Trabalho de Conclusão de Curso

Explorando métricas de código para a detecção de *Long Envious Methods*

Audrey Emmely Rodrigues Vasconcelos
aerv@ic.ufal.br

Orientadores:

Prof. Dr. Baldoino Fonseca dos Santos Neto
MSc. Ana Carla Gomes Bibiano

Maceió, Janeiro de 2023

Audrey Emmely Rodrigues Vasconcelos

Explorando métricas de código para a detecção de *Long Envious Methods*

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Prof. Dr. Baldoino Fonseca dos Santos Neto

MSc. Ana Carla Gomes Bibiano

Maceió, Janeiro de 2023

Catalogação na Fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

V331e Vasconcelos, Audrey Emmely Rodrigues.
Explorando métricas de código para a detecção de *Long Envious Methods* / Audrey Emmely Rodrigues Vasconcelos. – 2023.
29 f. : il.

Orientador: Baldoino Fonseca dos Santos Neto.
Co-orientadora: Ana Carla Gomes Bibiano.
Monografia (Trabalho de conclusão de curso em Ciência da Computação) – Universidade Federal de Alagoas, Instituto de Computação. Maceió, 2023.

Bibliografia: f. 24-29.

1. Anomalias de código. 2. *Software* - Qualidade. 3. Métricas de código. I. Título.

CDU: 004.4

Agradecimentos

Gostaria de agradecer primeiramente aos meus pais, que me deram o suporte necessário para que eu chegasse até aqui e também por todo o sacrifício realizado, mesmo diante de dificuldades, para sempre garantir uma educação de qualidade para mim.

Ao meu orientador Prof. Dr. Baldoino Fonseca dos Santos Neto, pela oportunidade e apoio na elaboração deste trabalho. À minha coorientadora MSc. Ana Carla Gomes Bibiano por todo apoio dado durante este trabalho que foi essencial para a elaboração e finalização dele.

A todos os projetos de pesquisa e extensão que participei, em especial ao Laboratório de Engenharia e Sistemas (EASY), agradeço aos membros e aos professores coordenadores por todo conhecimento adquirido ao longo desses anos. Sem dúvida alguma, essa experiência contribuiu de forma essencial para a minha vida acadêmica e profissional.

A todos os meus amigos do Instituto de Computação que sempre me ajudaram e me apoiaram durante esses anos na graduação.

Por fim, e não menos importante, agradeço à banca examinadora, pela leitura atenta, questionamentos e sugestões.

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

– Fowler, *Martin*

Audrey Vasconcelos

Resumo

Atualmente os projetos estão em constante evolução para introduzir novas funcionalidades e adaptá-las a diferentes contextos de execução. Com isso, alguns problemas na qualidade do projeto de *software* podem ser introduzidos, podendo estar relacionados à presença de anomalias de código, geralmente chamadas de *code smells* ou *bad smells* em inglês, que são estruturas pobres de código. As anomalias são categorizadas em tipos e alguns desses tipos são *Long Method*, quando um método tem várias linhas de código, e *Feature Envy*, quando um método geralmente usa recursos de outras classes. Como a identificação manual dessas anomalias de código é custosa, várias ferramentas de anomalias de código foram propostas. Essas ferramentas utilizam estratégias de detecção baseadas em métricas de código, mas elas focam na remoção de uma única anomalia e um estudo recente indicou que um elemento de código pode ter duas ou mais anomalias de código. Assim, esses elementos podem estar com suas métricas degradadas e os limiares utilizados por essas ferramentas não são suficientes para indicar a gravidade deles. Essa monografia visa explorar métricas de código de métodos que são *Long Method* e *Feature Envy*. Esses métodos são chamados de *Long Envious Method*, um novo tipo de anomalia que pode ser causado por excessivas linhas de código e a implementação de duas ou mais funcionalidades. O objetivo é explorar quais métricas de código podem ser usadas para a identificação desse método. Mais de 8.000 instâncias de anomalias de código foram investigadas em três projetos de software, onde 4.707 (54%) são *Long Methods* e 3.910 (45.3%) métodos são *Long Envious Methods*. Ou seja, *Long Envious Methods* são frequentes e muitas vezes as ferramentas não conseguem detectá-los. Pode-se observar também que as métricas *cyclomaticComplexity* (complexidade do código), *NumberOfCatchStatements* (tratamento de exceções) e *couplingIntensity* (acoplamento do código fonte) têm uma diferença significativa entre *Long Envious Methods* e *Long Methods*. Por fim, conclui-se que essas métricas podem ser usadas para a identificação de *Long Envious Methods*.

Palavras-chave: Anomalias de código, qualidade de software, métricas de código, ferramentas de detecção

Abstract

Currently, projects are constantly evolving to introduce new features and adapt them to different execution contexts. With this, some problems in the quality of the software project can be introduced, which may be related to the presence of code anomalies, usually called code smells or bad smells in English, which are poor code structures. Anomalies are categorized into types and some of these types are Long Method, when a method has several lines of code, and Feature Envy, when a method often uses features from other classes. As manual identification of these code anomalies is costly, several code anomaly tools have been proposed. These tools use detection strategies based on code metrics, but they focus on removing a single anomaly and a recent study indicated that a code element can have two or more code anomalies. Thus, these elements may have their metrics degraded and the thresholds used by these tools are not sufficient to indicate their severity. This monograph aims to explore code metrics of methods that are Long Method and Feature Envy. These methods are called the Long Envious Method, a new type of anomaly that can be caused by excessive lines of code and the implementation of two or more features. The goal is to explore which code metrics can be used to identify this method. More than 8000 instances of code anomalies were investigated in three software projects, where 4707 (54%) are Long Methods and 3910 (45.3%) methods are Long Envious Methods. That is, Long Envious Methods are frequent and tools often fail to detect them. It can also be seen that the metrics cyclomaticComplexity, NumberOfCatchStatements and couplingIntensity have a significant difference between Long Envious Methods and Long Methods. Finally, it is concluded that these metrics can be used to identify Long Envious Methods.

Key-words: Code smells, software quality, code metrics, detection tools

Lista de Tabelas

Conteúdo

1

Introdução

Neste capítulo é introduzido a contextualização e a motivação desta pesquisa, a qual tem por finalidade explorar métricas de código para a detecção de *Long Envious Methods*. Na primeira seção são discutidas a contextualização e motivação dessa monografia. Na segunda seção as limitações serão explanadas. E, por fim, na terceira e última seção, a estrutura da monografia será descrita, com a organização dos capítulos restantes.

1.1 Contextualização e Motivação

As evoluções tecnológicas ocorridas na última década têm importância fundamental nas empresas - sejam elas públicas ou privadas - pois auxiliam a troca de informações que quando transformadas em conhecimento podem ser valiosas cultural, econômica e socialmente. A aplicação de metodologias de desenvolvimento e a escrita eficiente de códigos são de grande valia para as organizações, visto que projetos de *software* bem desenvolvidos agregam valor aos serviços e dessa maneira as equipes de desenvolvimento desempenham um papel cada vez mais estratégico (Ghapanchi et al., 2014).

A construção e entrega de projetos de *software* com qualidade passou a ser uma atividade crítica para as organizações, havendo a necessidade de ter um planejamento onde os possíveis riscos devem ser mitigados. Esses projetos evoluem continuamente para introduzir novas funcionalidades e adaptá-los a diferentes contextos de execução. Dessa forma, alguns problemas na qualidade do projeto de *software* podem ser introduzidos (Lehman and Belady, 1985), como lentidão para execução de algumas funcionalidades. Em tal situação, e especialmente quando o tempo é crucial ou quando existem outros prazos rigorosos, os desenvolvedores são obrigados a tomar decisões de design e implementar mudanças no menor tempo possível (Brown et al., 2010; Kruchten et al., 2012; Shull et al.). Como consequência direta, os desenvolvedores nem sempre têm a chance de aplicar mudanças que preservem a qualidade do projeto do sistema,

podendo levar à introdução de problemas na qualidade do projeto, também conhecidos como dívida técnica (Avgeriou et al., 2016; Izurieta et al., 2016).

Entre os diversos tipos de dívida técnica, um dos muitos problemas que envolvem o código é a presença de anomalias de código. Fowler (1999) define anomalias de código, mais comumente conhecidas como *code smell* ou *bad smell*, em inglês, como estruturas pobres de código. Essas estruturas podem ser representadas por funções (métodos) muito complexos ou com muitas linhas de código, ou que implementam múltiplas funcionalidades. Essas anomalias de código podem tornar o projeto não performático, complexo, difícil de mudar e manter. Fowler (1999) cita algumas anomalias de código em seu livro. Alguns tipos citados são *Long Method* que é quando um método tem múltiplas linhas código e *Feature Envy* que é quando um método utiliza muitas vezes funcionalidades de outras classes.

