exemplo, o método calcularTotalCompra apresenta dez parâmetros. Isso pode tornar o código mais difícil de ser entendido, mantido e reutilizado. Além disso, se novos parâmetros forem adicionados posteriormente, o código pode se tornar ainda mais difícil de entender e de utilizar corretamente.

Uma possível solução para esse *smell* seria aplicar a refatoração de *extract method*, assim, o método poderia ser dividido em submétodos menores e mais específicos, cada um contendo menos parâmetros. Essa abordagem tornaria o código mais legível, modular e fácil de ser mantido.

A Lista Longa de Parâmetros é um *code smell* que pode dificultar, principalmente, a leitura e manutenção de códigos e, por isso, é importante estar atento a esse tipo de *smell* durante o desenvolvimento do sistema. Refatorar métodos com muitos parâmetros pode melhorar a qualidade do código, tornando-o mais fácil de entender e modificar.

2.3.4 Feature Envy

Feature Envy refere-se a um método ou um trecho de código que acessa muitos dados de outras classes e poucos de sua própria classe (FOWLER et al., 1999). Fowler ainda ressalta que um método deve estar na classe que possui a maior quantidade de informações necessárias para fazê-lo funcionar. Dessa forma, se um método usa mais informações de outra classe do que a própria classe que ele se encontra, isso pode indicar que este método pode estar na classe errada. Esse comportamento pode levar a uma dependência excessiva entre as classes, o que pode dificultar a manutenção no sistema.

Para corrigir o *smell Feature Envy*, é recomendado que o método ou o trecho de código seja movido para a classe correta, onde ele pode acessar diretamente às informações necessárias.

```
1  package Model;
2
3  import java.util.List;
4
5  public class Pedido {
6
7      private List<Item> itens;
8
9      public double getValorTotal() {
10         double total = 0;
11         for (Item item : itens) {
12             total += item.getQuantidade() * item.getPreco();
13         }
14         return total;
15     }
16
17     // Outros métodos e atributos
18 }
19
```

Figura 2.7 Exemplo do *code smell Feature Envy*

A Figura 2.7, trás um exemplo do *code smell Feature Envy*. No exemplo, o método getValorTotal() da classe Pedido utiliza os atributos e métodos da classe Item. Neste exemplo, a linha 12 do método getValorTotal() é que faz acesso aos atributos da classe Item (por meio de métodos de acesso). Podemos usar a refatoração *extract method* para extrair essa linha para um método e, em seguida, mover esse novo método para a classe Item. Essa modificação tornaria o código mais coeso e diminuiria o acoplamento entre as classes.

2.4 MÉTRICAS DE CÓDIGO

À medida que os sistemas se tornaram mais complexos, foi necessário desenvolver técnicas para avaliar e medir diferentes aspectos do *software* durante todo o seu ciclo de vida. Uma métrica representa uma medida quantitativa que torna possível a medição de um produto. Pressman (2016) aponta que as métricas de *software* podem ser utilizadas para

avaliar aspectos como qualidade, desempenho, eficiência e complexidade de um sistema. Lanza e Marinescu (2010) ressaltam que as métricas são ferramentas importantes para o desenvolvimento do *software*, pois são úteis para compreender, orientar e controlar o desenvolvimento de *softwares* complexos.

De acordo com Abreu e Carapuça (1994), as métricas de *software* têm um papel fundamental que inclui, mas não se limita a, possibilitar um planejamento mais eficiente, avaliar melhorias, reduzir a imprevisibilidade, identificar precocemente possíveis problemas e avaliar a produtividade.

No livro Engenharia de *Software*, de Sommerville (2011), destaca-se que as métricas de *software* possibilitam medir atributos internos de um sistema. Segundo ele, essas métricas ajudam a avaliar a complexidade, a compreensibilidade e a manutenibilidade de um sistema ou componentes de um sistema de *software*.

Pesquisas em engenharia de *software* tem apresentado diversos tipos de métricas para analisar a estrutura e, por consequência, a qualidade do código. A Tabela 2.1, apresenta exemplos de algumas dessas métricas, conforme é visto em Bavota (2015) e Nuñez *et al.* (2017).

Tabela 2.1 Métricas de código

Métrica	**Descrição**
Lines of Code (LOC)	Número de linhas de código sem contar comentários e linhas em branco.
Number of Attributes (NOA)	Número de atributos de uma classe.
Coupling Between Object Classes (CBO)	Refere-se ao número de classes acopladas a uma classe particular.
Response for a Class (RFC)	Número de métodos que podem ser invocados por uma classe.
Number of Children (NOC)	Número de subclasses imediatas de uma determinada classe.
Weighted Method per Class (WMC)	A soma das complexidades dos métodos de uma única classe.
Tight Class Cohesion (TCC)	Número relativo de métodos de uma classe que acessam atributos em comum.

É importante ressaltar que, apesar das métricas serem importantes para o desenvolvimento de *software* de qualidade, elas não devem ser consideradas como uma solução única ou universal para o *design* de *software*. Conforme é apontado por Lanza e Marinescu (2010), boas métricas de qualidade de projeto não são necessariamente evidências de bons projetos. De modo semelhante, métricas ruins de qualidade de projeto não são necessariamente indicativos de projetos ruins. As métricas de *software* também apresentam limitações e, assim, devem ser usadas com cuidados. É essencial que o engenheiro de *software* entenda o contexto em que as métricas estão sendo aplicadas e saiba interpretar bem os resultados.

As métricas de *software* são boas para resumir aspectos específicos do código e identificar valores discrepantes em grandes volumes de dados. Elas ampliam a visão e oferecem uma base sólida para sintetizar os inúmeros detalhes que compõem o *software* (LANZA;

MARINESCU, 2010). Devido ao potencial, as métricas de código têm sido utilizadas em diversos estudos experimentais que buscam investigar vários aspectos relacionados à qualidade do código. Por exemplo, Singh e Saha (2012) conduziram um estudo com o objetivo de prever o esforço necessário para escrever testes para uma classe, utilizando métricas de código. Por outro lado, Nisa e Ahsan (2015) procuraram, por meio das métricas, desenvolver um modelo de previsão de falhas para *software*.

Além de possibilitar analisar a evolução da qualidade de um sistema, Marinescu (2004) afirma que através das métricas é possível detectar desarmonias estruturais no *software*. Desse modo, as métricas permitem analisar problemas de qualidade do código e identificar oportunidades de refatorações.

Por exemplo, a métrica LOC de um determinado método, quando apresenta um valor elevado, pode indicar que este método representa um problema de qualidade código, sendo um *code smell* do tipo *Long Method*. Sendo assim, a métrica poderia estar indicando que esse método poderia ser alvo da refatoração do tipo *extract method* para eliminar o *smell* e, consequentemente, melhorar a qualidade do código. Além disso, Lanza e Marinescu (2010) demostraram que as métricas de código-fonte também podem ser utilizadas de maneira combinadas para identificar componentes de *software* de baixa qualidade. Essa técnica de utilizar métricas de maneira combinada é conhecida como estratégia de detecção, e é uma forma de melhorar a identificação de *code smells*, através da utilização de métricas de *software*.

2.4.1 Valores Limiares de Métricas

Como foi dito anteriomente, as métricas possibilitam identificar problemas na qualidade do código. Contudo, para que isso ocorra é necessário que exista um ponto de referência para indicar que determinada característica de fato representa ou não um problema de qualidade. O valor limiar é o ponto em que uma métrica pode ser dividida em mais de uma região. A partir da região em que um valor da métrica se encontra, é possível fazer uma avaliação sobre a característica medida (MARINESCU, 2004).

Assim, um valor limiar pode indicar se uma determinada característica do código representa um problema de qualidade. Sendo assim, muitas ferramentas de detecção de *code smells* são baseadas em métricas. No entanto, a definição de um valor limiar não é uma tarefa muito simples. Por exemplo, qual valor de métrica pode definir que um método é longo? Em muitos casos, a definição do valor limiar é algo subjetivo e complexo. Com isso, essas ferramentas acabam sendo propensas a apresentar muitos falsos-positivos, pois dependem muito dos valores limiares das métricas (SHARMA; SPINELLIS, 2018).

Desse modo, a comunidade de engenharia de *software* tem realizado muitas tentativas para identificar valores limiares ideais, de modo a contribuir para a produção de ferramentas de detecção automática de *code smells* e de recomendação de refatorações que forneçam maior assertividade em seus resultados (DÓSEA; SANT'ANNA; SILVA, 2018; FIGUEIREDO et al., 2008; FONTANA et al., 2015; VALE; FERNANDES; FIGUEIREDO, 2019; VALE; FIGUEIREDO, 2015).

2.5 MINERAÇÃO DE REPOSITÓRIO

A Mineração de Repositórios de *Software* (MRS) busca extrair informações relevantes sobre sistemas de *software* a partir da análise de grandes quantidades de dados de *software*. Hoje, repositórios de *software* como GitHub tem uma imensa quantidade de dados produzidos em projetos de desenvolvimento de *software* disponíveis gratuitamente. A análise desses dados é muito relevante para pesquisas em engenharia de *software*, pois possibilita estudar de forma extensiva o desenvolvimento de *software*, a partir de projetos reais.

Segundo Luzgin e Khold (2020), no ano de 2019, o GitHub continha mais de 100 milhões de projetos hospedados, sendo que esses projetos apresentam diversos tipos de fontes de informações, como o código, revisões de código, históricos de alterações de código, discussões, entre outros. Assim, várias estratégias de mineração podem ser aplicadas para produzir pesquisas em engenharia de *software*. Atualmente, diversas pesquisas relacionadas a refatoração de código vem sendo realizadas através da análise de dados obtidos através da mineração de repositório, buscando compreender melhor as operações de refatoração (ALOMAR; MKAOUER; OUNI, 2019; CEDRIM et al., 2017; PAIXÃO et al., 2020; SILVA, 2014).

Visto isso, a mineração de repositório de *software* representa uma oportunidade interessante para-se investigar aspectos relacionados a qualidade de *software*. Nesta pesquisa, buscou-se minerar refatorações de *extract method* de projetos de *software* disponíveis no GitHub.

2.5.1 Ferramentas de Mineração de Refatoração

As ferramentas de mineração de refatoração são *softwares* com o objetivo de identificar as refatorações que ocorreram ao longo da evolução de um determinado sistema. Essas ferramentas são cruciais para o desenvolvimento de diversos estudos científicos sobre qualidade de código, evolução do sistema, testes e falhas. Elas possibilitam aos cientistas estudar a prática de refatoração em sistemas de *software*, permitindo uma melhor compreensão do processo de desenvolvimento de sistemas e o aprimoramento da qualidade do código. Ao longo dos anos, a literatura tem apresentado diversas ferramentas para mineração de refatoração.

Dig *et al.* (2006) desenvolveram o RefactoringCrawler, uma ferramenta capaz de detectar refatorações realizadas durante a evolução dos componentes de um sistema escrito em Java. O RefactoringCrawler aplica uma combinação de análise sintática rápida para detectar candidatos a refatoração e uma análise semântica mais precisa para refinar os resultados. A análise sintática é baseada em técnicas de recuperação de informações, onde o algoritmo procura encontrar fragmentos de código semelhantes entre diferentes versões de um projeto. Já a análise semântica se baseia em grafos de referência que representam as relações entre entidades de nível de origem, como chamada entre métodos. Essa análise leva em conta o relacionamento semântico entre as entidades candidatas a fim de determinar se elas representam uma refatoração. O RefactoringCrawler é capaz de detectar sete tipos diferentes de refatorações.

Kim *et al.* (2010) apresentaram o Ref-Finder, um *plug-in* do Eclipse capaz de identificar automaticamente 63 tipos de refatorações do catálogo de Fowler. O Ref-Finder recebe duas versões de um sistema como entrada e extrai fatos lógicos sobre a estrutura sintática do programa. Através da aplicação do mecanismo de programação lógica de Tyruba, descrito em Volder (1998), a ferramenta utiliza consultas lógicas predefinidas para detectar diferenças no programa que correspondem a cada tipo de refatoração.

Silva e Valente (2017) desenvolveram o RefDiff, uma abordagem automatizada que detecta refatorações realizadas em um sistema através da análise do histórico de versões do sistema. Para isso, o RefDiff utiliza uma combinação de heurísticas baseadas em análise estática e similaridade de código, permitindo a detecção de 13 tipos de refatorações. Em 2020, os autores lançaram o RefDiff 2.0, uma atualização que aprimorou o algoritmo de detecção de refatoração e possibilitou a detecção de refatorações em múltiplas linguagens de programação. O RefDiff 2.0 foi a primeira ferramenta a possibilitar a detecção de refatorações em sistemas escritos com outras linguagens de programação além do Java. Para demonstrar isso, os autores implementaram módulos de extensão no *plug-in* da ferramenta para três linguagens: Java, JavaScript e C (SILVA et al., 2021).

Tsantalis *et al.* (2018) desenvolveram o RefactoringMiner, uma ferramenta de mineração de refatoração com capacidade de identificar 15 tipos de refatoração. O RefactoringMiner, também conhecido como RMiner pelos autores, recebe como entrada duas versões de um sistema Java e gera uma lista operações de refatorações aplicadas entre essas versões. Em um repositório de controle de versão baseado em Git, cada *commit* (operação que registra alterações em um projeto) representa uma versão do sistema. O RMiner utiliza um algoritmo de correspondência de instruções baseado em Árvore Sintática Abstrata (AST) e um conjunto de regras de detecção que abrangem os 15 tipos de refatorações. Esse algoritmo emprega duas estratégias: a abstração, que lida com modificações na estrutura de AST das declarações devido às refatorações, e a argumentação, que lida com alterações nas subexpressões dentro das declarações devido à parametrização.

Posteriormente, Tsantalis *et al.* (2020) lançaram o RefactoringMiner 2.0, uma nova versão da ferramenta com grandes melhorias incrementadas. A versão 2.0 pode detectar até 40 tipos de refatorações para 6 tipos diferentes de elementos de código, como métodos, pacotes, declarações de tipo, campos, variáveis e referências de tipo. Além de suportar refatorações de baixo nível que ocorrem dentro dos métodos, o RefactoringMiner 2.0 traz um diferencial em relação as outras ferramentas de mineração de refatoração. Ele é capaz de analisar apenas os arquivos adicionados, excluídos ou alterados entre as versões do sistema. Ferramentas como Ref-Finder e RefactoringCrawler analisam todos os arquivos em duas versões. Essa característica torna o RefactoringMiner 2.0 mais eficiente, visto que tem menos elementos de código para analisar e comparar, o que melhora o desempenho. Na avaliação proposta pelos autores da ferramenta, os resultados indicaram que o RefactoringMiner 2.0 atinge a maior média de precisão (99,6%) e recall (94%) entre as ferramentas competitivas (RefactoringMiner 1.0, RefDiff 1.0 e RefDiff 2.0), e é 2,6 vezes mais rápido em relação à segunda ferramenta competitiva (RefDiff 2.0).

2.6 TRABALHOS RELACIONADOS

Alguns estudos já analisaram mensagens de *commits* buscando compreender melhor a atividade de refatoração, dentre eles, destacamos Stroggylos e Spinellis (2007), Murphy-Hill *et al.* (2012), Rebai *et al.* (2020), Alomar *et al.* (2019) e Aalizaded (2021). Stroggylos e Spinellis (2007) pesquisaram por palavras provenientes do verbo refatorar, como refatoração e refatorado, buscando identificar *commits* relacionados a refatoração. O estudo buscou examinar como as métricas de *software* são afetadas pelo processo de refatoração, a fim de avaliar se essa atividade de fato era utilizada para melhorar a qualidade do *software*. As métricas consideradas no estudo foram: WMC, DIT (Árvore de Profundidade de Herança), NOC, CBO, RFC e LCOM (Falta de Coesão dos Métodos). A pesquisa mostrou que, embora as refatorações ocorressem buscando melhorar a qualidade do código, as métricas indicaram que esse processo geralmente tem resultado oposto. Apesar do nosso trabalho também ter buscado identificar atividades de refatorações em *commits* e analisar uma métrica de código, nosso trabalho se diferecia desse, pois a análise das mensagens de *commit* foi feita manualmente, foi considerado apenas um tipo de refatoração (*extract method*) e buscamos identificar as motivações para as refatorações ocorrerem.

Murphy-Hill *et al.* (2012) também utilizaram uma abordagem baseada em palavras-chave para identificar atividades de refatoração em mensagens de *commit*. No estudo, foi considerada uma lista de 13 palavras-chaves, sendo elas: *refactor, restruct, clean, not used, unused, reformat, import, remove, replace, split, reorg, rename,* e *move*. Os pesquisadores concluíram que as mensagens de *commit* não são indicadores confiáveis de atividades de refatoração. Isso foi atribuído ao fato de que os desenvolvedores não documentam consistentemente as atividades de refatoração nas mensagens de *commit*. Na nossa pesquisa, as mensagens de *commit* foram analisadas qualitativamente, visando identificar a intenção dos desenvolvedores para aplicarem as refatorações.

Rebai *et al.* (2020) analisaram mesnsagens de *commit*, buscando extrair informações sobre a intenção dos desenvolvedores ao aplicar as refatorações no código. A pesquisa apresentou uma ferramenta de recomendação de refatoração, na qual as recomendações são baseadas nas mensagens de *commit*, diferenciando-se de outras abordagens que consideram somente a eliminação de antipadrões e a melhoria da qualidade do código. Essa abordagem visa capturar as necessidades dos desenvolvedores a partir de suas mensagens de *commit* e propor refatorações para aprimorar suas modificações, a fim de resolver melhor os problemas de qualidade do código. Os resultados apresentaram evidências convincentes sobre o valor de usar informações das mensagens de *commit* para recomendar refatorações. A análise das mensagens foi realizada de forma automática considerando as mensagens que continham problemas de qualidade ou refatorações com base em uma lista de 87 palavras-chave.

Alomar *et al.* (2019) buscaram identificar como os desenvolvedores documentam suas atividades de refatorações durante o ciclo de vida do *software*. Para isso, as mensagens de *commit* foram analisadas em busca de informações relevantes em relação às refatorações aplicadas. Os resultados mostraram que os desenvolvedores tendem a mencionar explicitamente a melhoria de atributos de qualidade e *code smells* nas mensagens de *commit*. Além disso, foi visto que mensagens com esses conteúdos tendem a ter atividades de

refatorações mais significativas do que aquelas sem. Nosso estudo também buscou analisar as mensagens de *commit* visando identificar como as refatorações de *extract method* eram descritas. Contudo, nossa abordagem objetiva extrair as motivações por trás das refatorações, enquanto que essa pesquisa visava investigar como as refatorações eram descritas pelos desenvolvedores.

Aalizadeh (2021) propôs o Motivation Extractor, um sistema baseado no RefactoringMiner, que utiliza uma abordagem sensível ao contexto para detectar automaticamente 11 motivações que impulsionam a aplicação de operações de *extract method*. Essa abordagem também verifica as mensagens de *commit* das refatorações. As motivações foram baseadas nos resultados de Silva *et al.* (2016). O autor conduziu um estudo em larga escala com 325 repositórios de sistemas escrito em Java, buscando detectar automaticamente as motivações das instâncias de refatoração. Para isso, foi desenvolvido regras de detecção de motivação para detectar as motivações do desenvolvedor com base no contexto de uma operação de refatoração no *commit*. Os resultados apontaram que as quatro principais motivações para a refatoração de *extract method* são: extração de método reutilizável, remover duplicações, facilitar a extensão e decomposição de método para melhorar a legibilidade. Esse trabalho relacionado buscou estabelecer regras para a detecção de motivações já existentes na literatura. Assim, o estudo não teve a possibilidade de identificar novas motivações para a refatoração de *extract method*.

Alguns estudos na literatura já se propuseram a investigar as motivações por trás das refatorações. São eles: Paixão *et al.* (2020), Pantiuchina *et al.* (2020), Liu e Liu (2016) e Silva *et al.* (2016). Os dois últimos trabalhos são os que mais se assemelham ao nosso, pois também investigam as motivações para o uso do *extract method*.

Tabela 2.2 Motivações para *extract method*, retirado de Liu e Liu (2016)

Motivação	**Descrição**
Decomposição de métodos	Utilização do *extract method* para decompor métodos longos e complexos. Esses métodos geralmente são difíceis de ler ou modificar, portanto, o *extract method* pode ajudar a melhorar a legibilidade e manutenção do *software*.
Resolução do clone	Aplicação do *extract method* para remover código duplicado. Extrair código duplicado como um método modularizado é um meio eficaz de remover código duplicado.
Reutilização atual	Aplicação do *extract method* para facilitar a reutilização atual de um fragmento de código durante a evolução do *software*.
Reutilização futura	Em alguns casos, um fragmento de código pode ser extraído porque o fragmento provavelmente será reutilizado no futuro.

Paixão *et al.* (2020) buscaram compreender o contexto e as motivações nos quais os desenvolvedores realizam rafatorações durante o processo de revisão de código. Para isso, um estudo experimental foi desenvolvido, onde mensagens de *commit* e discussões entre desenvolvedores foram inspecionadas. Os resultados apontaram que, em apenas 31% das revisões de código que empregaram operações de refatorações, os desenvolvedores tinham intenção explícita de refatorar. Além disso, o estudo identificou cinco intenções para os desenvolvedores refatorarem. Elas foram: adicionar ou aprimorar um recurso, reestruturação do sistema, corrigir *bug*, atualização da plataforma e mesclar *commit*.

Contudo, a abordagem utilizada no estudo considerou as refatorações de modo geral, não verificando tipos específicos de refatorações.

Pantiuchina *et al.* (2020) executaram um estudo em larga escala para investigar por que os desenvolvedores realizam refatorações em projetos de código aberto. O estudo utilizou uma abordagem quantitativa e qualitativa. Nele, foram exploradas 287.813 operações de refatoração de 25 tipos diferentes realizadas em 150 sistemas. A pesquisa analisou manualmente as discussões dos desenvolvedores para identificar as razões para realizar as operações de refatorações. Os resultados apontaram 67 motivações por trás das refatorações.

Os resultados também apontaram que as refatorações ocorrem mais por mudanças de requisitos e menos para eliminar *code smells*. A principal diferença para o nosso trabalho é que analisamos também a métrica LOC dos métodos refatorados para avaliar a relação entre as motivações e o tamanho do método. Além disso, a nossa coleta de dados foi realizada apenas com mensagens de *commits*.

Tabela 2.3 Motivações para *extract method*, retirado de Silva *et al.* (2016)

Motivação	**Descrição**
Extração de método reutilizável	Extrair um pedaço de código reutilizável para um único local e chamar o método extraído em vários locais.
Introduzir assinatura de método alternativo	Introduzir uma assinatura alternativa para um método existente e tornar o método original delegado ao extraído.
Decomposição de método para melhorar a legibilidade	Extrair um trecho de código com uma funcionalidade distinta em um método separado para facilitar o entendimento do método original.
Facilitar a extensão	Extrair um pedaço de código em um novo método para facilitar a implementação de um recurso ou correção de *bug*.
Remover duplicação	Extrair um pedaço de código duplicado de vários lugares e substituir os trechos duplicados por chamadas para o método extraído.
Preservar a compatibilidade com versões anteriores	Novo método que substitui um existente para melhorar seu nome ou remover parâmetros não utilizados. O método original é preservado para compatibilidade com versões anteriores.
Melhorar a testabilidade	Extrair um trecho de código em um método separado para permitir seu teste de unidade isoladamente do restante do método original.
Habilitar substituição	Extrair um trecho de código em um método separado para permitir que as subclasses substituam o comportamento extraído por um comportamento mais especializado.
Habilitar recursão	Extrair um trecho de código para torná-lo um método recursivo.
Introduzir o método de fábrica	Extrair uma chamada de construtor em um método separado.
Introduzir operação assíncrona	Extrair um trecho de código em um método separado para executá-lo em uma *thread*.

Para investigar as motivações para a refatoração de *extract method*, Liu e Liu (2016) realizaram entrevistas com 25 desenvolvedores, buscando compreender como são conduzidas as refatorações desse tipo. Os resultados apontaram que a reutilização atual, decomposição de métodos longos, resolução de clones e a reutilização futura são as principais motivações para o uso de *extract method*. A Tabela 2.2 apresenta a descrição de cada motiviação encontrada no estudo. Além disso, os autores buscaram validar os resul-

tados, analisando o histórico de refatoração de sete sistemas de código aberto. Na nossa pesquisa, optamos por utilizar uma abordagem diferente desse estudo para identificar as motivações para o *extract method*, ao invés de realizar entrevistas optamos por analisar mensagens de *commit*.

Silva *et al.* (2016) buscaram investigar as motivações reais por trás das operações de refatorações aplicadas por desenvolvedores. Para isso, os autores monitoraram projetos Java hospedados no GitHub para detectar operações de refatorações realizadas em 124 sistemas. Quando uma refatoração era detectada, os autores enviavam um e-mail ao desenvolvedor responsável pela refatoração, buscando coletar informações sobre os motivos por trás das decisões de refatorar o código. Além disso, as mensagens de *commits* também eram examinadas para identificar a motivação por trás da refatoração. Os resultados apontaram 267 ocorrências de refatorações, sendo que a refatoração de *extract method* correspondeu a 138 delas. Assim, foi desenvolvido um catálogo com 44 motivações distintas para 12 tipos de refatorações. Como o *extract method* foi o tipo de refatoração que ocorreu com maior frequência, a Tabela 2.3 apresenta as 11 motivações encontradas na pesquisa para o uso do *extract method*.

Capítulo 3

ESTUDO PRELIMINAR

Este capítulo descreve nosso estudo preliminar, apresentando os procedimentos adotados no estudo e os resultados obtidos. O objetivo deste estudo preliminar foi investigar as motivações para realizar as operações de *extract method* e avaliar se o tamanho do método tem relação com a motivação do desenvolvedor para usar a refatoração de *extract method*.

3.1 PROJETO DO ESTUDO

Com intuito de responder às questões de pesquisa e verificar a viabilidade deste projeto, foi conduzido um estudo experimental preliminar. Nesta seção, serão apresentadas em detalhe as configurações adotadas no estudo, incluindo os parâmetros específicos, os métodos de coleta e análise de dados, assim como as etapas de execução e os recursos utilizados.

3.1.1 Sistema Alvo

A primeira etapa para realização deste estudo foi a escolha do sistema a ser analisado. Desse modo, buscamos um sistema orientado a objetos, desenvolvido na linguagem Java e com código disponível no repositório GitHub. A escolha da linguagem Java ocorreu devido ao fato de que as principais ferramentas de mineração de refatoração são específicas para sistemas nessa linguagem. Além disso, como nosso estudo se baseia em mensagens de *commit*, buscamos ainda por um sistema onde tais mensagens descrevessem relativamente bem o que foi feito naquele *commit*. O sistema utilizado nesse estudo foi o WebAnno[1], descrito em Castilho *et al.* (2016), uma ferramenta web de anotação de propósito geral para uma ampla gama de anotações linguísticas, incluindo várias camadas de anotações morfológicas, sintáticas e semânticas. Esse *software* possui 802 classes, 5109 métodos e 48464 linhas de código. Como esse é um estudo preliminar, decidimos ainda usar apenas um sistema.

[1]https://github.com/webanno/webanno

3.1.2 Apoio Ferramental

Para identificar as refatorações realizadas no sistema alvo, foi utilizada uma ferramenta de mineração de refatorações chamada *RefactoringMiner* 2.0, desenvolvida por Tsantalis *et al.* (2020). A *RefactoringMiner* 2.0 analisa as alterações de código em repositório Git para detectar refatorações aplicadas ao longo da evolução do sistema. Ela suporta a detecção de até 40 tipos de refatoração para seis tipos diferentes de elementos de código. Entretanto, este estudo considerou apenas a refatoração *extract method*. Mais especificamente, consideramos o que a *RefactoringMiner* 2.0 identifica como *EXTRACT OPERATION* ou *EXTRACT AND MOVE OPERATION*. As duas operações referem-se ao *extract method*, mas, na segunda, além de ser extraído, o método é movido para outra classe. Em um estudo, realizado pelos autores da ferramenta, que envolveu 7226 instâncias de refatorações diferentes (TSANTALIS; KETKAR; DIG, 2020), ela apresentou precisão de 99,6% e recall de 94%, de acordo com a validação de quatro especialistas em refatoração. Esses resultados foram melhores que todas as outras ferramentas avaliadas.

3.1.3 Procedimentos do Estudo

O estudo foi dividido em quatro etapas distintas. A Figura 3.1 apresenta as etapas que foram seguidas para realização o estudo experimental.

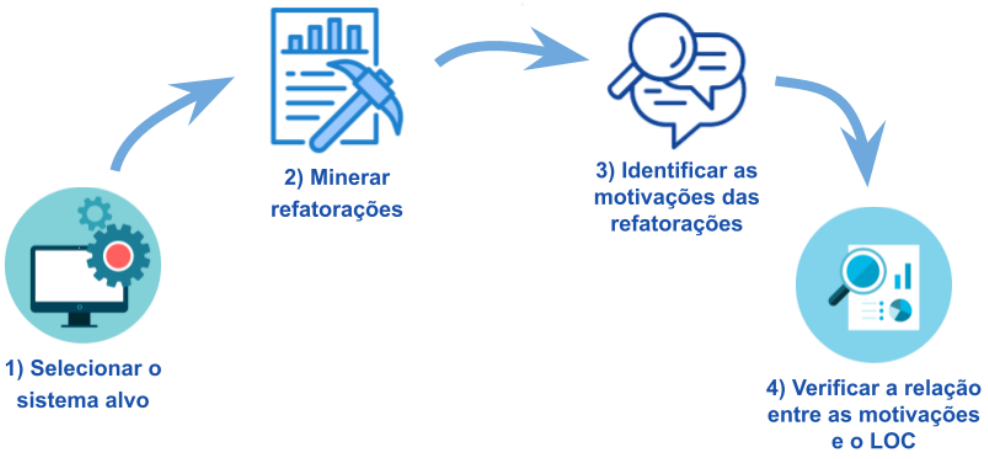

Figura 3.1 Etapas do Estudo

Após selecionar o sistema alvo na primeira etapa do estudo, a segunda etapa foi iniciada com objetivo de coletar os dados necessários para a análise. Para isso, a *RefactoringMiner* 2.0 foi utilizada no histórico de *commits* do sistema alvo desse estudo. Com isso, foram identificadas as refatorações de *extract method* que ocorreram ao longo da evolução do sistema. A ferramenta captura as refatorações e apresenta informações referentes a elas como: nome do método refatorado; nome da classe; identificador do *commit*; tipo de refatoração; número de linhas de código do método refatorado; nome do

método para o qual o trecho de código foi movido; e a mensagem de *commit*.

Ao todo, a ferramenta detectou 411 refatorações de *extract method* que ocorreram durante 110 *commits* existentes no sistema. Com isso, a terceira etapa buscou identificar as motivações para essas refatorações ocorrerem. Assim, as mensagens de *commit* foram analisadas qualitativamente, de modo a extrair as motivações por trás dessas refatorações. Para isso, a análise temática foi utilizada como metodologia (CRUZES; DYBA, 2011). A análise temática é uma abordagem de pesquisa qualitativa para a análise de dados. Este método enfatiza a identificação, análise e interpretação de padrões e significados (temas) em dados textuais qualitativo (CRUZES; DYBA, 2011). Uma característica da análise temática é a sua flexibilidade. Esse método possibilita aos pesquisadores diferentes abordagens, tanto a análise exploratória, investigando temas que surgem dos dados, quanto para análise dedutiva, usando temas predefinidos com base em teorias existentes. É importante ressaltar que os temas representam conceitos que surgem a partir da análise dos dados.