Macia et al. mostra que a maioria dos problemas de arquiteturais do código surgiu de elementos com anomalias de código. Seus resultados sugerem que a remoção sistemática dessas anomalias pode ser usada para combater efetivamente os sintomas de degradação da arquitetura do sistema. Dito isso, é indispensável a identificação dessas anomalias, preferencialmente o mais cedo possível no processo de desenvolvimento, possibilitando dessa maneira reduzir o esforço e o custo de manutenção. Esses problemas podem ser resolvidos por meio de transformações no código, sendo comum um tipo de transformação conhecida como refatoração. A refatoração melhora a extensibilidade, manutenção, compreensibilidade e reutilização do *software*, visando aprimorar sua estrutura sem alterar o seu comportamento externo (Fowler, 1999). Os desenvolvedores aplicam a refatoração de código com o objetivo de remover as anomalias de código (Silva et al., 2016). Apesar de seus benefícios, a refatoração não é uma atividade trivial. Um dos principais passos para aplicação de refatoração é identificar quais elementos de código (pacotes, classes e métodos) estão sendo afetados por anomalias de código. No entanto, na prática, a identificação manual de anomalias de código é muito custosa, pois desenvolvedores precisam olhar muitos elementos de código e entender quais tipos de anomalias afeta o código para saber qual transformação de refatoração é mais apropriada para a remoção dessa anomalia.

1.2 Problemática

A identificação de anomalias de código torna-se ainda mais custosa em grandes sistemas de *software*, e pode vir a ser uma atividade propensa a erros que consome tempo e recursos. Ao longo do tempo, várias ferramentas de detecção de anomalias de código foram propostas, dentre elas podemos citar o JDeodorant, DECOR, PMD e Organic. Essas ferramentas utilizam estratégias de detecção baseadas em métricas de código. As métricas de código são meios para avaliar a qualidade e identificar anomalias relacionadas ao projeto de *software* (Marinescu, 2004). Tradicionalmente, elas utilizam métricas como: tamanho da classe, acoplamento, coesão e complexidade para avaliar o *software*. Tais métricas tradicionais são vistas como uma

solução prática para encontrar sintomas de anomalias de código. Em outras palavras, esses detectores analisam a qualidade do código através dessas métricas. Se os valores dessas métricas de código não estiverem dentro dos limiares esperados, então isso pode ser um indicador que há uma anomalia de código ali. Por exemplo, algumas ferramentas utilizam um limiar (*threshold*) aceitável para linhas de código de um método como 100 linhas de código, se um método tiver mais linhas que esse limiar, então esse método é considerado um *Long Method*, ou seja, um método longo.

Apesar do bom desempenho mostrado pelos detectores, estudos recentes destacam uma série de limitações importantes que ameaçam a adoção dos detectores na prática (Fernandes et al., 2016; Zhang et al., 2011). Em primeiro lugar, anomalias de códigos detectadas por abordagens existentes podem ser subjetivamente percebidas e interpretadas pelos desenvolvedores (Fontana et al., 2016a; Mäntylä and Lassenius, 2006), cada desenvolvedor tem uma visão diferente como deve ser mensurado esses limiares usados pelas ferramentas e além disso as ferramentas não deixam claro como esses limiares foram definidos. Em segundo lugar, a concordância entre os detectores é baixa, o que significa que diferentes ferramentas podem identificar as anomalias de diferentes elementos de código (Fontana et al., 2012). Algumas técnicas de aprendizado de máquina (*machine learning*) estão sendo adotadas para mitigar esses problemas nas anomalias de códigos (Fontana et al., 2016b). Normalmente, um método supervisionado é explorado, ou seja, um conjunto de variáveis independentes (também conhecidas como preditores) é usado para determinar o valor de uma variável dependente (ou seja, presença de uma anomalia ou grau de anomalia de um elemento de código) usando um aprendizado de máquina classificador (por exemplo, regressão logística (Alpaydin, 2014)).

Porém, essas abordagens focam na remoção de uma única anomalia de código, mas um estudo recente Bibiano et al. (2021) indicou que um elemento de código pode ter duas ou mais anomalias de código. Dessa forma, esses elementos de código podem estar com as métricas de código altamente degradadas e os limiares utilizados por essas ferramentas não são suficientes para indicar a gravidade do problema estrutural que tem esses elementos de código. A detecção dessas anomalias de código compostas (formadas por dois ou mais tipos de anomalias de código) é crucial, principalmente porque isso também pode afetar a escolha da(s) refatoração(ões) que deve(m) ser aplicada(s). Bibiano et al. (2021) aponta que anomalias de código que podem estar acontecendo frequentemente são *Long Method* e *Feature Envy*. No trabalho deles, eles observaram que métodos que são longos geralmente podem ter a anomalia de código *Feature Envy*. No entanto, eles não apresentaram a quantidade de métodos que tem essas anomalias de código e também não apresentaram os valores das métricas de código que tem esses métodos. A análise sobre as métricas de código desses métodos é muito importante para o melhoramento das estratégias de detecção, principalmente quando um elemento de código tem mais de uma anomalia de código.

1.3 Metodologia

Dada as problemáticas apresentadas na seção anterior, esse trabalho de conclusão de curso visa explorar métricas de código de métodos que são *Long Method* e *Feature Envy*. Esses métodos são chamados de *Long Envious Method*, um novo tipo de anomalia de código que pode ser causado por excessivas linhas de código e a implementação de duas ou mais funcionalidades. Foi escolhido investigar essa anomalia de código composta, pois estudos indicam que *Long Methods* e *Feature Envies* são as anomalias mais frequentes, também podem afetar múltiplas classes, e são as anomalias que desenvolvedores mais almejam a remoção. Logo, a identificação da composição dessas duas anomalias de código é de suma importância para pesquisadores e desenvolvedores. O objetivo é explorar quais métricas de código podem ser usadas para a identificação de *Long Envious Method*, como foi dito na Seção de Problemática, pode ser que as métricas e limiares usados nas ferramentas existentes não sejam suficientes para identificar um anomalia de código composta, nesse caso um *Long Envious Method*.

Foram investigados mais de 8.000 instâncias de anomalias de código em três projetos de *software* de diferentes domínios e dimensões. Essas instâncias de anomalias de código foram detectadas a partir do Organic, uma ferramenta que tem uma alta precisão de detecção, de acordo com estudos anteriores. O principal benefício para a utilização do Organic é que essa ferramenta fornece todos os valores de todas as métricas de código do método, mesmo que não sejam métricas de código usadas na estratégia de detecção da anomalia de código. Após a coleta das anomalias de código, (i) computou-se quantos métodos têm apenas *Long Method* e quantos métodos têm simultaneamente *Long Method* e *Feature Envy*, sendo estes possíveis *Long Envious Methods*; (ii) identificou-se quais métricas de código tem valores significativamente diferentes de métodos que são *Long Methods* e de métodos que são possíveis *Long Envious Methods*, para o cálculo dessa diferença significativa, foram calculados a média, mediana e realizado o teste estatístico F-Fisher (também conhecido como Teste Exato de Fisher).

1.4 Resultados

Foram encontrados 4.707 (54%) *Long Methods* e 3.910 (45.3%) métodos que são *Long Envious Methods*. Isso indica que *Long Envious Methods* são frequentes e muitas vezes as ferramentas só indicam que esses métodos são somente longos, porém há mais problemas estruturais nesses métodos. Além disso, a diferença entre a quantidade de *Long Methods* e *Long Envious Methods* pode variar por projeto. Um exemplo disso são os projetos Activiti e Checkstyle. No projeto Activiti, a ocorrência de *Long Envious Method* é de 1.665 (67%) métodos, indicando assim, que nesse projeto a maioria dos métodos implementam múltiplas funcionalidades. Porém, para o Checkstyle, 2.514 (77%) de 3.247 métodos são *Long Methods*, sugerindo então que esses métodos tem somente muitas linhas de código. Dessa maneira, propõe-se que pesquisadores e construtores de ferramentas de detecção de anomalias e de recomendação de refatoração

devem observar porque a maioria dos métodos implementam mais de uma funcionalidade para alguns projetos, e para outros projetos os métodos são simplesmente longos. O entendimento sobre essas práticas de desenvolvimento podem ajudar a melhorar a construção de ferramentas de detecção de anomalias e de recomendação de refatoração, aumentando assim a aceitação dos desenvolvedores para a utilização dessas ferramentas.