A análise temática foi desenvolvida da seguinte forma pelo próprio autor do trabalho. Primeiramente, foi realizada uma leitura inicial de todas as mensagens de *commit*. Depois, foram atribuídos códigos para cada mensagem, esses códigos são palavras que podem sugerir a motivação da refatoração. Em seguida, definiu-se temas com base nos códigos gerados anteriormente. Então, uma revisão dos temas foi realizada buscando juntar temas semelhantes e, finalmente, foram definidos os temas finais. Os temas representam as motivações para refatorar o método.

Para exemplificar melhor as etapas da análise temática, suponhamos que uma mensagem de *commit* esteja da seguinte forma: "*Removi o código que estava se repetindo em métodos na classe ModelConfig*". Durante a etapa de atribuição de códigos da análise temática, é possível atribuir o código 'remover código repetido' para esta mensagem. No entanto, dependendo da descrição de outras mensagens de *commit*, também é possível atribuir outros códigos como 'remoção de código duplicado' ou 'retirar código redundante'. Na etapa que busca definir temas com base nos códigos gerados, é possível determinar que o tema 'Remover Duplicação' seja apropriado para representar as motivações por trás das refatorações cujas mensagens foram atribuídas aos três códigos mencionados anteriormente. A etapa de revisão de temas é justamente para verificar se, entre os temas (motivações) definidos, existem temas com nomes diferentes que remetem ao mesmo objetivo. Nesse caso, é analisada a possibilidade unir temas semelhantes, resultando na definição dos temas finais. Dessa forma, foram definidas as motivações por trás das operações de *extract method*, por meio da análise das mensagens de *commit*.

A análise também permitiu classificar as motivações encontradas na análise das mensagens de *commit* em duas categorias. Com base nas descrições das mensagens, foi identificado que havia refatorações em que as mensagens indicavam intenção de melhorar a qualidade do código e refatorações em que essa intenção não estava presente. Assim, os códigos 'Sim' e 'Não' foram atribuídos para representar as categorias, respectivamente.

Por fim, a quarta etapa consistiu em uma análise quantitativa sobre os métodos refatorados, verificando a métrica de números de linhas de código dos métodos e relacionando-a com a motivação da refatoração. Desse modo, buscamos em um primeiro momento verificar a distribuição dos valores de LOC dos métodos refatorados, visando identificar em

quais faixas de LOC as refatorações ocorrem com maior frequência.

Em seguida, foi analisada a relação entre a métrica de LOC com as categorias encontradas nas mensagens analisadas, buscando investigar em quais níveis de LOC as refatorações com intenção de melhorar a qualidade do código ocorrem com maior frequência. Todos os dados considerados no estudo estão disponíveis publicamente[1].

3.2 RESULTADOS

Nessa seção são apresentados os resultados obtidos no estudo experimental visando responder as questões de pesquisa.

3.2.1 Relação entre Refatorações e LOC

Esta seção procura responder QP1: As refatorações de *extract method* ocorrem com maior frequência em métodos com valores de LOC alto? Para isso, os valores de LOC dos métodos refatorados foram analisados através de um gráfico de distribuição de frequência, visto na Figura 3.2. No gráfico, os valores dos intervalos do eixo x foram gerados automaticamente através da ferramenta Google Sheets. Desse modo, é possível verificar que as refatorações ocorreram em maior quantidade em métodos com valores de LOC na faixa entre 29 a 42. Contudo, é possível observar que um número grande de refatorações ocorreu em métodos com valores de LOC abaixo de 29 e também com valores acima de 42.

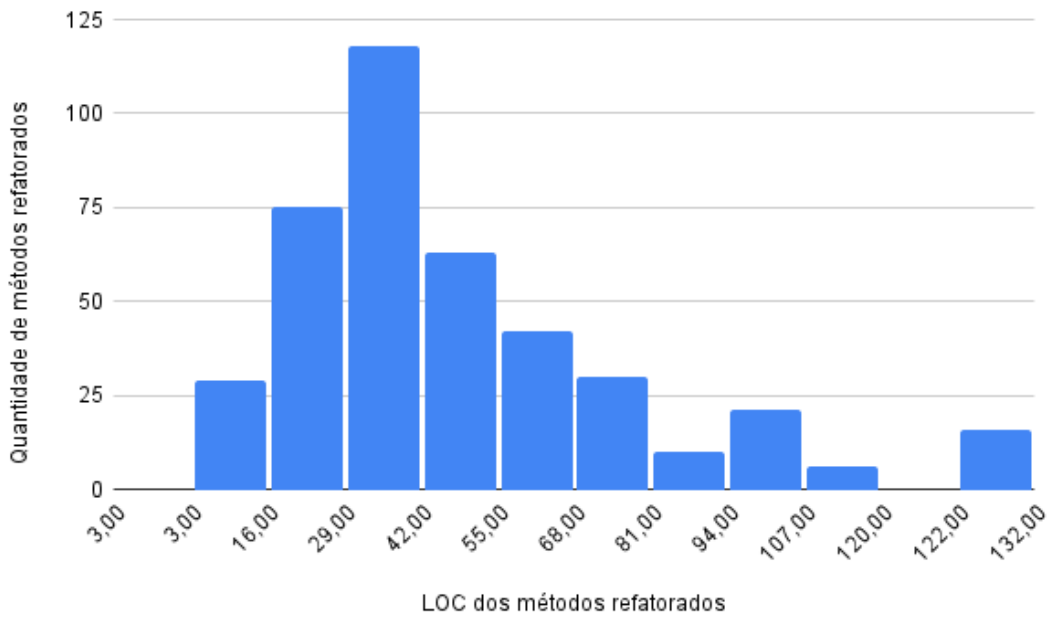

Figura 3.2 Distribuição de frequências dos valores de LOC dos métodos refatorados

[1]https://doi.org/10.5281/zenodo.5234317

Curiosamente, esses resultados revelam que as refatorações não ocorreram apenas em métodos com valores de LOC altos, ou seja, em métodos longos, que a literatura considera como um *code smell* que pode ser alvo de *extract method* para ser removido (FOWLER et al., 1999). Muitas das refatorações também ocorreram em métodos com valores de LOC baixo. Visto isso, identificar as motivações por trás das refatorações de *extract method* pode ajudar a compreender se há diferentes razões para os desenvolvedores refatorarem métodos com LOC baixo e métodos com LOC alto.

3.2.2 Motivações para Realizar Extract Method

Esta seção visa responder QP2: O que os desenvolvedores reportam como motivação para realizar o *extract method*? A partir da codificação e análise das mensagens de *commit* dos métodos refatorados, foi possível identificar as motivações que levaram à utilização do *extract method*. A Tabela 3.1 apresenta as onze diferentes motivações encontradas. A Tabela 3.1 também classifica as motivações nas categorias 'Sim' e 'Não', onde 'Sim' representa as refatorações onde as motivações apresentavam intenção explícita de melhorar a qualidade do código e 'Não' indica as refatorações que não tinham intenção explícita de melhorar a qualidade do código. Além disso, para cada motivação é apresentado a descrição e o número de ocorrências.

Tabela 3.1 Motivações para utilização do *extract method*

Categoria	Motivação	Descrição	Ocorrência	Total
Sim	Remover duplicação	Remover trechos de código que se repetem em vários locais no sistema.	38 (9,25%)	35%
Sim	Limpeza de código	Remoção de trechos de código legado ou desnecessário.	37 (9%)	
Sim	Organizar código	Modificações para organizar a estrutura do sistema, como mover trechos de um método para outro método ou classe mais adequada.	34 (8,27%)	
Sim	Melhorar a legibilidade	Alteração no método para torná-lo mais legível, ou seja mais fácil de ler.	15 (3,65%)	
Sim	Melhorar a testabilidade	Refatorações conduzidas a fim de tornar o método mais fácil de se testar.	14 (3,41%)	
Sim	Reduzir o método	Refatorar o método para resolver o problema de *smell Long Method*.	5 (1,22%)	
Não	Atualização de *Features*	Refatorações que ocorreram durante a manutenção das funcionalidades.	79 (19,22%)	39%
Não	Adicionar *Features*	Refatorações que ocorreram durante a inserção de novas funcionalidades.	37 (9%)	
Não	Correção de *Bug*	Refatorações realizadas durante o processo de correção de *bugs*.	21 (5,11%)	
Não	Melhorar o Desempenho	Refatorações que ocorreram em ações para melhorar a eficiência do processamento de determinadas funcionalidades.	13 (3,16%)	
Não	Remoção de *Features*	Remoção de funcionalidades no sistema.	12 (2,92%)	
	Sem Categoria	Refatorações das quais não foi possível identificar seus objetivos.	106 (25,79%)	26%

Sim: refatorações com intenção de melhorar a qualidade do código

Não: refatorações sem intenção de melhorar a qualidade do código

Dentre as motivações identificadas, seis delas visam melhorar a qualidade do código. Dentro desse grupo, a motivação 'Remover duplicação' é a que aparece com maior frequência (38 vezes). Ela representa refatorações para excluir trechos de código redundantes que se repetem em mais de um local. Alguns exemplos de trechos de mensagens de *commit* ilustram essa motivação: "*Extração de código redundante em métodos auxiliares como getCorrctionCas() ou actionRefreshDocument()*" ou "*Código movido para carregar restrições no serviço de repo em vez de tê-lo redundantemente em CorrectionPage, AnnotationPage, AutomationPage e CurationPage*".

A segunda motivação mais frequente é a 'Limpeza de código' (37 ocorrências), que representa as refatorações que ocorreram buscando remover trechos de códigos desnecessários ou que se encontravam legados. Isso pode ser visto em mensagens como "*Código legado removido*" e também em "*Código refatorado para torná-lo mais conciso e remover código desnecessário*".

Por sua vez, as refatorações para 'Organizar o código' aparecem em terceiro lugar (34 ocorrências) dentre aquelas com intenção de melhorar a qualidade do código. Tal motivação visa organizar a estrutura do código, movendo métodos ou trechos de código para locais mais adequados no sistema. Mensagens como "*Alguns métodos foram movidos para ficarem mais próximos de métodos semelhantes*" e "*Mover o método de renderização de BratAnnotationEditor para AnnotationEditorBase*" evidenciam essa motivação.

Outra motivação importante foi 'Melhorar a legibilidade', que ocorre em 15 ocasiões. São refatorações com intenção de tornar o código mais fácil de ser lido. Isso pode ser visto na mensagem: "*Refatorando a estrutura do projeto - rumo a actionSpanAnnotation() mais compreensível*".

Além disso, as mensagens de *commit* possibilitaram verificar que muitas das refatorações que ocorreram não tinham a intenção de melhorar a qualidade do código. Essas refatorações ocorreram durante algumas manutenções que aconteceram no sistema. Dentro desse grupo, a 'Atualização de *Features*' aparece com maior frequência (79 vezes). Elas são refatorações que ocorreram quando uma funcionalidade do sistema estava sendo modificada. A mensagem a seguir exemplifica esse comportamento: "*Atualizar o painel do editor de recursos para levar em conta os efeitos das restrições, como slots adicionados automaticamente para vincular recursos*".

Em seguida, 'Adicionar *Features*' aparece na sequência com 37 ocorrências, representando as refatorações que ocorreram quando novas funcionalidades foram inseridas no *software*. Isso pode ser ilustrado pela mensagem: "*Novo tipo de recurso 'link com função': Pode definir o slot agora, selecionando uma anotação existente ou selecionando um intervalo de texto*".

A análise identificou que refatorações também ocorreram durante 'Correção de *bug*' (21 ocorrências) e em manutenções que buscavam 'Melhorar o Desempenho' do sistema (13 ocorrências). Isso pode ser visto, respectivamente, nas seguintes mensagens: *"Bug corrigido: os recursos do rótulo nunca foram definidos no adaptador diff"* e *"Melhore o tempo de carregamento do CAS evitando chamadas para getJCas() depois que o CAS é criado"*. Por fim, a motivação 'Remoção de *Features*' aparece com 12 ocorrências. A Tabela 3.1 também apresenta o grupo Sem Categoria que representa as refatorações onde não foi possível identificar as motivações através das mensagens de *commit*, seja por falta

de informações ou por pobreza de conteúdo.

Em resumo, em aproximadamente 35% das refatorações, a mensagem de *commit* apresentou intenção de melhorar a qualidade do código (categoria 'Sim'), enquanto que 39% delas foram realizadas quando a intenção principal não era melhorar a qualidade do código (categoria 'Não'). Por fim, não foi possível identificar o objetivo de 26% das refatorações.

3.2.3 Relação entre Motivações e LOC

Esta seção visa responder QP3: Existe relação entre as motivações para refatoração extraídas das mensagens de *commits* e número de linhas de código do método refatorado? Para isso, foram definidos intervalos de valores de LOC para comparar os percentuais de refatorações de *extract method* que ocorreram com intenção ou sem intenção de melhorar a qualidade do código. A Figura 3.3 apresenta um gráfico com cinco intervalos de LOC e o percentual de ocorrência de refatorações de cada categoria em cada intervalo. Os intervalos de LOC do gráfico foram definidos pelo próprio autor desse trabalho, por considerar, com base em sua experiência como programador, que tais intervalos representam valores que diferenciam bem os tamanhos dos métodos entre si.

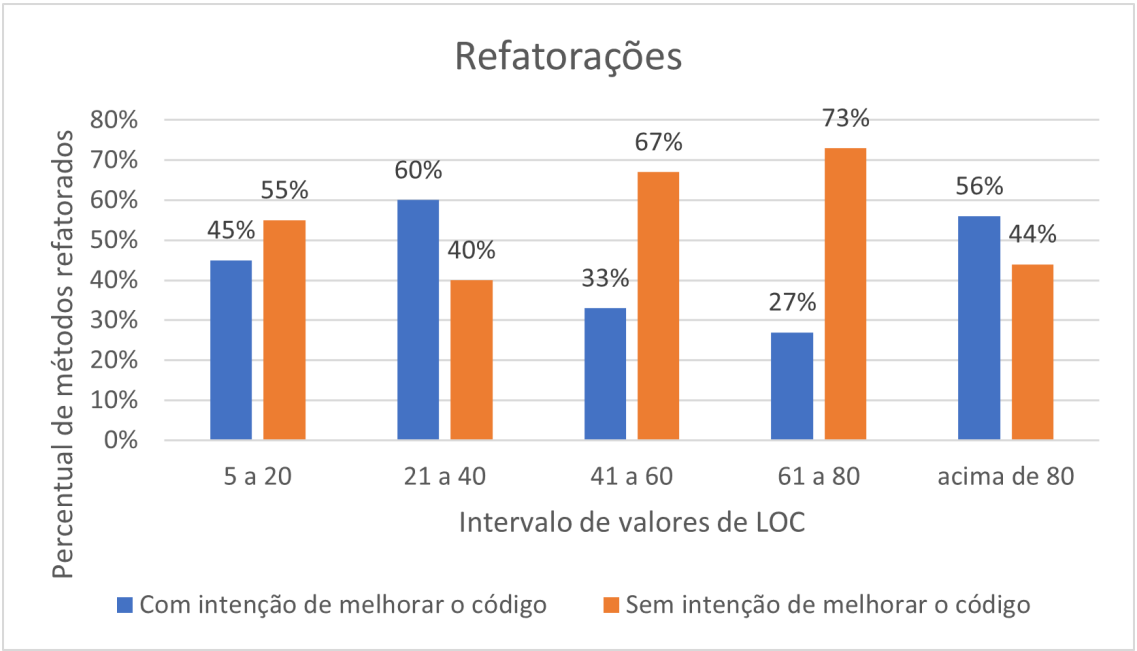

Figura 3.3 Quantidade de métodos refatorados em intervalos de valores de LOC

O gráfico mostra que, no primeiro intervalo (5 a 20), onde os valores de LOC são menores, 45% das refatorações buscaram melhorar a qualidade do código, enquanto que 55% não tinham essa intenção. Já o segundo intervalo, de métodos com valores de LOC entre 21 a 40, apresentou um percentual maior de refatorações com intenção de melhorar a qualidade: 60% contra 40%. Os intervalos com LOC de 41 a 60 e com LOC de 61 a 80 apresentaram baixo o número de refatorações classificadas como 'Sim', pois as refatorações que não tinham intenção de melhorar a qualidade do código ocorreram em

um percentual de 67% e 73%, respectivamente. Finalmente, o último intervalo (acima de 80) apresentou percentual de 56% para as refatorações classificadas como 'Sim', e 44% para aquelas classificadas como 'Não'.