Além disso, após observar a média, mediana e o teste estatístico F-Fisher com as métricas de código de métodos *Long Method* e *Long Envious Method*, levando em consideração os 3 projetos, conclui-se que as métricas *cyclomaticComplexity*, *maxNesting*, *MethodEffectiveLinesOfCode* e *NumberOfCatchStatements* podem ser usadas para a identificação de *Long Envious Methos* com limiares inferiores. É perceptível que essas métricas têm valores maiores frequentemente quando *Long Methods* são encontrados. Isso indica que métodos que são longos e que implementam funcionalidades somente inerentes à classe, tendem a ter uma complexidade maior (*cyclomaticComplexity*), um maior aninhamento de chamadas de métodos *maxNesting*, maior quantidade de linhas (*MethodEffectiveLinesOfCode*), e maior quantidade de tratamento de exceções (*NumberOfCatchStatements*). Esse resultado é surpreendente, pois esperava-se que essas métricas poderiam estar mais degradadas quando *Long Envious Methods* são encontrados. Supõe-se para esse resultado que desenvolvedores tendem a aumentar a complexidade e tamanho do método quando este implementa somente funcionalidades da classe de origem, pois isso pode facilitar a manutenção do método.

Também é notório que a métrica *couplingIntensity* tem uma diferença significativa, principalmente porque *Long Envious Methods* tende a ter um intenso acoplamento comparado a *Long Methods*, indicando que esses métodos (*Long Envious Methods*) tendem a ter uma implementação de múltiplas funcionalidades de outras classes concentradas em um grupo significativo de métodos. Logo, a conclusão é que a métrica *couplingIntensity* pode ser usada para a identificação de *Long Envious Method* com limiares superiores.

1.5 Estrutura do Trabalho

Finalmente, esse trabalho de conclusão de curso está estruturado da seguinte forma: O Capítulo 2 apresenta os principais conceitos para esse estudo, definindo o conceito de *code smells*, estratégias de detecção e *Long Envious Method*. A metodologia está detalhada no Capítulo 3, onde é apresentado o objetivo desse estudo, as questões de pesquisa e o principais passos do estudo. Os resultados e contribuições são explicados no Capítulo 4. A conclusão desse trabalho está no Capítulo 5.

2

Fundamentação teórica

Neste capítulo são apresentados os conceitos que fundamentam o desenvolvimento deste trabalho. Na seção 2.1 é introduzida a definição de *code smells*. Na seção 2.2 as estratégias para a detecção dos *code smells* e suas limitações são definidas. Por fim, na seção 2.3 é apresentado o *Long Envious Method*, sua definição e problemática.

2.1 Definição de *code smell*

Code smells são anomalias de código que podem indicar problemas relacionados a aspectos da qualidade do código, e como consequência, essas anomalias podem causar problemas para os desenvolvedores nas atividades durante a fase de manutenção de *software* (Fowler, 1999). Elas podem afetar qualquer tipo de sistema (Macia et al., 2012; Oizumi et al., 2016). Há muitas situações que podem causar essas anomalias, tais como dependências inadequadas entre módulos, uma atribuição incorreta de métodos a classes ou duplicação desnecessária de segmentos de código, podendo eventualmente causar problemas profundos de desempenho e dificultar a manutenção de aplicações críticas para os negócios. O impacto e os efeitos negativos que anomalias de códigos podem ocasionar são diversos, como por exemplo: diminuição da qualidade do design de *software* e impacto negativo nos atributos de qualidade (e.g., manutenibilidade, legibilidade e modificabilidade do código). Essas anomalias também se mostraram prejudiciais para uma boa arquitetura de *software* causando sua degradação, além de aumentar as violações de modularidade de um sistema (Lacerda et al., 2020). A maioria dos trabalhos da literatura sobre anomalias de códigos focam em sistemas orientado a objetos (OO) (Kaur and Dhiman, 2018). O conceito de *code smells* é proveniente da percepção de um código "cheirar mal".

Diferente de um *bug* ou defeito no código de um sistema, a presença de anomalias de códigos não significa a presença de defeitos no *software*. No entanto, essas anomalias podem trazer outras consequências como impacto de forma negativa na manutenção e evolução de um

determinado sistema (Lacerda et al., 2020). Trabalhos anteriores (Macia et al., 2012; Oizumi et al., 2016; Mello et al., 2019; Uchôa et al., 2020) mostraram evidências de que anomalias de códigos são fortes indicadores de partes do código afetadas por pobre decomposição de *features* em sistemas de *software*.

Os *code smells* foram catalogados, o catálogo mais amplamente usado foi compilado por Fowler (1999), e descreve 22 tipos de anomalias de código existentes. Outros pesquisadores, como van Emden and Moonen (2002), propuseram posteriormente mais tipos de anomalias. Nos últimos anos, o *code smell* foi catalogado para outras linguagens de programação orientadas a objetos, além de Java, como Matlab e Python (dos Reis et al., 2020). Nesse trabalho abordaremos 2 dessas anomalias, sendo elas *Long Method* e *Feature Envy*, em códigos que foram escritos na linguagem Java.

Long Method, ou simplesmente LM – Essa anomalia de código surge quando um método implementa uma funcionalidade principal junto com funções auxiliares que deveriam ser gerenciadas em diferentes métodos. A refatoração associada a tal anomalia é o *Extract Method*, que permite identificar partes do método que devem ser tratadas separadamente, com o objetivo de criar novos métodos para gerenciá-los (Fowler, 1999). Vale ressaltar que a definição da anomalia difere fortemente de seu nome, pois esse *smell* está apenas parcialmente relacionado ao tamanho de um método. Em vez disso, está relacionado se o método gerencia mais de uma responsabilidade da classe, ou seja, se um método viola o princípio da responsabilidade única.

Feature Envy, ou simplesmente FE – De acordo com a definição de Fowler (1999), este *smell* ocorre quando um método está mais interessado em outra classe do que aquela em que realmente está. Assim, um método afetado por *Feature Envy* não é colocado corretamente, pois exibe alto acoplamento com uma classe diferente daquela em que está localizado. Para remover esse *smell*, é necessário uma refatoração *Move Method* cujo objetivo é movê-lo para a classe mais adequada.

Infelizmente, detectar manualmente *code smells* na prática não é fácil. Muitas definições de anomalia de código são vagas e as decisões de refatoração dependem fortemente da intuição e experiência humanas (Fowler, 1999). Os desenvolvedores frequentemente precisam detectar anomalias em códigos desconhecidos, o que é um desafio. Uma solução para esse problema é desenvolver ferramentas para detectar essas anomalias no código automaticamente, e muitas dessas ferramentas já foram propostas (Kovačević et al., 2022).

2.2 Estratégias de Detecção

A maioria das estratégias de detecção identificam anomalias de código a partir do código. Elas aplicam regras de detecção para extrair anomalias de códigos dos projetos de *software*. Muitas estratégias de detecção de anomalia de código usam métricas de código para detectar essas anomalias (Macia et al., 2013). Essas estratégias calculam métricas diretamente do có-

digo ou usam métricas extraídas de ferramentas de terceiros, tais com o número de linhas de código, o nível de complexidade de um método. Elas aplicam limiares (*thresholds*) em métricas obtidas do código em análise e parecem mais atraentes para automação de suporte devido à sua reutilização. Exemplo desses limiares são um número limite para quantidade de linhas de código de uma classe, se uma classe tiver mais que esse número limite, então essa classe tem uma determinada anomalia de código relacionada a essa métrica de código. No entanto, essas estratégias têm duas pré-condições importantes para sua aplicação efetiva: Em primeiro lugar, o mesmo conjunto de métricas não pode ser usado para detectar todas as anomalias, por causa das diferentes variações de implementação de anomalias de código semelhantes. Em segundo lugar, o modelo resultante deve ser calibrado com um conjunto personalizado de métricas que impliquem uma validação empírica com base em dados de classificação existentes (Rasool and Arshad, 2015).

Ferramentas para detecção automática ou semiautomática de *code smells* auxiliam os desenvolvedores na identificação de entidades com anomalias. A implementação de estratégias de detecção permite que as ferramentas destaquem as entidades que provavelmente apresentam mais anomalias de códigos. Felizmente, existem muitas ferramentas de análise de *software* disponíveis para detectar *code smells* (Fernandes et al., 2016; Murphy-Hill and Black, 2010; Tsantalis et al., 2008). Em geral, esse fato indica uma conscientização da comunidade de engenharia de *software* sobre a importância do controle da qualidade estrutural das funcionalidades em desenvolvimento (Fontana et al., 2012). Por outro lado, traz um novo desafio sobre como avaliar e comparar ferramentas e selecionar a ferramenta mais eficiente em contextos de desenvolvimento específicos.

Na literatura, muitos estudos abordaram o problema de detecção de anomalias de códigos e refatoração. As estratégias de detecção podem ser classificadas principalmente em: abordagens manuais, abordagens baseadas em métricas, análise de código estático, análise de alterações de projetos de *software* e abordagens baseadas em pesquisa (Paiva et al., 2017).

Abordagens manuais. A maneira mais confiável de detectar *code smells* manualmente segundo Fowler (1999) é por revisões de código, onde o código é inspecionado para identificar fragmentos de código com anomalias. Essa abordagem apresenta muitas desvantagens, por exemplo, é demorado, não repetível, propenso a erros e não escalável (Mäntylä et al., 2004; Mäntylä and Lassenius, 2006; Marinescu, 2001; Travassos et al., 1999). No geral, abordagens manuais exigem um grande esforço humano para interpretar e analisar o código e, consequentemente, eles estão restritos a pequenos sistemas. Outra questão é que a detecção manual é altamente subjetiva, contando com a experiência do desenvolvedor e seus conhecimentos do sistema e seu domínio.

Abordagens baseadas em métricas de código. A maioria das abordagens propostas consiste em identificar uma anomalia de código combinando métricas de código em regras de detecção. As regras são compostas por: (i) identificar todos os sintomas (tais como múltiplas linhas de código ou implementação de múltiplas responsabilidades) a partir da definição do

code smell, (ii) mapear os sintomas para as métricas de código correspondentes e seus limites (por exemplo, linhas de código > k), (iii) combinar os sintomas na regra final. As variações dessas técnicas estão agrupadas principalmente em um conjunto de métricas selecionadas, seus limiares e como eles são combinados para uma determinada anomalia de código. Por exemplo, Marinescu (2004) define um mecanismo chamado de estratégia de detecção, que combina métricas usando os operadores lógicos AND/OR. Outra abordagem de Moha et al. (2010) apresentou o DECOR, um método para especificar e detectar anomalias usando uma linguagem específica de domínio.

Análise de código estático. A proposta da análise estática é identificar muitos problemas comuns de codificação automaticamente antes de um programa ser lançado. A análise estática visa examinar o texto de um programa estaticamente, sem tentar executá-lo. Teoricamente, as ferramentas de análise estática, tais como PMD, Organic e Checkstyle, podem examinar o código de um programa ou uma forma compilada do programa para igual benefício, embora o problema de decodificar o último possa ser difícil (Gomes et al., 2009).

Análise de mudanças no projeto de *software*. Para essa abordagem, as anomalias de código são caracterizadas a partir de mudanças no código ao longo do tempo a partir de métricas de código ou outras informações extraídas de um *snapshot* do código. Por exemplo, Palomba et al. (2013) propõe HIST (*Historical Information for Smell deTection*) para detectar anomalias de códigos. Essa abordagem extrai o histórico de alterações de um projeto a partir de um sistema de controle de versão (ex: Git) e analisa as co-alterações entre os artefatos do código (Paiva et al., 2017).

Abordagens baseadas em pesquisa. A engenharia de *software* baseada em pesquisa usa abordagens baseadas em busca para resolver problemas de otimização. Uma vez que os problemas de engenharia de *software* são modelados como problemas de otimização, algoritmos de busca podem ser aplicados para resolvê-los. Kessentini et al. (2010) formulam a detecção de anomalias de códigos como um problema de otimização, onde os algoritmos detectam anomalias seguindo a suposição de que o que diverge significativamente das boas práticas de design provavelmente representa um problema de design. Outros estudos propõem abordagens baseadas em aprendizado de máquina que são capazes de encontrar anomalias de código relatando classes semelhantes às classes com anomalias dos dados de treinamento. No entanto, o principal desafio para essas abordagens é o alto nível de falsos positivos, uma vez que dependem da qualidade das instâncias de anomalias de código no conjunto de treinamento (Paiva et al., 2017).

Todavia, essas estratégias de detecção possuem diversas limitações, dentre elas podemos mencionar: (i) falta de alinhamento com a prática, por exemplo, estudos recentes mostram que algumas anomalias de código ocorrem frequentemente junto no mesmo escopo de código (método ou classe), mas as estratégias existentes geralmente focam em identificar somente a ocorrência de um *code smell*; (ii) falta de customização das estratégias de detecção de acordo com características dos projetos, a maioria das ferramentas detectam anomalias de códigos através

de *thresholds* estáticos. Logo, essas estratégias ficam limitadas a determinados limiares, mas cada projeto de *software* tem valores de métricas diferentes, então essas estratégias precisam se alinhar aos valores das métricas de cada projeto.

2.3 *Long Envious Method (LEM)*

Como foi dito na seção anterior, as estratégias existentes focam em detectar somente uma anomalia de código, mas estudos recentes mostram que alguns tipos de anomalias de códigos ocorrem juntas no mesmo escopo de código. Um exemplo disso é a ocorrência das anomalias de código *Long Method* e *Feature Envy* (definidos na Seção 2.1). Essas anomalias de código são as mais frequentes na prática e acreditava-se que elas aconteciam isoladamente na maioria das vezes. Porém, um estudo recente apresentou que métodos que são longos, geralmente tendem a ter *Feature Envy*, métodos com esses problemas estruturais são chamados *Long Envious Method* (ou LEM), de acordo com Bibiano (2022).

Para a remoção dessa anomalia, Bibiano (2022) propôs um catálogo de recomendações de refatorações baseado no desenvolvimento de múltiplos projetos reais. As recomendações de refatorações anteriores focavam na remoção de uma única anomalia de código, mas o catálogo fornecido no artigo mostra quatro recomendações que removem até três tipos de anomalias de códigos, agrupados em dois novos tipos de anomalias. Esse catálogo foi gerado através de resultados de um estudo anterior de Bibiano et al. (2021). Nesse estudo foi observado que algumas anomalias de código são removidas juntas com frequência, essas anomalias de código são *Long Method*, *Feature Envy* e *Duplicated Code*. Com base nisso, foram coletadas quais combinações de refatorações comuns removem completamente essas anomalias de código e seus efeitos colaterais. Dessa forma, foi observado que as anomalias de código geralmente ocorrem em pares de anomalias sobre o mesmo elemento de código. Sendo assim, foi criado um novo tipo de *code smell*. Chamado *Long Envious Method*, quando um *Long Method* e *Feature Envy* são detectados em um mesmo método. No estudo de Bibiano (2022) foram propostos outros tipos de anomalias de código, mas nesse trabalho de conclusão de curso é investigado o *Long Envious Method*, pois é formado por duas anomalias de código que são frequentemente detectadas de forma isolada por ferramentas existentes, como apresenta a Tabela 2.1.

Tabela 2.1: Ferramentas que Detectam Long Method e Feature Envy

Code Smell	**JDeodorant**	**Stench Blossom**	**InFusion**	**iPlasma**	**PMD**	**DECOR**	**Organic**
Feature Envy	X	X	X	X	-	-	X
Long Method	X	X	-	X	X	X	X

3

Metodologia

Esse capítulo apresenta o objetivo desse estudo, as questões de pesquisa e os principais passos para a condução desse estudo. O **objetivo geral** desse estudo é identificar métricas de código que possam ser usadas em estratégias de detecção da anomalia de código *Long Envious Method*. Dessa forma, tem-se os seguintes *objetivos específicos*:

- Avaliar se *Long Envious Method* é mais frequente que *Long Method* e *Feature Envy*, quando estes são encontrados isoladamente.
- Comparar valores de métricas de código em métodos que são *Long Envious Method* e métodos que só são *Long Method* ou *Feature Envy*.
- Agrupar valores de métricas de código que podem ser indicadores para identificar *Long Envious Method* por projeto de *software*.

3.1 Questões de Pesquisa

A partir do nosso objetivo geral e objetivos específicos, foram elaboradas as seguintes questões de pesquisa (QP).

Questão de Pesquisa Geral: Quais métricas de código podem ser indicadoras de um *Long Envious Method*?

QP_1: - Todo *Long Method* é um método que tem *Feature Envy*?