Esses resultados possibilitaram observar que, mesmo em métodos com valores menores de LOC, ocorreram muitas refatorações com intenção de melhorar o código. Por outro lado, também foi identificado que, em métodos com altos valores de LOC, a maioria das refatorações que ocorreram não visavam melhorar a qualidade do código. Isso mostra um ponto curioso, pois esperava-se que, devido ao alto valor de LOC, as refatorações que seriam aplicadas nesses métodos seriam para buscar melhorá-los. Entretanto, foi visto que as refatorações com essa motivação ocorreram com maior frequência em métodos com tamanho na faixa de 21 a 40 linhas de código do que nos intervalos com valores maiores.

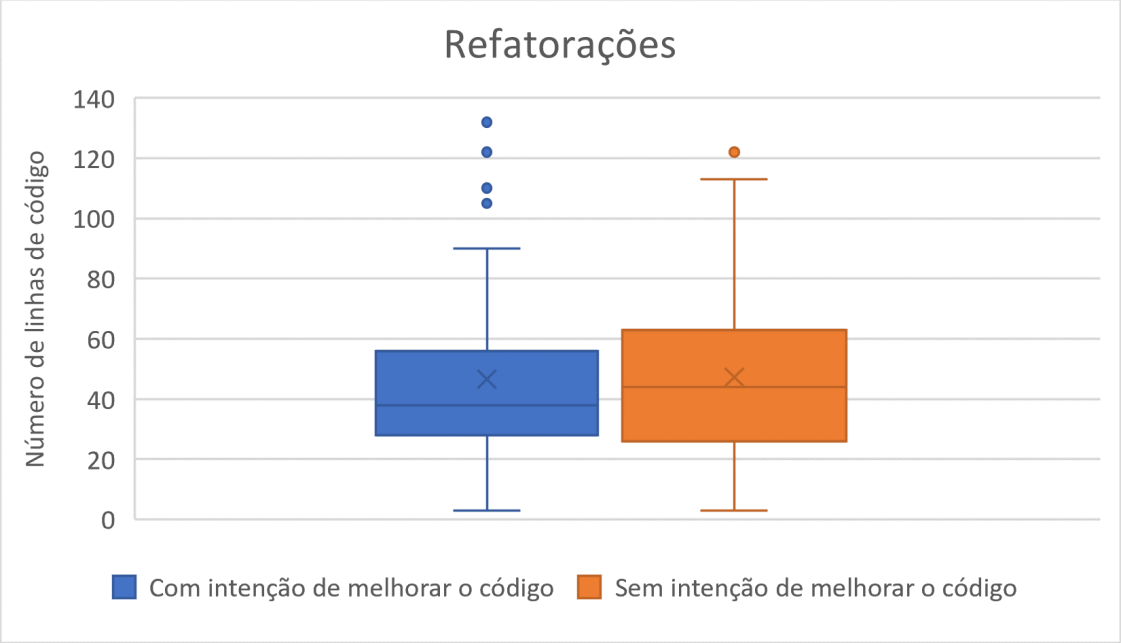

Figura 3.4 Distribuição de LOC entre as refatorações com e sem intenção de melhorar o código

Além da análise com base na Figura 3.3, analisou-se também se existia diferença significativa em valores de LOC dos métodos que sofreram refatoração cuja mensagem de *commit* apresentava intenção explícita de melhorar a qualidade do código em comparação com os métodos onde a mensagem de *commit* não indicava essa intenção. Assim, foi construído um *boxplot*, visto na Figura 3.4, comparando os valores de LOC dos métodos de ambas categorias. O gráfico mostra que não existe diferença significativa de LOC dos métodos analisados. Apesar das refatorações com intenção de melhorar a qualidade do código ocorrerem mais em métodos com valores de LOC menores, essa diferença não é estatisticamente significativa.

3.3 AMEAÇAS À VALIDADE

Esta seção apresenta as ameaças à validade do estudo e ações tomadas para minimizá-las.

Validade de Construto: Uma ameaça para esse estudo foi encontrar sistemas cujas mensagens de *commits* tivessem a descrição das atividades realizadas pelos colaboradores durante a manutenção. Em muitos sistemas, as mensagens de *commit* não apresentam conteúdo ou são pobres de informações, o que dificultaria nossa análise. Desse modo, avaliamos alguns sistemas antes de selecionarmos um que apresentava mensagens de *commit* que informavam relativamente bem o que foi feito naquele *commit*.

Outra ameaça para o estudo é que as mensagens de *commit* foram avaliadas e classificadas por apeanas uma pessoa, sem revisão de outros. Isso diminui a confiança de que as mensagens realmente tinham a intenção com as quais elas foram classificadas. No próximo estudo, planeja-se que cada mensagem seja analisada por, pelo menos, duas pessoas.

Validade de Conclusão: Aqui, a principal ameaça é a pequena quantidade de sistemas e *commits* considerados no estudo. Porém, esse é um estudo preliminar que servirá de base para outro estudo de larga escala, considerando diversos sistemas de *software*.

Além disso, este estudo não utilizou testes estatísticos para verificar se existia diferença significativa em relação ao número de linhas de código entre os grupos de refatorações com intenção e sem intenção de melhorar a qualidade do código. Embora gráficos tenham sido utilizados para responder essa questão, a falta de um teste estático limita a validação dos resultados obtidos. Contudo, no estudo final, serão aplicados testes estatísticos de modo a garantir inferências sólidas e confiáveis sobre os resultados obtidos.

Validade Externa: Os resultados encontrados neste estudo não devem ser generalizados para outros sistemas. Para isso, seriam necessários mais estudos com uma quantidade maior de sistemas, considerando diferentes contextos e domínios.

3.4 CONCLUSÃO

Neste trabalho foi realizado um estudo experimental para analisar as refatorações de *extract method*, buscando compreender suas motivações e a relação dessas motivações com a quantidade de linhas de código dos métodos refatorados. Os resultados preliminares indicaram 11 tipos de motivações para a realização de *extract method*. Dentre os tipos de motivação, foi visto que seis deles visavam melhorar a qualidade do código. Dentre eles, os mais frequentes foram: remover código duplicado, realizar limpeza, organizar código e melhorar a legibilidade. Além disso, foi visto que muitas refatorações de *extract method* ocorreram durante atividades como inserção de uma nova funcionalidade, atualização de uma funcionalidade e até mesmo em manutenções para melhorar o desempenho do sistema ou para corrigir *bugs*.

O estudo também permitiu observar que, das refatorações que ocorreram, apenas 35% visavam melhorar a qualidade do código, enquanto que 39% apresentavam outras motivações. As mensagens de *commit* das 26% restantes não permitiram identificar a motivação. Por fim, os resultados apontaram que as refatorações ocorreram com maior frequência em métodos com valores de LOC na faixa de 29 a 42, e que as refatorações realizadas com intenção de melhorar a qualidade do código foram mais frequentes em métodos com números de linha de código entre 21 a 40 e menos frequente entre métodos

com valores de 61 a 80.

Capítulo

4

ESTUDO FINAL

Este capítulo descreve o estudo experimental final, apresentando os procedimentos adotados para execução do estudo e os resultados obtidos. Este segundo estudo teve como objetivo, assim como no estudo preliminar, investigar as motivações para realizar as operações de *extract method* e avaliar se o tamanho do método tem relação com a motivação do desenvolvedor para usar a refatoração de *extract method*. No entanto, para este segundo estudo, procuramos aprofundar as investigações através da inclusão de mais sistemas no estudo e alguns ajustes no processo de análise dos dados.

4.1 PROJETO DO ESTUDO

Com base nos resultados do estudo preliminar, este estudo experimental final foi conduzido com a intenção de verificar os resultados considerando uma quantidade maior de sistemas. Este estudo segue parâmetros similares ao do estudo preliminar, apenas considerando um número maior de repositórios de *software*. Os resultados do estudo preliminar não foram incorporados neste estudo pois ambos foram executados em momentos diferentes. Esta seção descreve as configurações do novo estudo realizado.

4.1.1 Seleção dos Repositórios de Software

A princípio, foi realizado uma busca para identificar repositórios de *software* alvos a serem analisados. Sendo assim, buscamos por sistemas orientado a objetos, que atendessem aos seguintes critérios:

- Ser desenvolvido na linguagem Java. Muitos sistemas importantes são desenvolvidos em Java, com isso, diversos estudos similares encontrados na literatura foram baseados em sistemas escrito nessa linguagem. Além disso, as principais ferramentas para identificação de refatorações são específicas para linguagem Java.
- Utilizar sistema de controle de versão. Com o controle de versão é possível analisar a evolução do *software* ao longo do tempo, obtendo todas as operações de refatorações de *extract method* que ocorreram durante a evolução do sistema.

- Apresentar mensagens de *commit* descrevendo as alterações. Como o estudo se baseia em mensagens de *commit*, buscamos por sistemas onde apresentassem mensagens que descrevessem relativamente bem o que foi feito em cada *commit*.

Para selecionar os sistemas que atendessem os critérios listados acima, foi utilizado o GitHub como plataforma de hospedagem de código-fonte. Com isso, foi utilizado a seguinte *string* de busca nas opções de busca avançada do GitHub: "comments:200..500 language:Java". Basicamente, essa foi a forma encontrada para localizar sistemas escritos em Java e que apresentassem uma boa quantidade de mensagens de *commit*.

O resultado dessa busca apresenta *issues* que tiveram um número significativo de comentários (entre 200 à 500). No GitHub, uma *issue* é uma funcionalidade que possibilita aos usuários reportarem problemas ou *bugs* encontrados no sistema. Uma *issue* também é utilizada para que os colaboradores discutam aprimoramentos e atribuam tarefas. O propósito de uma *issue* é facilitar a comunicação, colaboração e o acompanhamento do progresso no desenvolvimento do projeto. De modo geral, uma *issue* começa com um *post* inicial que descreve o problema, tarefa ou melhoria, e, posteriormente, os usuários podem adicionar comentários para discutir detalhes e propor soluções. Possivelmente, o fato de uma *issue* apresentar muitos comentários pode ser um indicativo que pode haver um hábito dos desenvolvedores de comentarem as ações realizadas nos *commits* através das mensagens de *commit*.

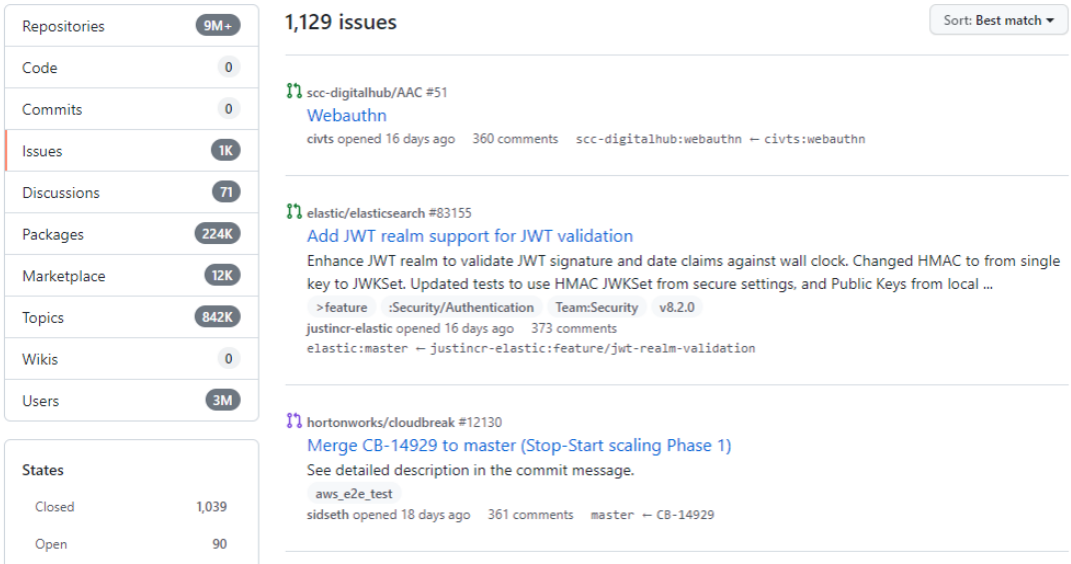

Figura 4.1 Issue options

Sendo assim, essa busca resultou em 1.129 *issues*, visto na Figura 4.1. A partir disso, foram abertos os sistemas das *issues* localizadas, e foi realizada, pelo autor desta pesquisa, uma verificação das três últimas listas de *commits* de cada sistema, a fim de validar se, de fato, tais *commits* apresentavam mensagens que descrevessem as modificações ocorridas, a Figura 4.2 apresenta um exemplo de uma lista de *commit*. A partir da lista de *commit*, era verificado o conteúdo de cada mensagem. A Figura 4.3 exemplifica

uma mensagem de *commit*. Caso os *commits* analisados apresentassem mensagens que descrevessem as modificações realizadas, esse sistema era selecionado para fazer parte do estudo final. No entanto, se as mensagens de *commits* não apresentassem conteúdo ou tivessem pouquíssimas informações, esse sistema seria rejeitado para a pesquisa, pois seria inviável identificar as motivações através da análise das mensagens.

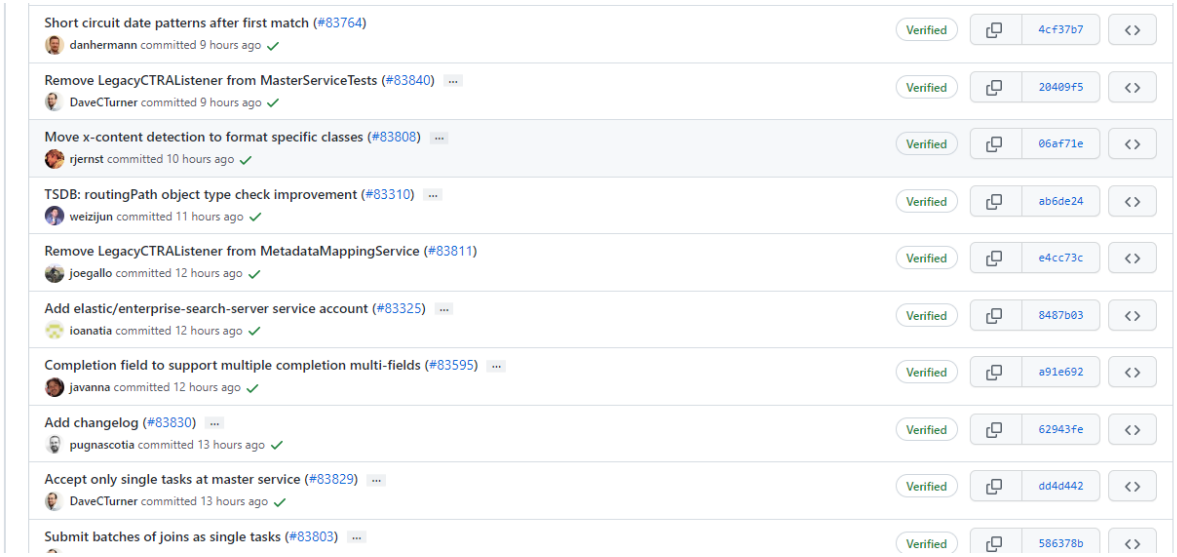

Figura 4.2 Lista de *commit*

Esse procedimento foi aplicado aos 570 primeiros *issues* dos 1.129 identificados. Esse número foi suficiente para localizar uma quantidade interessante de repositórios para compor o estudo final. Dessa forma, foram encontrados 69 repositórios de *softwares* que atenderam aos critérios estabelecidos para o estudo final.