Long Method e *Feature Envy* são anomalias de código frequentes durante o desenvolvimento de projetos de *software*. Além disso, são anomalias de código que desenvolvedores frequentemente almejam detectar e remover, pois são problemas estruturais que podem envolver múltiplas classes, aumentando assim a degradação do código. Um estudo recente indica que métodos longos podem também ter uma *Feature Envy* (Bibiano et al., 2021), porém esse estudo

não apresenta a quantidade de métodos que possuem ambas as anomalias de código. Dessa maneira, buscou-se através dessa questão de pesquisa, investigar a quantidade de métodos que são *Long Method* e a quantidade de métodos que são ambos, *Long Method* e *Feature Envy*. Para responder essa pergunta de pesquisa, essas anomalias de código foram coletadas em 19 projetos de *software* e programas foram criados para computar a quantidade de métodos que possuem ambos os *code smells* no mesmo *commit*. Através dessa questão de pesquisa é possível indicar aos pesquisadores e desenvolvedores que métodos longos também tem outros problemas estruturais, tal como a alta dependência de métodos de outras classes (principal problemática da *Feature Envy*).

QP_2: - Quão significante é a diferença das métricas de código entre métodos que são só *Long Method* e possíveis *Long Envious Methods*?

Após analisar a frequência de métodos que possuem *Long Method* e *Feature Envy* simultaneamente (QP_1), é possível analisar se e quais métricas de código tem alguma diferença estatisticamente significante das métricas de código em métodos que são possíveis *Long Envious Method* em comparação com métodos que são somente longos. Através dessa análise podemos responder nossa segunda questão de pesquisa e identificar quais métricas podem ser relevantes para detectar *Long Envious Method*. Para responder essa pergunta de pesquisa, foram realizadas as seguintes análises: (i) coleta das médias e medianas de cada métrica de código para métodos LM e métodos LEM; (ii) avaliação de quais médias e medianas possuem diferença modular significante; (iii) execução do teste estatístico F-Fisher para validar quais métricas de código tem relevância significativa para a identificação de *Long Envious Method*.

3.2 Principais Passos do Estudo

Passo 1: Seleção de Projetos de *Software* – Foram selecionados 19 projetos de *software*. A escolha desses projetos se deu a partir dos seguintes critérios: (i) os projetos de *software* devem ser implementados em Java, pois além de Java ser uma das linguagens mais populares, a maioria das ferramentas de detecção de anomalias de código são para projetos Java (Bibiano et al., 2021); (ii) os projetos devem estar disponibilizados em sistemas de controle de versão Git, pois dessa forma podemos coletar o todas anomalias de código ao longo do histórico de desenvolvimento do projeto, também podemos garantir que os métodos possuem as duas anomalias de código investigadas ao mesmo tempo (no mesmo *commit*); projetos que foram investigados anteriormente por estudos empíricos sobre anomalias de código, dessa maneira é possível validar a presença de anomalias de código de acordo com bases de dados existentes e fornecidas por esses estudos.

Passo 2: Detecção de Anomalias de Código – Analogamente a estudos anteriores (Bibiano et al., 2020, 2021), foi utilizada a ferramenta Organic para detectar anomalias de código. Organic é uma ferramenta estática para a detecção de 19 tipos de anomalias de código. Orga-

Tabela 3.1: Anomalias de Código coletadas Nesse Estudo

Tipo de Anomalia de Código	ID	Definição
Nível de Método		
Brain Method	BrM	Método com implementação de excessivas funcionalidades
Dispersed Coupling	DsC	Método que chama muitos métodos
Divergent Change	DiC	Método que muda frequentemente com outros métodos
Feature Envy	FeE	Método que utiliza muitas funcionalidades de outras classes
Intensive Coupling	InC	Método que depende muito de outros métodos
Long Method	LoM	Método muito longo e complexo
Long Parameter List	LPL	Muitos parâmetros em um método
Message Chain	MeC	Cadeia muito longa de chamadas de método
Shotgun Surgery	ShS	Método cujas alterações afetam outros métodos
Nível de Classe		
Brain Class	BrC	Classe com implementação de excessivas funcionalidades
Class Data should be Private	CDSBP	Classe que superexpõe seus atributos
Complex Class	CoC	Implementação muito complexas em uma classe
Data Class	DaC	Excessivo número de dados (tais como atributos) em uma classe
God Class	GoC	Muitas funcionalidades em uma classe
Large Class	LgC	Classe muito grande
Lazy Class	LaC	Classe muito pequena e simples
Refused Bequest	ReB	Subclasse que raramente usa recursos da superclasse
Spaghetti Code	SpC	Muito desvio de código e aninhamento
Speculative Generality	SpG	Classe abstrata inútil

nic usa estratégias de detecção baseadas métricas de código para coletar anomalias de código. Essas estratégias de detecção foram avaliadas por estudos existentes (Fernandes et al.; Oizumi et al., 2016; Cedrim et al., 2017), comprovando assim a alta acurácia e precisão da ferramenta. A Tabela 3.1 apresenta os tipos de anomalias de código que Organic detecta. Essa ferramenta detecta anomalias de código de dois níveis de escopo de código: a nível de método e a nível de classe. *Long Method* e *Feature Envy* são anomalias de código, mas podem estar relacionadas com problemas estruturais de outras classes, por isso também coletou-se anomalias de código a nível de classe, para poder analisar essas anomalias de forma aprofundada. Organic só coleta anomalias de código a partir de um único *commit*, foi preciso criar programas para coletar as anomalias de todos os *commits* do histórico do projeto. Essas anomalias de código estão armazenadas em um banco de dados não relacional MongoDB [1](versão 6.0.3) e os programas foram desenvolvidos em Java e estão disponíveis em repositórios Github [2]. Foram armazenados dados tais como: o nome da anomalia do código, nome dos elementos de código (classe e método) que possuem as anomalias, e os valores das métricas dos elementos de código envolvidos nessas anomalias e o *commit* que as anomalias foram encontradas.

Passo 3: Validação Manual – Uma amostra de 36 anomalias de código do nosso banco de dados para a validação manual foi selecionada aleatoriamente. Seis desenvolvedores validaram se as anomalias de código foram encontradas. Esses desenvolvedores têm em média 6 a 10 anos de experiência em desenvolvimento. Cada desenvolvedor(a) teve uma semana para avaliar as anomalias de código de acordo com a disponibilidade deles. Eles validaram 28 anomalias de código e confirmaram que 24 amostras são de fato anomalias de código de acordo com a percepção dos desenvolvedores.

[1]https://www.mongodb.com/try/download/community
[2]https://github.com/

Passo 4: Separação de LM e LEM – Com a base de dados validada com as anomalias de código, era necessário então separar os *Long Methods* dos possíveis *Long Envious Method*. Com a criação de programas em Python [3] (versão 3.10.8) conseguimos filtrar os métodos que só são *Long Methods*[4] e os métodos que são possíveis LEM, isto é, métodos que possuem *Long Method* e *Feature Envy* simultaneamente (em termos do mesmo *commit*). Após isso, os resultados desses filtros foram salvos em duas tabelas em arquivos do tipo csv, uma para LM e outra para possíveis LEM. Dentro desses arquivos contém informações sobre o nome do método e classe envolvidos, nomes das anomalias de código encontradas, *commit* que foi detectada as anomalias e os valores das métricas de código dos elementos de código envolvidos.

Passo 5: Comparação entre LM e LEM – Após a separação dos métodos que são LM dos possíveis LEM, algumas análises foram executadas para comparar se há alguma diferença significativa entre as métricas de código de métodos que são LM e métodos que são possíveis LEM. A Tabela 3.2 apresenta as métricas de código exploradas nesse estudo. As siglas de cada métrica são apresentadas, os termos em inglês que são usados para descrever cada métrica e a descrição em português de cada métrica. Essas métricas são comumente investigadas para detectar anomalias de código (Cedrim et al., 2017; Bibiano et al., 2019, 2021) e para investigação sobre atributos de qualidade interna do código (Chávez et al., 2017; AlOmar et al., 2019; Bibiano et al., 2020), tais como coesão, complexidade, acoplamento, dentre outros. As médias e medianas de cada métrica dos métodos LM e dos possíveis LEM foram coletadas para todos os projetos, de um modo geral, e segmentando por cada projeto. Após isso, a diferença modular dessas médias e medianas foram computadas para verificar quais métricas possuem uma diferença significativa entre esses métodos. Para mitigar a ameaça de que as médias e medianas iguais ou próximas representam um conjunto de valores com diferenças não significantes, o teste estatístico F-Fisher ou também conhecido como Teste Exato de Fisher (Fisher et al., 1963; Arcuri and Briand, 2011) foi executado. Esse teste é usado para calcular a significância da diferença entre duas populações (Fisher et al., 1963; Arcuri and Briand, 2011). No caso desse estudo, as duas populações são A e B. Sendo A = o conjunto de valores de métrica M_i para os métodos LM e B = o conjunto de valores de métrica M_j para os possíveis métodos LEM; sendo j cada tipo de métrica de código. Por fim, as médias e medianas foram calculadas através de programas em Python e o Teste Exato de Fisher utilizando a linguagem R.

[3] https://www.python.org/downloads/

[4] Esses métodos podem ter outras anomalias de código que não são *Feature Envy*

Tabela 3.2: Métricas de Código investigadas Nesse Estudo

Nome	**Termo usado em inglês**	**Descrição**
CLOC	Class Lines Of Code	Número de linhas de código de classe
MLOC	Method Lines Of Code	Número de linhas de código de método
CC	Cyclomatic Complexity	Complexidade ciclomática
IsAbstract	IsAbstract	Define quando uma classe é abstrata
MaxCallChain	Max Call Chain	Tamanho máximo de uma cadeia de chamada de métodos
ParameterCount	Parameter Count	Quantidade de parâmetros
OverrideRatio	Override Ratio	Define quando um método é sobrescrito
PublicFieldCount	Public Field Count	Quantidade de atributos públicos
TCC	Tight Class Cohesion	Peso de uma coesão de classe
MaxNesting	Max Nesting	Tamanho máximo de uma aninhamento de métodos
NOAV	Number Of Accessed Variables	Número de variáveis acessadas
NOAM	Number Of Accessor Methods	Número de acessadores de métodos
WMC	Weighted Method Count	Peso de complexidade de um método
WOC	Weigh Of Class	Peso de complexidade de classe
CINT	Coupling Intensity	Intensidade de acoplamento
CDISP	Coupling Dispersion	Dispersão de acoplamento
ChangingClasses	Changing Classes	Quantidade de mudanças em classes
ChangingMethods	Changing Methods	Quantidade de mudanças em métodos
LCOM	Lack Of Cohesion Of Methods	Falta de coesão de métodos (limiar 1)
LCOM2	Lack Of Cohesion Of Methods 2	Falta de coesão de métodos (limiar 2)
LCOM3	Lack Of Cohesion Of Methods 3	Falta de coesão de métodos (limiar 3)
MethodEffectiveLinesOfCode	Method Effective Lines Of Code	Número de statements/expressões de um método
NumberOfFinallyStatements	Number Of Finally Statements	Número de statements finally
NumberOfCatchStatements	Number Of Catch Statements	Número de statements catch
ExceptionalLOC	ExceptionalLOC	Quantidade de linhas de código de tratamento de Exceptions
NumberOfDummyExceptionHandlers	Number Of Dummy Exception Handlers	Quantidade de linhas de código de tratamento de Exception Handlers
NumberOfThrowStatements	Number Of Throw Statements	Número de statements throw
NumberOfTryStatements	Number Of Try Statements	Número de statements try
NumberOfTryStatementsWithNoCatchAndFir	Number Of Try Statements With No Catch And Finally	Número de statements sem Catch e com Finally
ThrownExceptionTypesCount	Thrown Exception Types Count	Contagem de lançamento de tipos de Exception

4

Resultados e Discussões

Esse capitulo apresenta os resultados desse trabalho de conclusão de curso. Como mostrado no capítulo anterior, o objetivo desse estudo é identificar as métricas de código que podem ser relevantes para a identificação de *Long Envious Methods*, então para isso é preciso entender se *Long Envious Methods* são frequentes em relação a métodos longos (nossa primeira questão de pesquisa) e quais métricas de código possuem diferença significativas entre métodos LEM e métodos LM (nossa segunda questão de pesquisa). As respostas das questões de pesquisas estão detalhadas nas seções a seguir.

4.1 QP_1: Frequência de Long Methods e possíveis Long Envious Method

Um estudo recente sugere que métodos longos geralmente são métodos que têm várias chamadas para classes externas, tendo assim a ocorrência simultânea de duas anomalias de código *Long Method* e *Feature Envy* (Bibiano et al., 2021). No entanto, não há estudos existentes que apresentem a frequência de métodos que possuem as duas anomalias de código simultaneamente. Como mencionado na Seção 3.2, foram criados programas para contar quantos métodos são *Long Methods* e quantos têm a ocorrência simultânea de *Long Method* e *Feature Envy* (Bibiano et al., 2021), considerando este como possível ocorrência de *Long Envious Method*.

As anomalias de código de 3 projetos de *software* (Activiti, Bytebuddy e Checkstyle) foram coletadas, após isso computou-se quais métodos tinham *Long Method* ao longo do histórico desses projetos. Depois, foi verificado se esses métodos tinham outras anomalias de código no mesmo *commit* que foi encontrado o *Long Method*, para assim certificar a simultaneidade. Para garantir que esses métodos são os mesmos, foram verificados a assinatura completa do método formada pelo nível de acessibilidade, nome e parâmetros do método. Com isso, foi feita a

separação dos métodos que só tinham *Long Methods* [1] dos métodos que são possíveis *Long Envious Methods* [2].

A quantidade de instâncias de *Long Methods* e *Long Envious Methods* para cada projeto foi tabulada e depois somamos a quantidade total de LM e LEM encontradas. A Tabela 4.1 apresenta assim a contagem dessas instâncias. 8.617 métodos que tinham essas anomalias de código foram coletados. Sendo que 4.707 (54.7%) métodos são *Long Methods* e 3.910 (45.3%) são possíveis *Long Envious Methods*. Através desse resultado já é possível indicar que *Long Envious Method* é uma anomalia de código frequente e que é comum que métodos com muitas linhas de código implementem mais de uma funcionalidade, principalmente quando são funcionalidades referentes a outras classes. Esse resultado sugere que desenvolvedores precisam de abordagens que ajudem a separar melhor as funcionalidades em métodos diferentes. Dessa forma, tem-se o nosso primeiro resultado sumarizado desse estudo.

> **Sumário 1: *Long Envious Method* é uma anomalia de código frequente (45.3%) em projetos de *software*.**

Activiti é o projeto que tem uma maior diferença entre *Long Envious Method* e *Long Method*, dos 2.450 métodos investigados, 1.665 (67%) métodos podem estar implementando múltiplas funcionalidades, podendo assim aumentar o processo de refatoração para a melhoria da qualidade do código, e como consequência dificultando a evolução do sistema.

Bytebuddy é um projeto onde a diferença entre *Long Envious Methods* e *Long Methods* é menor. Essa diferença é equivalente a 104 métodos (1.512 menos 1.408). Dessa forma, pode-se concluir que 1.512 (52%) dos 2.920 métodos são longos e implementam funcionalidades externas à classe do método, porém 1.408 (48%) têm múltiplas linhas de código que implementam funcionalidade inerentes a classe do método. Logo, consegue-se observar que cerca de 50% desse projeto está com problemas de modularização, pois a metade da quantidade de métodos está implementando mais de uma funcionalidade.

Checkstyle é um projeto diferente dos dois anteriores, nesse caso tem-se mais *Long Methods* que *Long Envious Methods*. Para esse projeto, 2.514 (77%) de 3.247 métodos tem somente muitas linhas de código. Então temos o nosso segundo resultado sumarizado e a resposta da nossa primeira questão de pesquisa, observando que nem todo *Long Method* é um *Long Envious Method*.

> **Sumário 2: A diferença entre a quantidade de *Long Envious Methods* e *Long Methods* pode variar por projeto.**

A partir desses resultados é possível observar que cada projeto tem características diferentes em termos de divisão de funcionalidades em métodos, essas características devem ser levadas em consideração na implementação de estratégias de detecção de anomalias de código.

[1] Esses métodos podem ter outros tipos de anomalias de código que não sejam *Feature Envy* simultaneamente.
[2] Esses métodos também podem ter outros tipos de anomalias de código

Tabela 4.1: Total de Long Methods e Long Envious Methods

Project	Long Method	Long Envious Method	Total de Métodos
Activiti	785	1665	2450
Bytebuddy	1408	1512	2920
Checkstyle	2514	733	3247
Total	**4707**	**3910**	**8617**

Pesquisadores e construtores de ferramentas de detecção de anomalias e de recomendação de refatoração devem observar porque a maioria dos métodos implementam mais de uma funcionalidade para alguns projetos, e para outros projetos os métodos são simplesmente longos. O entendimento da implementação desses métodos irão ajudar a melhorar a construção das estratégias de detecção para cada projeto e, como consequência, o melhoramento das recomendações de refatoração para remoção dessas anomalias, podendo assim aumentar a aceitação dos desenvolvedores para o uso dessas ferramentas.