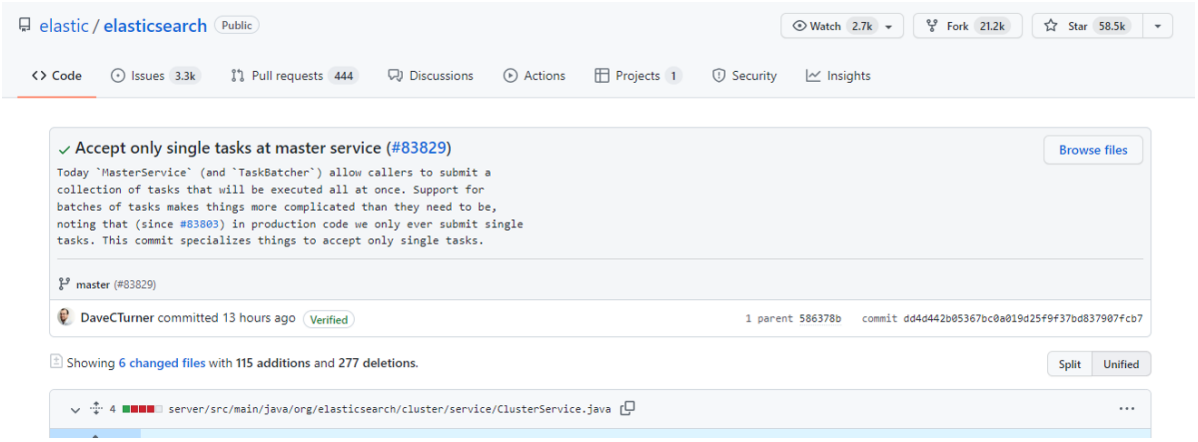

Figura 4.3 Mensagem de *commit*

4.1.2 Procedimentos do Estudo

Do mesmo modo do estudo preliminar, este estudo foi dividido em quatro etapas distintas. Podemos ver um resumo das etapas que foram realizadas na Figura 4.4. Depois de selecionar os repositórios de *software* na primeira etapa do estudo, a segunda etapa foi iniciada com objetivo de coletar os dados necessários para a análise. Este estudo final utilizou a mesma ferramenta de mineração de refatorações utilizada no estudo preliminar. Sendo assim, durante o período entre abril e julho de 2022, a *RefactoringMiner* 2.0 foi utilizada no histórico de *commits* dos *softwares* selecionados para esse estudo. O intervalo entre as versões dos *softwares* que foram consideras para a mineração das refatorações foram baseadas nas *tags* dos sistemas disponíveis no GitHub. Em um sistema de controle de versões, as versões de um determinado *software* são conhecidas como *tags*. As *tags* representam cópias de um estado do sistema, em geral, um estado estável. Sendo assim, a mineração das refatorações foi realizada a partir da primeira *tag* que representa uma versão estável do *software*, até a última *tag* do repositório, evitando assim as versões iniciais dos sistemas, onde podem apresentar um estágio muito instável do sistema. A mineração das refatorações foi executada em um notebook com processador core i7, 8 GB de memória RAM, disco rígido de 1 *terabyte* e utilizando o sistema operacional *Windows* 10.

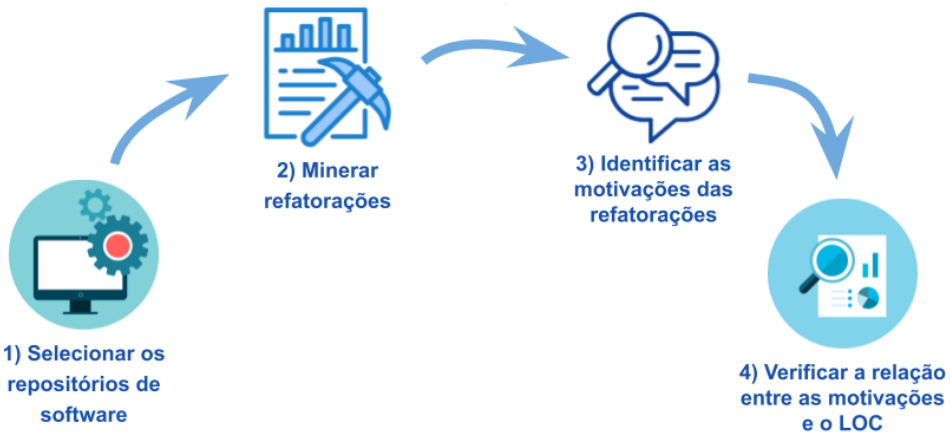

Figura 4.4 Etapas do Estudo Final

Dos 69 repositórios selecionados, foi possível coletar os dados referentes as refatorações de *extract method* de apenas 34 sistemas. Isso se deve ao fato de que em muitos sistemas não foi possível executar a mineração das refatorações devido ao tamanho do sistema. Dado os recursos computacionais disponíveis para o estudo, tais recursos não foram suficientes para minerar parte dos sistemas selecionados inicialmente. Apesar da ferramenta passar diversos dias minerando as refatorações nos repositórios de *software*, em alguns casos a ferramenta apresentava falha ao longo do processo de mineração e não armazenava os dados que foram minerados até o momento da falha. A falha apresentada era referente

a quantidade de memória disponível. A descrição detalhada dos 34 *softwares* selecionados, contendo o nome, intervalo de tempo entre o lançamento da primeira e última versão, link e informações estruturais de tamanho em termos de numéro de classes, número de métodos e número de linhas de código estão na Tabela 4.1. De modo geral, considerando o tamanho dos sistemas em relação a quantidade de classes, métodos e linhas de código, o maior sistema analisado foi o Openj9, e o menor foi Claw-compiler.

Tabela 4.1 Repositórios minerados com RefactoringMiner 2.0

Software	Período	Classes	Métodos	LOC	Link
Anki-Android	11 anos	593	6941	69620	https://github.com/ankidroid/Anki-Android
AntennaPod	9 anos	490	4130	41696	https://github.com/AntennaPod/AntennaPod
Appinventor-sources	9 anos	809	9383	90379	https://github.com/mit-cml/appinventor-sources
Armeria	6 anos	1501	14906	116994	https://github.com/line/armeria
Biojava	11 anos	1108	11383	116135	https://github.com/biojava/biojava
Blueocean-plugin	3 anos	545	3986	32628	https://github.com/Jankyboy/blueocean-plugin
Claw-compiler	5 anos	252	3209	28388	https://github.com/claw-project/claw-compiler
Core-java-spring	1 ano	634	7932	66169	https://github.com/eclipse-arrowhead/core-java-spring
Drill	8 anos	2348	35948	266538	https://github.com/apache/drill
Essentials	10 anos	442	4071	37366	https://github.com/EssentialsX/Essentials
ExoPlayer	8 anos	1175	17833	173019	https://github.com/google/ExoPlayer
Fhir	2 anos	2352	50358	428462	https://github.com/IBM/FHIR
Flow-components	1 ano	1592	17334	101000	https://github.com/vaadin/flow-components
Gadgetbridge	7 anos	958	9251	87984	https://github.com/Freeyourgadget/Gadgetbridge
Graphhopper	7 anos	771	7378	69046	https://github.com/graphhopper/graphhopper
Helix	8 anos	1243	10602	127098	https://github.com/apache/helix
HubTurbo	2 anos	1243	10602	127098	https://github.com/HubTurbo/HubTurbo
Hudi	5 anos	2348	35948	266538	https://github.com/apache/hudi
Identity-inbound	6 anos	665	6790	68284	https://github.com/wso2-extensions/identity-inbound-auth-oauth
Kafka	8 anos	2823	33668	301207	https://github.com/apache/kafka
Mindustry	3 anos	592	6057	75681	https://github.com/Anuken/Mindustry
Netty	11 anos	1645	21605	187434	https://github.com/netty/netty
NewPipe	6 anos	278	3318	32703	https://github.com/TeamNewPipe/NewPipe
Open-smart-grid-platform	3 anos	2937	19449	138641	https://github.com/OSGP/open-smart-grid-platform
Openj9	3 anos	4297	55763	496293	https://github.com/eclipse-openj9/openj9
Pdfbox	4 anos	1224	11978	111625	https://github.com/apache/pdfbox
PojavLauncher	1 ano	897	11360	103183	https://github.com/PojavLauncherTeam/PojavLauncher
Pravega	2 anos	1799	18585	186746	https://github.com/pravega/pravega
Runelite	5 anos	1273	6393	176479	https://github.com/runelite/runelite
Sdl java suite	6 anos	1005	8777	87518	https://github.com/smartdevicelink/sdl$_{j}ava_{s}uite$
Signal-Android	5 anos	2352	24604	190384	https://github.com/signalapp/Signal-Android
Strimzi-kafka-operator	4 anos	865	10374	95542	https://github.com/strimzi/strimzi-kafka-operator
Weblogic-kubernetes-operator	4 anos	688	12679	81669	https://github.com/oracle/weblogic-kubernetes-operator
xDrip	5 anos	1047	9661	126817	https://github.com/NightscoutFoundation/xDrip

Com isso, foram identificadas as refatorações de *extract method* que ocorreram ao longo dos *commits* nos 34 sistemas. Ao todo, a ferramenta detectou 7.138 refatorações de *extract method* que ocorreram durante 2787 *commits* existentes nos sistemas. Assim, foi iniciado a terceira etapa do estudo, onde buscamos identificar as motivações por trás das refatorações coletadas. Desse modo, todas as mensagens de *commit* foram analisadas qualitativamente, de modo a extrair as motivações por trás dessas refatorações. Assim como no estudo preliminar, a análise temática foi utilizada como metodologia (CRUZES; DYBA, 2011). Essa análise foi desenvolvida da seguinte forma. Primeiramente, foi realizada uma leitura inicial de todas as mensagens de *commit*. Depois, foram atribuídos códigos para cada mensagem. Em seguida, definiu-se temas com base nos códigos gerados anteriormente. Então, uma revisão dos temas foi realizada buscando juntar temas semelhantes e, finalmente, foram definidos os temas finais.

Diferente do estudo preliminar, de modo a aumentar a confiabilidade dos resultados, neste estudo final as etapas citadas acima foram realizadas de forma independente por dois

desenvolvedores. Os desenvolvedores foram o autor deste trabalho e um profissional com formação em engenharia da computação. Durante a realização deste estudo, o primeiro desenvolvedor possuía dois anos de vivência na área de pesquisa em refatoração, enquanto o segundo desenvolvedor tinha mais de dez anos de experiência em desenvolvimento de *software*.

Depois da análise ser concluída por ambos os desenvolvedores, foram realizadas reuniões para definir os temas finais. Nestas reuniões, os desenvolvedores verificavam as motivações atribuídas para as mensagens de *commit*. Se os dois atribuíram motivações equivalentes para uma mensagem de *commit*, logo esta motivação se tornaria definitiva. No entanto, quando os desenvolvedores atribuíam motivações diferentes, iniciavam uma discussão para definir qual seria a motivação mais apropriada para tal mensagem. Em 86% dos casos, os desenvolvedores atribuíram os mesmos temas. Das mensagens com temas divergentes, em 13% das ocasiões, um desenvolvedor concordou com a motivação apresentada pelo outro desenvolvedor. No 1% restante, a motivação final acabou sendo diferente em relação às sugestões iniciais dos dois desenvolvedores.

Para exemplificar, umas das mensagens de *commit* analisadas tinha a descrição: "*Refatorar para eliminar a duplicação de código*". Com base nessa mensagem, os dois pesquisadores classificaram como 'Remover duplicação'. Portanto, essa foi a motivação final para essa mensagem.

No entanto, na mensagem de *commit* "*PebbleMovement: Melhore a eficiência do armazenamento de dados*", o primeiro pesquisador classificou a motivação como 'Atualizar features', por entender que se tratava de uma alteração de funcionalidade. Já o segundo pesquisador classificou como 'Melhorar o desempenho', pois uma melhor eficiência no armazenamento tem como impacto a economia de recursos de memória e redução de tempo de acesso aos dados. Após os pesquisadores discutirem, o primeiro pesquisador aceitou o tema proposto pelo outro pesquisador. Os dois concordaram que a motivação 'Melhorar o desempenho' era mais adequada para essa refatoração.

Por fim, em algumas situações as motivações atribuídas pelos dois pesquisadores eram diferentes. Por exemplo, na mensagem de *commit* "*DownloadManagerTest: Remover DownloadRunner. Seu único propósito real é encapsular um ID de download, mas na verdade não salva nenhum código, e argumentavelmente torna mais complicado ao ter múltiplas listas de instâncias de Downloader, indexadas por chave, e também por ser outra classe (não documentada) para entender.*". Inicialmente, o primeiro pesquisador classificou como 'Remover *features*', porém, o segundo pesquisador classificou como 'Simplificar código', como as motivações eram distintas, os pesquisadores discutiram e chegaram a um consenso de que a motivação ideal para essa mensagem seria 'Limpeza de código', pois o desenvolvedor estava removendo uma classe não muito necessária, pois o '*DownloadRunner*' não adiciona benefícios reais ao projeto. Nessa ocasião o tema final foi diferente do que os pesquisadores sugeriram inicialmente.

A partir da análise qualitativa, foi possível identificar as motivações por trás das operações de *extract method*. Essas motivações foram classificadas em duas categorias com base na sua intenção. A primeira categoria representavam as motivações com intenção de melhorar a qualidade do código. Essa categoria foi representada com a palavra 'Sim'. Já a segunda categoria representava as motivações que não tinha intenção de melhorar a

qualidade do código. A palavra 'Não' foi utilizada para representar essa categoria.

Finalmente, uma análise quantitativa foi realizada na quarta etapa para verificar a métrica LOC dos métodos refatorados, buscando analisar a relação entre o tamanho do método e a motivação da refatoração. Assim, foi verificado a distribuição dos valores de LOC dos métodos refatorados, objetivando identificar em quais faixas de valores de LOC as refatorações de *extract method* são mais frequentes. Além disso, foi analisada a relação entre a métrica de LOC com as categorias encontradas nas mensagens analisadas, buscando investigar em quais níveis de LOC as refatorações com intenção de melhorar a qualidade do código ocorrem com maior frequência. Todos os dados considerados no estudo final estão disponíveis publicamente[1].

4.2 RESULTADOS

Nesta seção, compartilhamos os resultados obtidos por meio do estudo experimental final realizado, com o objetivo de responder às questões de pesquisa formuladas. Os resultados encontrados são apresentados de forma detalhada, de modo a permitir uma compreensão aprofundada dos padrões e tendências observadas no estudo realizado.

4.2.1 Relação entre Refatorações e LOC

De modo a responder a primeira questão de pesquisa: "QP1: As refatorações de *extract method* ocorrem com maior frequência em métodos com valores de LOC alto?", analisamos os valores de LOC de todos os métodos refatorados através de um gráfico de distribuição de frequência, conforme a Figura 4.5. Assim como no estudo preliminar, os valores dos intervalos do eixo x do gráfico foram gerados automaticamente através da ferramenta *Google Sheets*. Deste modo, com base no gráfico da Figura 4.5, é visto que as refatorações de *extract method* são mais frequentes em métodos na faixa entre 19 e 34 números de linhas de código.