4.2 QP_2: Análise Estatística de Métricas de Código de Long Methods e possíveis Long Envious Methods

4.2.1 Média e mediana geral

Ao olhar para as medidas de média e mediana levando em consideração os 3 projetos, na Tabela 4.2, percebe-se que há um aumento nas métricas *couplingIntensity*, *couplingDispersion* e *maxCallChain* referentes à anomalia de código *Long Envious Method* quando comparadas aos valores apresentados para o *Long Method*. A métrica *couplingIntensity* mede a intensidade do acoplamento, isso é calculado contando o número de métodos distintos chamados pelo método medido, quando seu valor é alto isso pode indicar que desenvolvedores degradaram a qualidade interna do projeto de *software* criando acoplamentos desnecessários. Já a *couplingDispersion* mede a dispersão de acoplamento, este é o número de classes em que as operações chamadas são definidas dividido pelo *couplingIntensity*.

4.2.2 Média e mediana por projeto

Observando as medidas de modo separado por projeto, nas Tabelas 4.3, 4.4 e 4.5, pode-se perceber que a média das métricas *couplingIntensity*, *couplingDispersion* e *maxCallChain* da anomalia *Long Envious Method* se mantém elevada. No caso dos projetos Bytebuddy e Checkstyle é possível notar um aumento considerável na média da métrica *cyclomaticComplexity*, ela refere-se ao número máximo de caminhos linearmente independentes em um método. Uma trajetória é linear se não houver ramo no fluxo de execução do código correspondente.

Por fim, pode-se considerar o aumento das métricas *couplingIntensity*, *couplingDispersion*,

Tabela 4.2: Média e Mediana Geral

Métrica	Long Method	Long Envious Method
Média		
MethodEffectiveLinesOfCode	66,320	50,413
NumberOfFinallyStatements	0,244	0,075
NumberOfCatchStatements	0,694	0,075
couplingIntensity	2,620	15,953
couplingDispersion	0,279	0,513
maxCallChain	1,882	3,917
numberOfAccessedVariables	24,257	15,386
Mediana		
couplingDispersion	0	0,513
couplingIntensity	0	13,000
cyclomaticComplexity	3,000	2,000

Tabela 4.3: Média e Mediana projeto Activiti

Métrica	Long Method	Long Envious Method
Média		
couplingDispersion	0,319	0,490
couplingIntensity	4,570	16,087
cyclomaticComplexity	10,575	8,570
maxCallChain	2,901	4,216
maxNesting	2,388	2,093
numberOfAccessedVariables	10,103	13,938
Mediana		
couplingDispersion	0,166	0,473
couplingIntensity	1,000	14,000
numberOfAccessedVariables	8,000	12,000

maxCallChain, *maxNesting* e *cyclomaticComplexity* como sendo interessantes para anomalias do tipo *Long Envious Methods*. Também é perceptível que métricas relacionadas a exceções têm um aumento significativo quando métodos são longos, mas não são *Long Envious Method*.

4.2.3 Teste F-Fisher

Como foi explicado no passo 5 da seção 3.2, o Teste Estatístico F-Fisher (ou Teste Exato de Fisher) foi executado para validar se a diferença entre as métricas de LM e LEM são significativas. Pois, somente a média e a mediana podem trazer algumas ameaças pelos valores da amostra, a partir desse teste estatístico é aceitável ter mais propriedade em dizer que essas diferenças significativas podem ser encontradas em outros projetos de *software*, aumentando assim a representatividade dos resultados encontrados.

O teste foi executado da seguinte forma, seja um tipo métrica de código M, tem-se que M_{LEM} é o valor da métrica de código para possíveis *Long Envious Methods*, e que M_{LM} é o

Tabela 4.4: Média e Mediana projeto Bytebuddy

Métrica	Long Method	Long Envious Method
Média		
couplingIntensity	3,207	19,689
maxCallChain	2,585	4,620
parameterCount	1,628	0,441
Mediana		
couplingIntensity	2,000	16,000
maxCallChain	2,000	4,000

Tabela 4.5: Média e Mediana projeto Checkstyle

Métrica	Long Method	Long Envious Method
Média		
numberOfAccessedVariables	35,579	23,549
maxCallChain	1,170	1,178
parameterCount	0,496	1,167
cyclomaticComplexity	6,000	14,000
couplingIntensity	1,682578	7,941337
Mediana		
cyclomaticComplexity	6,000	14,000
maxNesting	0,000	3,000
MethodEffectiveLinesOfCode	57,000	39,000
couplingIntensity	0,000	7,000000

valor da métrica de código para *Long Method*. Após isso foi executado o teste para cada par M_{LEM} e M_{LM}. Para cada par de métrica é gerado o valor das variáveis p e F. A variável p representa o nível de confiança que a diferença entre o par de métricas é significativa. Para a diferença entre o par de métricas ser significativa, p tem que ser menor ou igual a p_value. A constante $p\_value = 0.05$ foi o valor encontrado por Fisher que determina que a diferença entre 2 populações é significativa, representando assim que esse resultado tem a precisão maior ou igual 95% (0.95) (Fisher et al., 1963; Arcuri and Briand, 2011). Se $p <= p\_value$, então quer dizer que a diferença entre os pares de métricas é significativa e pode ser encontrada utilizando outros conjuntos de dados. Sendo assim, a variável F indica a probabilidade que essa diferença pode ser encontrada utilizando outras bases de dados, esse valor F é representado em número decimal, mas geralmente é convertido para porcentagem, sendo assim $F = 1.0$ interpretado como 100% de probabilidade da mesma diferença acontecer em outros conjuntos de dados.

Dessa forma, o teste de Fisher foi executado entre cada par de métrica por projeto e observaram-se os seguintes resultados. Tabela 4.6 apresenta as métricas que tiveram diferença significativa dos projetos Activiti e Checkstyle, respectivamente. Nota-se que para o projeto Activiti, as métricas *cyclomaticComplexity*, *maxNesting*, *MethodEffectiveLinesOfCode* e *NumberOfCatchStatements* têm diferenças significativas entre métodos *LEM* e *LM* e que a probabi-

Tabela 4.6: Principais Resultados do Teste Exato de Fisher

Métrica	**p: Nível de Confiança**	**F: Probabilidade**
Activiti		
cyclomaticComplexity	0,0001137	1,2623 (126%)
maxNesting	0,001747	0,82315 (82%)
MethodEffectiveLinesOfCode	0,002177	1,2037 (120%)
NumberOfCatchStatements	0,02179	1,1491 (115%)
Checkstyle		
couplingIntensity	0,004785	1,1864 (118%)

lidade dessa diferença acontecer em outros conjuntos de dados é 110%.

Como discutido na seção anterior 4.2.1, foi observado que essas métricas têm valores maiores frequentemente quando *Long Methods* são encontrados. Isso indica que métodos que são longos e que implementam funcionalidades somente inerentes à classe, tendem a ter uma complexidade maior (*cyclomaticComplexity*), uma maior aninhamento de chamadas de métodos (*maxNesting*), maior quantidade de linhas (*MethodEffectiveLinesOfCode*), e maior quantidade de tratamento de exceções (*NumberOfCatchStatements*). Esse resultado é surpreendente, pois esperava-se que essas métricas estariam mais degradadas quando *Long Envious Methods* são encontrados. A suposição para esse resultado é que desenvolvedores tendem a aumentar a complexidade e tamanho do método quando este implementa somente funcionalidades da classe de origem, visto que isso pode facilitar a manutenção do método. Mas esse resultado também indica que limiares inferiores para essas métricas podem ser usados para diferenciar *Long Methods* de *Long Envious Methods*, uma vez que métodos *Long Envious Methods* tendem a ter valores menores para essas métricas.

Para o projeto Checkstyle, percebe-se que a métrica *couplingIntensity* tem uma diferença significativa, principalmente porque *Long Envious Methods* tende a ter um intenso acoplamento comparado a *Long Methods*. Esse resultado confirma o que foi apresentado na seção 4.2.1, métodos *Long Envious Methods* tendem a ter um acoplamento muito maior que métodos só longos, principalmente no projeto Checkstyle. Essa diferença tem a probabilidade de 118% de acontecer em outros projetos de *software*. Surpreendentemente, apesar do Checkstyle ser o projeto que tem menos métodos *Long Envious Methods* (vide resultados da seção 4.1), esses métodos são bem mais acoplados que outros métodos. Esse resultado indica que esses métodos tendem a ter uma implementação de múltiplas funcionalidades de outras classes concentrados em um grupo significativo de métodos desse projeto.