Além disso, os resultados apontaram que um número considerável de refatorações também ocorreram em métodos com valores de LOC na faixa entre 4 e 19, e entre 34 e 49. Novamente, esses dados confirmam que as refatorações de *extract method* não ocorrem apenas em métodos longos, ou seja, métodos com valores de LOC alto, que representam problema de baixa qualidade de código. As refatorações também ocorrem com bastante frequência em métodos com baixos valores de LOC. Contudo, as refatorações podem ocorrer por diversos motivos, nem sempre o objetivo é reduzir o tamanho do método para melhorar a qualidade daquele código, em muitos casos, fatores como mudança de requisitos podem estar atrelado a refatoração, como foi observado anteriormente no estudo preliminar.

4.2.2 Motivações para Realizar Extract Method

Esta seção tem como objetivo responder a QP2: O que os desenvolvedores reportam como motivação para realizar o *extract method*? Após a análise das mensagens de *commit* de

[1]https://zenodo.org/records/10699514

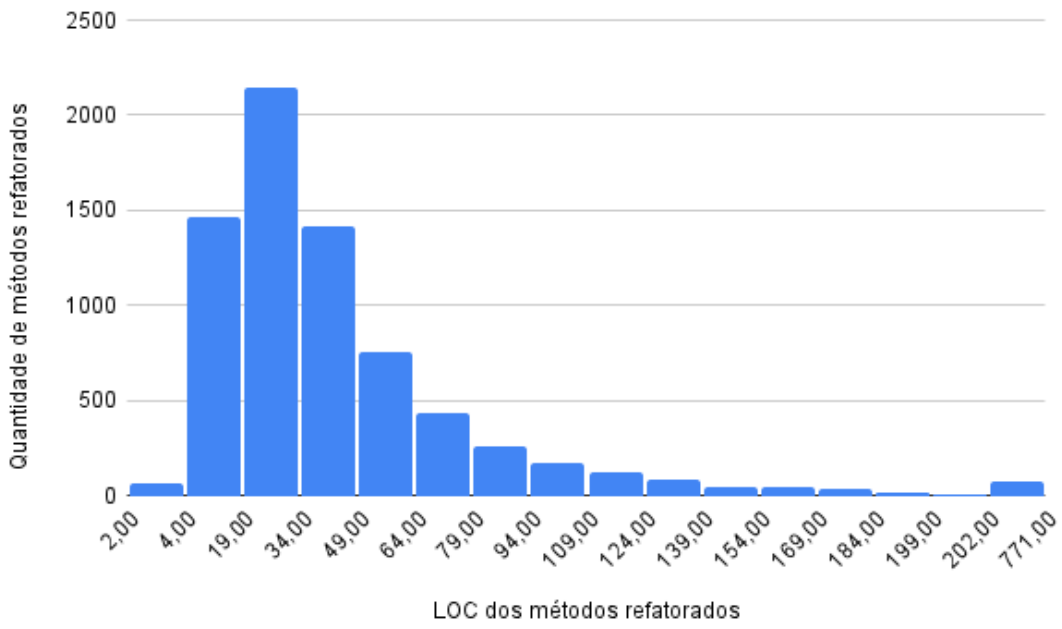

Figura 4.5 Distribuição de frequências dos valores de LOC dos métodos refatorados

todas refatorações coletadas no estudo, foram identificadas as motivações que levaram a aplicação do *extract method*. Ao todo, o estudo final identificou dezesseis motivações para o uso do *extract method*. A Tabela 4.2 apresenta as motivações, a descrição de cada motivação e a quantidade de ocorrências. Além disso, também é apresentado as categorias das motivações, sendo a palavra 'Sim' para representar as motivações que apresentavam intenção de melhorar a qualidade do código e 'Não' para indicar as refatorações que não buscavam melhorar a qualidade do código.

Das dezesseis motivações identificadas, dez foram classificadas como refatorações que buscavam melhorar a qualidade do código. Essa categoria representa 16% das refatorações analisadas no estudo. Dentro deste grupo, a motivação 'Limpeza de código' aparece com maior número de ocorrências. Foram 348 operações com essa motivação, retratando um percentual de 4,88% do total de refatorações. Essa motivação representa as refatorações que ocorreram buscando remover trechos de códigos desnecessários ou códigos que se encontravam legados. Algumas mensagens como: "*Código não utilizado removido*" e "*Limpeza geral do código*", exemplifica essa motivação.

A segunda motivação mais frequente é 'Organizar o código' com 168 (2,35%) ocorrências. Essa motivação busca melhorar a organização da estrutura do código, como mover métodos ou trechos de código para locais mais adequados no sistema. Mensagens como "*Mova a lógica 'Delete Remove from Queue' para DBWriter*" e "*Refatorar o carregamento de coleção assíncrona para uma única classe de carregador*" demonstram essa motivação.

Em seguida, as refatorações para 'Remover duplicação' aparecem em terceiro lugar (163 ocorrências) dentre aquelas com intenção de melhorar a qualidade do código. Essa motivação representa refatorações para excluir trechos de código redundantes que se re-

Tabela 4.2 Motivações para utilização do *extract method*

Categoria	Motivação	Descrição	Ocorrência	Total
Sim	Limpeza de código	Remoção de trechos de código legado ou desnecessário.	348 (4,88%)	16%
Sim	Organizar código	Modificações para organizar a estrutura do sistema, como mover trechos de um método para outro método ou classe mais adequada.	168 (2,35%)	
Sim	Remover duplicação	Remover trechos de código que se repetem em vários locais no sistema.	163 (2,28%)	
Sim	Reduzir o método	Refatorar o método para resolver o problema de *smell Long Method*.	123 (1,72%)	
Sim	Melhorar a legibilidade	Alteração no método para torná-lo mais legível, ou seja mais fácil de ler.	99 (1,39%)	
Sim	Melhorar a testabilidade	Refatorações conduzidas a fim de tornar o método mais fácil de se testar.	98 (1,37%)	
Sim	Simplificar código	Modificações realizadas para simplificar a lógica implementada no método, reduzir a complexidade.	87(1,22%)	
Sim	Reutilizar código	Refatoração realizada para facilitar a reutilização de código.	23 (0,32%)	
Sim	Reduzir acoplamento	Refatoração para reduzir o acoplamento no sistema.	6 (0,08%)	
Sim	Reduzir parâmetros	Refatorar para diminuir a quantidade de parâmetros de um método.	6 (0,08%)	
Não	Atualização de *Features*	Refatorações que ocorreram durante a manutenção das funcionalidades.	3064 (42,93%)	79%
Não	Adicionar *Features*	Refatorações que ocorreram durante a inserção de novas funcionalidades.	1495 (20,94%)	
Não	Correção de *Bug*	Refatorações realizadas durante o processo de correção de *bugs*.	889 (12,45%)	
Não	Remoção de *Features*	Remoção de funcionalidades no sistema.	123 (1,72%)	
Não	Melhorar o Desempenho	Refatorações que ocorreram em ações para melhorar a eficiência do processamento de determinadas funcionalidades.	102 (1,43%)	
Não	Melhorar a Segurança	Refatorações que ocorreram visando melhorar a segurança do sistema.	10 (0,14%)	
	Sem Categoria	Refatorações das quais não foi possível identificar seus objetivos.	334 (4,68%)	5%

Sim: refatorações com intenção de melhorar a qualidade do código

Não: refatorações sem intenção de melhorar a qualidade do código

petem em mais de um local no sistema. Alguns exemplos de trechos de mensagens de *commit* ilustram essa motivação: "*Extraindo código duplicado para um método. [...]*" e "*Código comum refatorado em UnlimitedRawBatchBuffer e SpoolingRawBatchBuffer em BaseRawBatchBuffer*".

Outra motivação que aparece com bastante frequência (123) foi a motivação 'Reduzir o método'. O objetivo dessa motivação é reduzir o tamanho do método para remover o *code smell Long Method*. Métodos menores são mais fáceis de entender e modificar. Isso pode ser visto em mensagens como: "*Dividir o método de 252 linhas em submétodos*" e em "*Extrair métodos em ChooseDataFolderDialog. Isso torna o principal método 'showDialog()' um pouco mais curto, extraindo o comportamento em métodos mais curtos.*".

A quinta motivação a aparecer na tabela foi 'Melhorar a legibilidade', que ocorre em

99 ocasiões. São refatorações com intenção de tornar o código mais fácil de ser lido. Isso pode ser visto nas mensagens: "*Tornar o código mais fácil de ler*" e "*Refatorou algumas assinaturas de métodos internos de alinhamento para melhorar a legibilidade do código*".

Já a motivação 'Melhorar a testabilidade' obteve 98 ocorrências e representa as refatorações que buscavam facilitar a atividade de teste de *software* nos métodos. Essa motivação pode ser observada na mensagem: "*Adicionar método auxiliar para testes mais fáceis*" e, também, na mensagem: "*Refatora o código de classificação externo em pequenos blocos que podem ser testados por unidade*".

Em seguida, 'Simplificar código' é a motivação que aparece com 87 ocorrências, representando refatorações que objetivava reduzir a complexidade do método, simplificando a lógica definida. As mensangens: "*[...] Substitua o item de opção relacionado e simplifique a lógica handleSessionEstablishmentError*" e "*Reduza a complexidade refatorando encryptData em três partes*" exemplifica essa motivação.

A motivação 'Reutilizar código' apareceu em 23 ocorrência, correspondendo apenas ao percentual de 0,32%. Essa motivação tem como propósito melhorar o método para que aquele código possa ser reutilizado no sistema. As mensagens "*Extraia um método para reutilização mais fácil, se necessário*" e "*Generaliza código em Dijkstra bidirecional para poder reutilizá-lo para CH baseado em borda*" exemplificam essa motivação.

Por fim, as motivações 'Reduzir acoplamento' e 'Reduzir parâmetros' surgem como as últimas do grupo das refatorações com o objetivo de aprimorar a qualidade do código. Cada uma delas foi mencionada em 6 ocorrências, o que faz cada uma representar 0,08% do total. A motivação 'Reduzir acoplamento', tem como propósito buscar reduzir o acoplamento no sistema. Segundo Pressman e Maxim (2016), o acoplamento refere-se ao grau em que os componentes de um sistema estão ligados entre si e dependentes uns dos outros. Um acoplamento baixo indica que as partes do sistema estão menos dependentes, o que facilita a manutenção e extensibilidade do sistema. A mensagem "*Adicionadas interfaces entre 'StreamPartitionAssignor' e 'StreamThread' para reduzir o acoplamento.*" e também a mensagem "*Reduzir o acoplamento entre Http2FrameCodec e Http2Multiplex* (9273). Motivação: Http2MultiplexCodec e Http2MultiplexHandler tiveram um acoplamento muito forte com Http2FrameCodec que podemos reduzir facilmente. O objetivo final deve ser não ter nenhum acoplamento*" indicam essa motivação.

A motivação 'Reduzir parâmetros' tem a proposta de diminuir a quantidade de parâmetros de um método e pode ser observada nas mensagens: "*Reescrever a criação do pedido de download (5530). O Android tem um limite no tamanho dos parâmetros do Intent. Ao enfileirar um grande número de itens, ele simplesmente ignorou o argumento e não chamou onNewIntent. Agora carregamos a lista em DownloadService*"e "*Reduza o número de parâmetros necessários para criar um HlsMediaChunk*".

Na segunda categoria, onde as refatorações não apresentavam intenção de melhorar a qualidade do código, foram identificadas seis motivações. Essa categoria contem a maioria das refatorações que foram analisadas neste estudo (79%). Dentre elas, a 'Atualização de *Features*' aparece com maior frequência (3064 vezes), correspondendo a 42,93% do total. Elas são refatorações que ocorreram quando uma funcionalidade do sistema estava sendo modificada. A mensagem a seguir exemplifica esse comportamento: "*Atualização do FusionTableComponent: autenticação de serviço e funções de consulta simples*".

Em segundo lugar, 'Adicionar *Features*' aparece na sequência com 1495 (20,94%) ocorrências, representando as refatorações que ocorreram quando novas funcionalidades foram inseridas no *software*. Isso pode ser ilustrado pelas mensagens: "*Adicione 'Selecionar nenhum', bem como 'Selecionar tudo'*" e "*adicionar novo recurso de deck*".

A análise identificou que refatorações também ocorreram durante 'Correção de *bug*' (889) e em manutenções que buscavam 'Remoção de *Features*', que aparece com 123 ocorrências. Isso pode ser visto, respectivamente, nas seguintes mensagens: *"Correção de bug: A caixa de diálogo sem token era exibida toda vez que um feed era atualizado se o usuário não estivesse logado e tivesse clicado no botão "flattr this"antes"* e *"Remova requestProperties de todos os métodos FHIRRestHelper"*.

Por fim, a motivação 'Melhorar o desempenho' do sistema (102 ocorrências) e a motivação 'Melhorar a segurança' (10 ocorrências) aparecem com menor frequência no grupo das motivações sem intenção de melhorar a qualidade do código. As mensagens "*Melhoria do desempenho - avaliação do FHIRPath do protetor durante a interação de pesquisa*"e "*[SECURITY-1204] Escape XSS antes da saída do valor no gravador json. Para segurança extra, cdata os scripts para que nenhuma tag não autorizada possa sair*"ilustram, respectivamente, tais motivações.

Na última linha da Tabela 4.2, há também o grupo denominado Sem Categoria, com um total de 334 ocorrências, equivalendo aproximadamente a (5%) do total de refatorações. Essas são refatorações em que não foi possível identificar as motivações por meio das mensagens de *commit*, seja pela ausência de informações ou pela escasse de conteúdo relevante.

Para resumir, das refatorações analisadas neste estudo, 16% apresentaram intenção de melhorar a qualidade do código (categoria 'Sim'). Já as refatorações que não apresentaram intenção de melhorar a qualidade do código (categoria 'Não'), apresentaram um índice superior de 79%. Por fim, em 5% das refatorações não foi possível identificar o objetivo, e foram classificadas como 'Sem Categoria'.

4.2.3 Relação entre Motivações e LOC

O objetivo desta seção é responder à QP3: Existe relação entre as motivações para refatoração de *extract method* extraídas das mensagens de *commits* e número de linhas de código do método refatorado? Para atingir este objetivo, foram estabelecidos intervalos de valores de LOC. Em cada intervalo, compararam-se os percentuais de refatorações de *extract method* que ocorreram com intenção ou sem intenção de melhorar a qualidade do código. Para visualizar os resultados, a Figura 4.6 apresenta um gráfico com cinco intervalos de LOC e o percentual de ocorrência de refatorações de cada categoria em cada intervalo. Os intervalos de LOC apresentados no gráfico foram cuidadosamente determinados pelo autor deste estudo, levando em consideração sua experiência como programador. Esses intervalos foram escolhidos de forma a destacar claramente as diferenças de tamanho entre os métodos.

De acordo com o gráfico apresentado, o primeiro intervalo (de 2 a 20) apresentou uma quantidade bem maior de refatorações sem intenção de melhorar a qualidade do código, com um percentual de 86%, enquanto que apenas 14% dessas refatorações tiveram essa

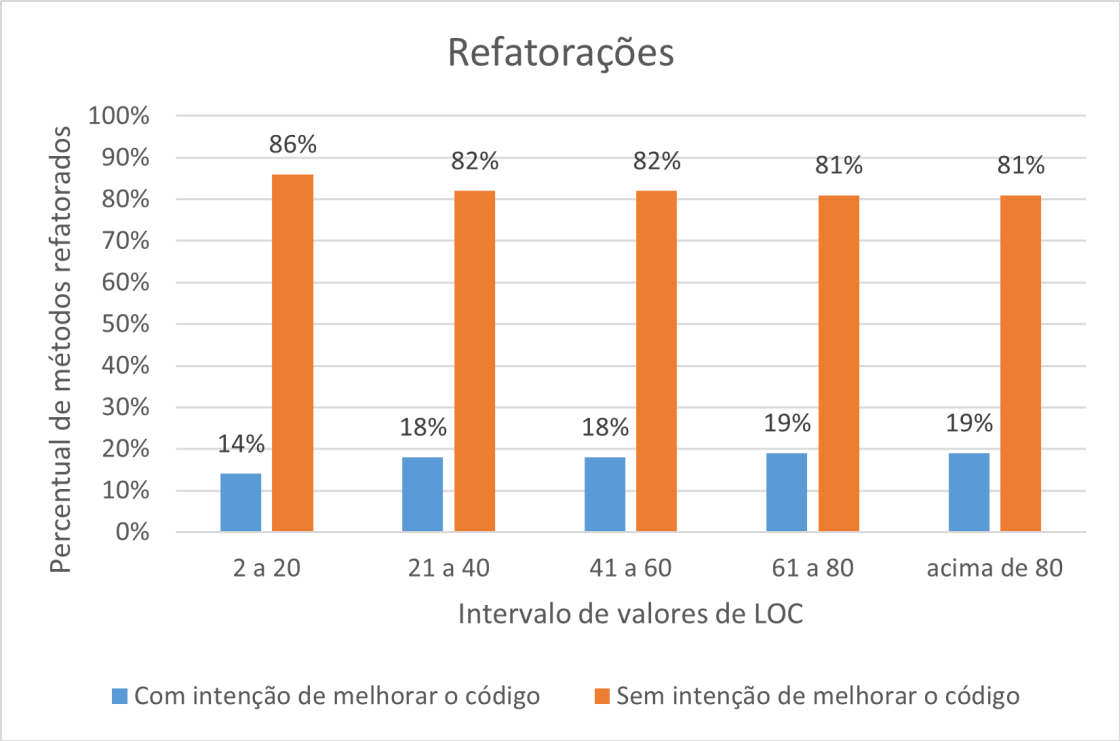

Figura 4.6 Quantidade de métodos refatorados em intervalos de valores de LOC

intenção. Já no segundo intervalo (de 21 a 40), o percentual de refatorações com intenção de melhorar a qualidade do código aumentou um pouco, chegando a 18%, enquanto que 82% das refatorações não tinham essa intenção. Nos intervalos de 41 a 60 e 61 a 80, o percentual de refatorações com intenção de melhorar a qualidade do código foram 18% e 19%, respectivamente. Por outro lado, nos mesmos intervalos, o percentual de refatorações sem essa intenção foi de 82% e 81%, respectivamente. No último intervalo (acima de 80), o percentual de refatorações com intenção de melhorar a qualidade do código foi de 19%, enquanto que 81% das refatorações não tinham essa intenção.

Os resultados obtidos sugerem que a extensa maioria das refatorações de *extract method* realizadas no estudo não tinham a intenção de melhorar a qualidade do código. Observou-se que as refatorações classificadas com intenção de melhorar o código foram menos frequentes em métodos com valores de LOC entre 2 a 20, e mais frequentes em métodos com valores de LOC na faixa de 61 a 80 e acima de 80. No entanto, o percentual de refatorações com intenção de melhorar a qualidade do código foi baixo em todas as faixas de valores de LOC observadas. Esses dados revelam que as refatorações com essa motivação não ocorrem exclusivamente em métodos com altos valores de LOC, mas sim de forma semelhante em ambas as faixas. No entanto, é um pouco menos frequente em métodos menores, com valores de LOC entre 2 a 20. A menor ocorrência de refatorações em métodos menores pode ser atribuída ao fato de que, devido ao seu tamanho, eles podem não representar um problema de qualidade de código, como o *code smell Long Method*.

Adicionalmente à análise efetuada com base na Figura 4.6, foi conduzida uma análise estatística para verificar se havia diferença significativa nos valores de LOC dos métodos que foram refatorados com intenção explícita de melhorar a qualidade do código em comparação com aqueles onde a mensagem de *commit* não indicava essa intenção. Primeiramente, buscamos verificar se os dados seguem uma distribuição normal. Dado o tamanho considerável da amostra, optamos por utilizar o teste de Kolmogorov-Smirnov. Ao aplicar o teste, o resultado revelou um p-value de 0,001, sugerindo que os dados não obedecem a uma distribuição normal, indicando que são não paramétricos. Diante disso, aplicamos o teste U de Mann-Whitney para verificar se há uma diferença significativa no número de linhas de código entre as duas categorias.

O teste U de Mann-Whitney compara as medianas das duas amostras. Nesse caso, a hipótese nula (H0) afirma que as duas amostras são provenientes da mesma população, enquanto a hipótese alternativa (Ha) considera a possibilidade das duas amostras serem provenientes de populações diferentes. Para a execução deste teste, foi utilizado a ferramenta *Statistical Package for the Social Science* (SPSS *Statistics*) da empresa *International Business Machines* (IBM).

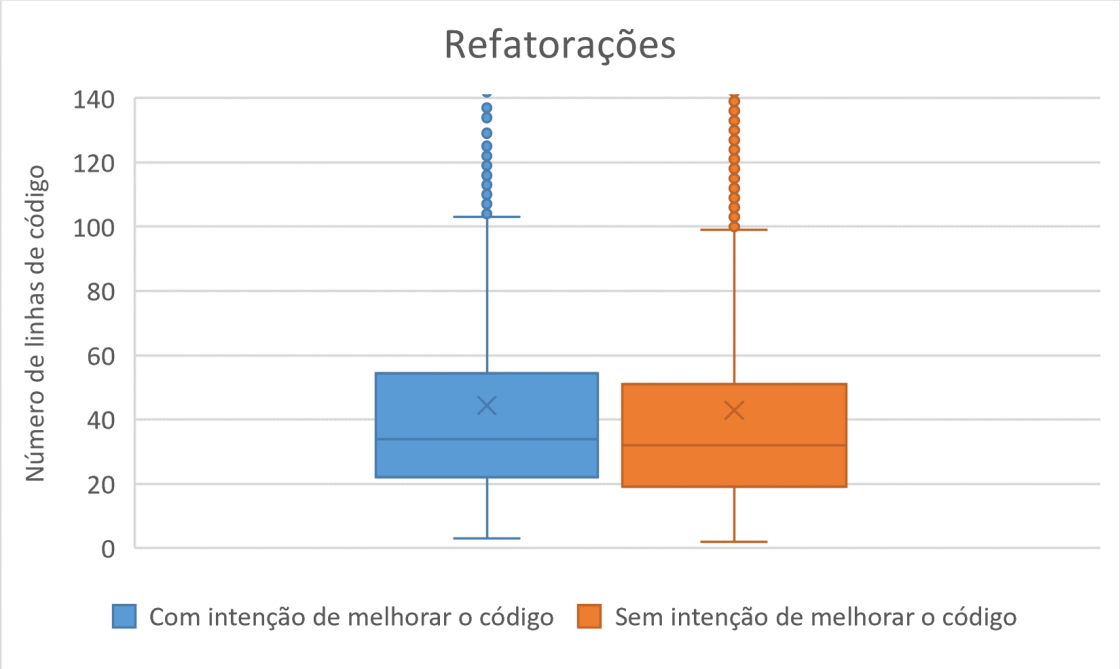

Figura 4.7 Distribuição de LOC entre as refatorações com e sem intenção de melhorar o código

O teste U de Mann-Whitney foi aplicado ao conjunto de dados adotando-se um nível de significância de 0,05 (5%). Os resultados revelaram que houve diferença estatisticamente significativa no número de linhas de código entre os dois grupos (U = 3091093,000, p $<$ 0,05). A mediana de LOC do grupo das refatorações com intenção de melhorar o código foi de 34 (amplitude interquartil = 32,50), enquanto a mediana de LOC do grupo das refatorações onde não tinha intenção de melhorar o código foi de 32 (amplitude interquartil = 32). O valor-p associado ao Mann-Whitney foi calculado como 0,009, o

que é menor do que o nível de significância de 0,05. Sendo assim, com base nos resultados, há evidências suficientes para rejeitar a hipótese nula. Isso indica que foi encontrada uma diferença significativa entre os dos dois conjuntos. Esses resultados sugerem que, no contexto deste estudo, houve diferença significativa entre o número de linhas de código dos métodos que sofreram *extract method* com intenção de melhor a qualidade de seu código e o número de linhas de código dos métodos que sofreram *extract method* sem tal intenção.

Esse resultado pode ser observado no *boxplot*, apresentado na Figura 4.7, que compara os valores de LOC dos métodos de ambas as categorias. As refatorações com intenção de melhorar a qualidade do código ocorreram com mais frequência em métodos com valores de LOC maiores, e essa diferença é estatisticamente significativa.

4.3 DISCUSSÃO DOS RESULTADOS

Os resultados encontrados neste estudo experimental mostraram diferenças expressivas em comparação ao estudo preliminar. Isso reforça a importância de realizar estudos envolvendo quantidades maiores de sistemas para obter resultados mais precisos.

Quanto ao resultado da QP1, o estudo final apresentou um resultado um pouco diferente ao estudo preliminar, nela foi observado que as refatorações de *extract method* são mais frequentes em métodos com valores de LOC entre 19 e 34, no estudo preliminar esse tipo de refatoração era mais comum em métodos com valores entre 29 a 42.

Já na QP2, o estudo final identificou cinco novas motivações para aplicação do *extract method*. Sendo quatro motivações relacionadas ao grupo de refatorações que apresentavam intenção de melhorar a qualidade do código e uma motivação do grupo que não apresentava intenção de melhorar a qualidade do código. As novas motivações encontradas foram: simplificar código, reutilizar código, reduzir acoplamento, reduzir parâmetros e melhorar a segurança. Um ponto interessante é que o estudo final mostra que as refatorações sem intenção de melhorar a qualidade do código ocorrem com frequência muito maior (5655 ocorrências) do que as refatorações com intenção de melhorar a qualidade do código (1149 ocorrências). Isso significa que as operações de *extract method*, em geral, são mais comum durante manutenções que visam alterar o comportamento do sistema.

No estudo preliminar, onde foi considerado apenas um sistema, a diferença entre as refatorações com intenção de melhorar a qualidade do código e as refatorações que não tinham essa intenção não foi tão expressiva quanto no estudo final. Na ocasião, 35% das refatorações tinham a intenção de melhorar a qualidade do código, enquanto 40% não tinham essa intenção. No estudo final, no entanto, os números mudaram, com 16% das refatorações tendo a intenção de melhorar a qualidade do código e 79% não tendo essa intenção.

Durante o estudo final, foi observado que alguns sistemas apresentavam uma quantidade razoável de refatorações *extract method* com intenção de melhorar a qualidade do código, enquanto outros tinham uma quantidade muito pequena de refatorações com essa intenção. A Tabela 4.3 apresenta os repositórios de *software* utilizados no estudo final, juntamente com percentual de refatorações aplicadas com a intenção de melhorar a qualidade do código, o percentual de refatorações aplicadas com outras motivações não

relacionadas à melhoria da qualidade do código e o percentual de refatorações para as quais não foram identificadas motivações, sendo classificadas como 'sem categoria'.

Tabela 4.3 Percentual de refatorações com e sem intenção de melhorar a qualidade do código.

Repositório	Refatorações *Extract Method*		
	Com intenção de melhorar o código	Sem intenção de melhorar o código	Sem categoria
Anki-Android	22%	66%	12%
AntennaPod	14%	77%	9%
Appinventor	7%	86%	7%
Artemis	44%	39%	17%
Biojava	28%	57%	15%
Blueocean	21%	58%	21%
Claw	41%	50%	9%
Core-java-spring	25%	45%	30%
Drill	27%	70%	3%
Essentials	18%	75%	7%
ExoPlayer	11%	89%	0%
Fhir	17%	80%	3%
Flow	11%	89%	0%
Gadgetbridge	25%	74%	1%
Graphhopper	27%	68%	5%
Helix	12%	88%	0%
HubTurbo	14%	79%	7%
Hudi	10%	90%	0%
Identity	14%	82%	4%
Kafka	9%	91%	0%
Mindustry	6%	85%	9%
Netty	14%	85%	1%
NewPipe	20%	79%	1%
Openj9	26%	74%	0%
Open-smart-grid-platform	14%	70%	16%
Pdfbox	39%	59%	2%
PojavLauncher	13%	71%	16%
Pravega	4%	95%	1%
Runelite	8%	89%	2%
Sdl-java-suite	20%	73%	7%
Signal-Android	2%	94%	4%
Strimzi-kafka	18%	79%	3%
Weblogic-kubernetes-operator	35%	48%	17%
xDrip	4%	80%	16%

Como pode ser visto na Tabela 4.3, o sistema Artemis apresentou um percentual de 44% de refatorações com a intenção de melhorar o código, superando o percentual de refatorações sem essa intenção, no qual o mesmo sistemas apresentou o percentual de 39%. Além disso, sistemas como Claw e Pdfbox apresentaram percentuais relativamente altos de refatorações com intenção de melhorar o código, correspondendo a 41% e 39%, respectivamente. Nesses dois sistemas, o percentual de refatoração sem intenção de melhorar o código foram 50% no Claw e 59% no Pdfbox. Por outro lado, os sistemas

Signal-Android, Pravega, e xDrip tiveram uma taxa muito baixa de refatorações com intenção de melhorar o código, com percentuais de apenas 2%, 4% e 4%, respectivamente. Nesses sitemas, as refatorações sem intenção de melhorar o código representaram 94%, 95% e 80%, respectivamente.

Esses dados revelam que, embora as refatorações sem intenção de melhorar a qualidade do código sejam predominantes no estudo final, representando 79% do total de refatorações, esses valores variam bastante de acordo com o repositório. Talvez isso possa estar relacionado com a natureza da equipe. Com base nas motivações identificadas, é possível notar que em alguns projetos a equipe tem uma preocupação maior em melhorar a qualidade do código, enquanto em outros projetos a maioria das refatorações é voltada para a modificação de funcionalidades. Isso sugere que diferentes equipes podem ter prioridades distintas em relação à qualidade do código.

Por fim, os resultados apresentados na QP3 ressaltam que as refatorações sem intenção de melhorar a qualidade do código ocorre com frequência similar (entre 81% e 86%) nos intervalos de valores de LOC. Durante o estudo preliminar os dados indicavam que os intervalos entre 21 a 40 e acima de 80 apresentavam um percentual maior de refarotações com intenção de melhorar a qualidade do código sobre as refatorações sem essa intenção. Contudo, o estudo final apresenta dados completamente diferentes. Apesar das refatorações com intenção de melhorar a qualidade do código serem mais frequente em métodos com valores de LOC entre 61 e 80 e acima de 80, essa quantidade não é expressiva, respondendo apenas a 19% das operações.

Estes estudos experimentais trazem resultados importantes para uma maior compreensão sobre as operações de refatoração de *extract method*. Em relação ao estudo de Liu e Liu (2016), que apresentou quatros motivações para aplicação do *extract method*, esta pesquisa confirma os resultados, pois também identificou as mesmas motivações e ainda mais treze. Neste sentido, a motivação 'decomposição de métodos' é similar a motivação 'reduzir métodos', a 'resolução de clones' se assemelha com 'remover duplicação' e as motivações 'reutilização atual' e 'reutilização futura' equipara-se a 'reutilizar código' no nosso estudo.

Quanto ao estudo de Silva *et al.* (2016), das onze motivações apresentadas no estudo, nossa pesquisa apontou quato motivações semelhantes, como pode ser visto na Tabela 4.3. As outras sete motivações restantes foram diferentes das motivações identificada neste estudo. Sendo assim, esta pesquisa contribuiu para identificação de mais doze novas motivações para aplicação da refatoração de *extract method*.

Tabela 4.4 Motivações semelhantes ao estudo de Silva *et al.* (2016)

Motivações - Silva *et al.* (2016)	**Motivações - Estudo Experimental**
Extração de método reutilizável	Reutilizar código
Decomposição de métodos para melhorar a legibilidade	Melhorar a legibilidade
Remover duplicação	Remover duplicação
Melhorar a testabilidade	Melhorar a testabilidade

4.4 AMEAÇAS À VALIDADE

Nesta seção, serão discutidas as potenciais ameaças à validade do estudo e as medidas adotadas para mitigá-las.

Validade de Construto: Assim como no estudo preliminar, uma questão que representou uma ameaça para este estudo foi a identificação de sistemas nos quais as mensagens de *commit* contivessem descrições detalhadas das atividades realizadas pelos colaboradores durante a manutenção. É bastante recorrente em muitos sistemas a presença de mensagens de *commit* que carecem de conteúdo ou apresentam informações limitadas. Como este estudo analisa mensagens de *commit*, é extremamente importante identificar sistemas que apresentem mensagens de *commit* contendo boas descrições, isto é, mensagens expondo as modificações realizadas. Sendo assim, realizamos uma avaliação criteriosa em diversos sistemas, a fim de selecionar aqueles que apresentavam mensagens de *commit* que forneciam informações relevantes sobre o que foi realizado naquele *commit* específico. No entanto, o próprio processo de seleção dos sistemas também representa uma ameaça, pois presumimos que sistemas com *issues* contendo muitos comentários poderiam indicar que os *commits* apresentavam mensagens descrevendo as modificações realizadas.

Outra ameaça para o estudo é assegurar que os resultados da análise qualitativa das mensagens de *commit* correspondam corretamente às reais motivações do desenvolvedor para refatorar. O estudo considerou uma grande quantidade de *commits* e a análise foi realizada com base nas mensagens associadas a esses *commits*. No entanto, para amenizar a possibilidade de erros, todas as mensagens foram analisadas por dois desenvolvedores.

Validade de Conclusão: Apesar desse estudo ter considerado uma quantidade muito maior de sistema em relação ao estudo preliminar, ainda assim, a quantidade de sistemas utilizados neste estudo representa uma ameaça. Foram analisadas 7.138 refatorações de *extract method* de 34 sistemas de *software*, no entanto, para obter resultados mais precisos é necessária uma quantidade maior de sistemas a serem utilizados no estudo.

Os intervalos de LOC utilizados para verificar a relação entre as motivações para refatorar e o tamanho do método também representam uma ameaça à validade de conclusão. Esses intervalos foram definidos pelo próprio autor com base em sua experiência como desenvolvedor. No entanto, considerando os resultados encontrados e a similaridade dos percentuais das refatorações com e sem intenção de melhorar o código, mesmo se outra forma de definir os intervalos de LOC fosse aplicada, os resultados não sofreriam muitos impactos.

Validade Externa: Os resultados encontrados neste estudo foram baseados na análise de 34 sistemas de código aberto. Sendo assim, não recomendamos que estes resultados sejam generalizados para outros *softwares*, sem considerar a similaridade deles com os sistemas avaliados neste estudo.

Esta pesquisa só analisou sistemas escritos na linguagem Java. Os achados desse estudo não foram comparados ou validados com *softwares* escrito em outras linguagens de programação.

Todas as aplicações utilizada no estudo experimental foram de repositórios de *software open source*, pois esses tem seus artefatos disponíveis facilmente e gratuitamente. Dessa forma, o estudo não avaliou *softwares* comerciais.

Capítulo 5

CONCLUSÃO

Nesta dissertação, foram apresentados dois estudos experimentais para analisar as refatorações de *extract method*, com objetivo de compreender as motivações por trás dessas operações e a relação dessas motivações com a quantidade de número de linhas de código dos métodos refatorados. Os resultados dessa pesquisa convergem, em parte, com estudos de Silva *et al.* (2016) e Liu e Liu (2016), ao apontarem motivações semelhantes para aplicação de refatorações de *extract method*. No entanto, os resultados dessa pesquisa foram além, apresentado novas motivações para o uso do *extract method*.

No total, o estudo final indicou 16 diferentes motivações para a realização de *extract method*. Dentre essas motivações identificadas, foi possível definir dois grupos, o primeiro referente a motivações com intenção de melhorar a qualidade do código e o segundo grupo relativo a motivações que ocorreram com outras intenções não associadas a melhoria da qualidade do código.

O primeiro grupo, das motivações com intenção de melhorar a qualidade do código, contém dez das dezesseis motivações. As motivações foram: limpeza de código, organizar código, remover duplicação, reduzir método, melhorar a legibilidade, melhorar a testabilidade, simplificar código, reutilizar código, reduzir acoplamento e reduzir parâmetros. Já o segundo grupo é composto por seis motivações que não apresentavam intenção de melhorar a qualidade do código. As motivações foram: atualização de *features*, adicionar *features*, correção de *bug*, remoção de *features*, melhorar o desempenho e melhorar a segurança.

O estudo também permitiu observar que as refatorações de *extract method* foram mais frequentes em métodos com números de linhas de código na faixa entre 19 e 34. Além disso, os resultados demostraram que esse tipo de refatoração é muito mais (aproximadamente 79% das ocorrências) impulsionado por mudanças nos requisitos do sistema como modificação de recursos, inserção de novas funcionalidades e correção de *bug*, e muito menos (16% das ocorrências) para melhoria da qualidade do código e resolução de *code smells*. A análise também não permitiu estabelecer uma relação clara entre as motivações para refatorar e o tamanho do método. As refatorações buscando melhorar a qualidade do código ocorrem, de forma quase uniforme, em métodos de todos os tamanhos.

Esses achados são importantes para aprimorar a caracterização da aplicação de refatorações de *extract method*, o que pode contribuir para o desenvolvimento de ferramentas de recomendação de refatoração que forneçam maior produtividade e assertividade aos desenvolvedores de *software*. Uma compreensão mais profunda das motivações dos desenvolvedores para aplicação do *extract method* pode auxiliar os pesquisadores na criação de sistemas de recomendação de refatorações mais alinhados com as necessidades reais dos desenvolvedores. Em geral, os sistemas de recomendação de refatoração se baseiam na resolução de *code smells*. No entanto, nossos resultados indicam que a maioria das refatorações aplicadas no estudo não tinha como intuito solucionar problemas de *code smells*. Portanto, analisar as motivações específicas por trás da aplicação de *extract method*, possibilitaria oferecer recomendações de refatorações mais relevantes, levando em consideração os problemas e intenções dos desenvolvedores, tornando as sugestões mais eficazes.

Essa pesquisa também tem o potencial de contribuir com a educação em engenharia de *software*. As refatorações desempenham um papel fundamental no desenvolvimento de código de qualidade. O conhecimento produzido neste estudo, especificamente sobre a refatoração de *extract method*, pode ser importante para professores e pesquisadores elaborarem materiais de ensino que abordem as motivações dos desenvolvedores para aplicar o *extract method*. Esses recursos educacionais podem tornar o aprendizado sobre refatorações mais prático, o que é fundamental para a formação de desenvolvedores que façam uso das boas práticas de codificação.

5.1 TRABALHOS FUTUROS

Como trabalhos futuros, apresentamos as seguintes propostas:

- Expandir esse estudo para considerar outros tipos de refatorações. Existem diversos outros tipos de refatorações que podem ser explorados em estudos que podem identificar as motivações por trás de suas ocorrências.
- Conduzir estudos que envolvam entrevistas com desenvolvedores para verificar qual a intenção deles quando aplicam as refatorações identificadas.
- Realizar estudos incluindo atributos de qualidade de código como compreensibilidade e manutenibilidade, de modo a verificar se as refatorações que apresentavam intenção de melhorar a qualidade do código apresentam melhorias significativas na qualidade do código em comparação com refatorações que não tem essa intenção.

REFERÊNCIAS BIBLIOGRÁFICAS

AALIZADEH, M. S. *Automatic Motivation Detection for Extract Method Refactoring Operations*. 76 f. Dissertação (Computer Science) — Concordia University, Montréal, 2021.

ABREU, F. B.; CARAPUÇA, R. Object-oriented software engineering: Measuring and controlling the development process. In: *Proceedings of the 4th international conference on software quality*. [S.l.: s.n.], 1994. v. 186.

ALOMAR, E.; MKAOUER, M. W.; OUNI, A. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In: *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. [S.l.: s.n.], 2019. p. 51–58.

BAVOTA, G. et al. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, v. 107, p. 1 – 14, 2015. ISSN 0164-1212. Disponível em: ⟨http://www.sciencedirect.com/science/article/pii/S0164121215001053⟩.

BOIS, B. D.; DEMEYER, S.; VERELST, J. Does the "refactor to understand"reverse engineering pattern improve program comprehension? In: *Ninth European Conference on Software Maintenance and Reengineering*. [S.l.: s.n.], 2005. p. 334–343.

CASTILHO, R. Eckart de et al. A web-based tool for the integrated annotation of semantic and syntactic structures. In: . Osaka, Japan: The COLING 2016 Organizing Committee, 2016. p. 76–84.

CEDRIM, D. et al. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 465–475. ISBN 9781450351058.

CRUZES, D. S.; DYBA, T. Recommended steps for thematic synthesis in software engineering. In: *2011 International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2011. p. 275–284.

DIG, D. et al. Automated detection of refactorings in evolving components. In: THOMAS, D. (Ed.). *ECOOP 2006 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 404–428. ISBN 978-3-540-35727-8.

DÓSEA, M.; SANT'ANNA, C. S.; SILVA, B. C. da. How do design decisions affect the distribution of software metrics? In: IEEE. *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. [S.l.], 2018. p. 74–7411.

FIGUEIREDO, E. et al. Evolving software product lines with aspects. In: IEEE. *2008 ACM/IEEE 30th International Conference on Software Engineering*. [S.l.], 2008. p. 261–270.

FONTANA, F. A. et al. Automatic metric thresholds derivation for code smell detection. In: IEEE. *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*. [S.l.], 2015. p. 44–53.

FOWLER, M. et al. *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201485672.

HENRIQUE, J.; DÓSEA, M.; SANT'ANNA, C. Minerando motivações para aplicação de extract method: Um estudo preliminar. In: SBC. *Anais do IX Workshop de Visualização, Evolução e Manutenção de Software*. [S.l.], 2021. p. 21–25.

KIM, M. et al. Ref-finder: A refactoring reconstruction tool based on logic query templates. In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2010. (FSE '10), p. 371–372. ISBN 9781605587912. Disponível em: ⟨https://doi.org/10.1145/1882291.1882353⟩.

LANZA, M.; MARINESCU, R. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 3642063748.

LIU, W.; LIU, H. Major motivations for extract method refactorings: analysis based on interviews and change histories. *Frontiers of Computer Science*, Springer, v. 10, n. 4, p. 644–656, 2016.

LUZGIN, V. A.; KHOLOD, I. I. Overview of mining software repositories. In: *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. [S.l.: s.n.], 2020. p. 400–404.

MARINESCU, R. Detection strategies: metrics-based rules for detecting design flaws. p. 350–359, 2004.

MARTIN, R. C. *Clean code: a handbook of agile software craftsmanship*. [S.l.]: Pearson Education, 2009.

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

MURPHY-HILL, E.; PARNIN, C.; BLACK, A. P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, v. 38, n. 1, p. 5–18, 2012.

NASCIMENTO, R.; SANT'ANNA, C. Investigating the relationship between bad smells and bugs in software systems. In: *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse*. [S.l.: s.n.], 2017. (SBCARS '17), p. 10. ISBN 9781450353250.

NISA, I. U.; AHSAN, S. N. Fault prediction model for software using soft computing techniques. In: *2015 International Conference on Open Source Systems Technologies (ICOSST)*. [S.l.: s.n.], 2015. p. 78–83.

NUÑEZ, A. S. et al. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, v. 128, p. 164 – 197, 2017. ISSN 0164-1212.

OIZUMI, W. et al. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2016. p. 440–451.

OPDYKE, W. F. *Refactoring object-oriented frameworks*. [S.l.]: University of Illinois at Urbana-Champaign, 1992.

PAIXÃO, M. et al. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. [S.l.]: Association for Computing Machinery, 2020. ISBN 9781450375177.

PANTIUCHINA, J. et al. Why developers refactor source code: A mining-based study. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, v. 29, n. 4, sep 2020. ISSN 1049-331X. Disponível em: ⟨https://doi.org/10.1145/3408302⟩.

PARNAS, D. L. Software aging. In: IEEE. *Proceedings of 16th International Conference on Software Engineering*. [S.l.], 1994. p. 279–287.

PRESSMAN, R. S. *Engenharia de Software: Uma Abordagem Profissional*. 8st. ed. [S.l.]: McGraw-Hill Education, 2016. ISBN 9788580555349.

REBAI, S. et al. Recommending refactorings via commit message analysis. *Information and Software Technology*, v. 126, p. 106332, 2020. ISSN 0950-5849.

SHARMA, T.; SPINELLIS, D. A survey on software smells. *Journal of Systems and Software*, v. 138, p. 158–173, 2018. ISSN 0164-1212.

SILVA, D. et al. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, v. 47, n. 12, p. 2786–2802, 2021.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of github contributors. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016. (FSE 2016), p. 858–870. ISBN 9781450342186.

SILVA, D.; VALENTE, M. T. Refdiff: Detecting refactorings in version histories. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2017. p. 269–279.

SILVA, D. F. *Recommending automated extract method refactorings*. 68 f. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal de Minas Gerais, Belo Horizonte, 2014.

SINGH, Y.; SAHA, A. Prediction of testability using the design metrics for object-oriented software. *Int. J. Comput. Appl. Technol.*, Inderscience Publishers, Geneva 15, CHE, v. 44, n. 1, p. 12–22, jul 2012. ISSN 0952-8091. Disponível em: ⟨https://doi.org/10.1504/IJCAT.2012.048204⟩.

SOARES, G. et al. Analyzing refactorings on software repositories. In: *2011 25th Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2011. p. 164–173.

SOMMERVILLE, I. *Engenharia de software*. Pearson Prentice Hall, 2011. ISBN 9788579361081. Disponível em: ⟨https://books.google.com.br/books?id=H4u5ygAACAAJ⟩.

STROGGYLOS, K.; SPINELLIS, D. Refactoring–does it improve software quality? In: *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. [S.l.: s.n.], 2007. p. 10–10.

TOM, E.; AURUM, A.; VIDGEN, R. An exploration of technical debt. *ournal of Systems and Software*, Morgan Kaufmann, v. 86, n. 6, p. 1498 – 1516, 2013. ISSN 0164-1212.

TSANTALIS, N.; KETKAR, A.; DIG, D. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, p. 1–1, 2020.

TSANTALIS, N. et al. Accurate and efficient refactoring detection in commit history. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2018. p. 483–494.

VALE, G.; FERNANDES, E.; FIGUEIREDO, E. On the proposal and evaluation of a benchmark-based threshold derivation method. *Software Quality Journal*, Springer, v. 27, n. 1, p. 275–306, 2019.

VALE, G. A. D.; FIGUEIREDO, E. M. L. A method to derive metric thresholds for software product lines. In: IEEE. *2015 29th Brazilian Symposium on Software Engineering*. [S.l.], 2015. p. 110–119.

VALENTE, M. T. *Engenharia de Software Moderna*. Independente, 2020. ISBN 6500019504. Disponível em: ⟨https://engsoftmoderna.info/⟩.

VOLDER, K. D. *Type-oriented logic meta programming*. Tese (Doutorado) — Citeseer, 1998.

YAMASHITA, A.; MOONEN, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2013. p. 682–691.