> **Sumário 3: As métricas *cyclomaticComplexity*, *maxNesting*, *MethodEffectiveLinesOfCode* e *NumberOfCatchStatements* podem ser usadas para a identificação de *Long Envious Methods* com limiares inferiores.**

Como também foi observado, a métrica de código *couplingIntensity* tende a ter uma diferença muito maior para *Long Envious Methods* ao analisar todos os projetos (Tabela 4.2) e para

os projetos Bytebuddy e Checkstyle (Tabelas 4.4 e 4.5, respectivamente), então conclui-se que essa métrica de código pode ser usada para identificação de *Long Envious Methods*, principalmente com limiares superiores.

Sumário 4: A métrica *couplingIntensity* pode ser usada para a identificação de *Long Envious Method* com limiares superiores.

Não foi possível encontrar diferenças significativas ao analisar todos os projetos e nem para o projeto Bytebuddy, mas é possível acreditar que isso pode ter acontecido por algum erro durante a execução dos programas dos testes. Isso será verificado com cautela posteriormente.

5

Conclusão

Esse trabalho de conclusão de curso explorou quais métricas de código de métodos podem ser usadas para a identificação de *Long Envious Method*. Nós investigamos mais de 8.000 instâncias de anomalias de código em três projetos de *software* de diferentes domínios e dimensões. Foi (i) computado quantos métodos tem apenas *Long Method* e quantos métodos tem simultaneamente *Long Method* e *Feature Envy*, sendo estes possíveis *Long Envious Methods*; (ii) identificou-se quais métricas de código tem valores significativamente diferentes de métodos que são *Long Methods* e de métodos que são possíveis *Long Envious Methods*, para o cálculo dessa diferença significativa, foram calculadas a média, mediana e executado o teste estatístico F-Fisher (também conhecido como Teste Exato de Fisher).

Foram encontrados 4.707 (54%) *Long Methods* e 3.910 (45.3%) métodos que são *Long Envious Methods*. Isso indica que *Long Envious Methods* são frequentes e muitas vezes as ferramentas só indicam que esses métodos são somente longos, porém há mais problemas estruturais nesses métodos. Também pode-se observar que métricas *cyclomaticComplexity* (relacionada à complexidade do código), *NumberOfCatchStatements* (relacionada ao stratamento de exceções) e *couplingIntensity* (relacionados ao acoplamento do código fonte) têm uma diferença significativa entre *Long Envious Methods* e *Long Methods*. Logo, conclui-se que essas métricas podem ser usadas para a identificação de *Long Envious Methods*. Então, sugere-se que pesquisadores e construtores de ferramentas de detecção de anomalias devem observar porque a maioria dos métodos implementam mais de uma funcionalidade para alguns projetos, e para outros projetos os métodos são simplesmente longos. O entendimento da implementação desses métodos irão ajudar a melhorar a construção das estratégias de detecção para cada projeto e, como consequência, o melhoramento das técnicas de remoção dessas anomalias, podendo assim aumentar a aceitação dos desenvolvedores para o uso dessas ferramentas.

Referências

Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2019. **DOI** 10.1109/ESEM.2019.8870177.

Ethem Alpaydin. *Introduction to Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, 3 edition, 2014. ISBN 978-0-262-02818-9.

Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd international conference on software engineering*, pages 1–10, 2011.

Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports*, 6(4):110–138, 2016.

Ana Carla Bibiano. Catalog of complete composites, 2022. URL https://compositerefactoring.github.io/catalog.

Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. A quantitative study on characteristics and effect of batch refactoring on code smells. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2019.

Ana Carla Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Lucas Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, Márcio Ribeiro, Caio Barbosa, and Daniel Oliveira. How does incomplete composite refactoring affect internal quality attributes? In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, pages 149–159. Association for Computing Machinery, 2020.

Ana Carla Bibiano, Wesley K. G. Assunção, Daniel Coutinho, Kleber Santos, Vinícius Soares, Rohit Gheyi, Alessandro Garcia, Baldoino Fonseca, Márcio Ribeiro, Daniel Oliveira, Caio Barbosa, João Lucas Marques, and Anderson Oliveira. Look ahead! revealing complete composite refactorings and their smelliness effects. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 298–308, 2021. **DOI** 10.1109/ICSME52107.2021.00033.

Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 47–52. Association for Computing Machinery, 2010.

Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 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*, ESEC/FSE 2017, page 465–475. Association for Computing Machinery, 2017. ISBN 9781450351058. **DOI** 10.1145/3106237.3106259.

Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro F. Garcia. How does refactoring affect internal quality attributes?: A multi-project study. *Proceedings of the XXXI Brazilian Symposium on Software Engineering*, 2017.

José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. Code smells detection and visualization: A systematic literature review. *Archives of Computational Methods in Engineering*, 29:47–94, 2020.

Eduardo Fernandes, Gustavo Vale, Leonardo Sousa, Eduardo Figueiredo, Alessandro Garcia, and Jaejoon Lee. No code anomaly is an island: Anomaly agglomeration as sign of product line instabilities. In *16th International Conference on Software Reuse (ICSR)*, pages 48–64.

Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16. Association for Computing Machinery, 2016. ISBN 9781450336918.

Ronald Aylmer Fisher, Frank Yates, et al. *Statistical tables for biological, agricultural and medical research, edited by ra fisher and f. yates*. Edinburgh: Oliver and Boyd, 1963.

Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11:5: 1–38, 2012.

Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 609–613, 2016a. **DOI** 10.1109/SANER.2016.84.

Francesca Arcelli Fontana, Mika Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21:1143–1191, 2016b.

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1 edition, 1999.

Amir Hossein Ghapanchi, Claes Wohlin, and Aybüke Aurum. Resources contributing to gaining competitive advantage for open source software projects: An application of resource-based theory. *International Journal of Project Management*, 32:139–152, 2014.

Ivo Vieira Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo M. L. M. Moreira. An overview on the static code analysis approach in software development. 2009.

Clemente Izurieta, Ipek Ozkaya, Carolyn Budinger Seaman, Philippe B Kruchten, Robert L. Nord, Will Snipes, and Paris Avgeriou. Perspectives on managing technical debt: A transition point and roadmap from dagstuhl. In *QuASoQ/TDA@APSEC*, 2016.

Amandeep Kaur and Gaurav Dhiman. A review on search-based tools and techniques to identify bad code smells in object-oriented systems. *Harmony Search and Nature Inspired Optimization Algorithms*, 2018.

Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 113–122. Association for Computing Machinery, 2010. ISBN 9781450301169. **DOI** 10.1145/1858996.1859015. URL https://doi.org/10.1145/1858996.1859015.

Aleksandar Kovačević, Jelena Slivka, Dragan Vidaković, Katarina-Glorija Grujić, Nikola Luburić, Simona Prokić, and Goran Sladić. Automatic detection of long method and god class code smells through neural source code embeddings. *Expert Syst. Appl.*, 204(C), 2022. ISSN 0957-4174. **DOI** 10.1016/j.eswa.2022.117607. URL https://doi.org/10.1016/j.eswa.2022.117607.

Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, 2020.

M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, 1985.

Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 277–286.

Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, AOSD '12, page 167–178. Association for Computing Machinery, 2012.

Isela Macia, Alessandro Garcia, Christina Chavez, and Arndt von Staa. Enhancing the detection of code anomalies with architecture-sensitive strategies. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 177–186, 2013. **DOI** 10.1109/CSMR.2013.27.

Mika Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006. ISSN 1382-3256.

M.V. Mäntylä, J. Vanhanen, and C. Lassenius. Bad smells - humans as code critics. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 399–408, 2004. **DOI** 10.1109/ICSM.2004.1357825.

R. Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*, pages 173–182, 2001. **DOI** 10.1109/TOOLS.2001.941671.

R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359, 2004. **DOI** 10.1109/ICSM.2004.1357820.

Rafael de Mello, Anderson Uchôa, Roberto Oliveira, Willian Oizumi, Jairo Souza, Kleyson Mendes, Daniel Oliveira, Baldoino Fonseca, and Alessandro Garcia. Do research and practice of code smell identification walk together? a social representations analysis. In

2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pages 1–6, 2019.

Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010. **DOI** 10.1109/TSE.2009.50.

Emerson Murphy-Hill and Andrew P. Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, page 5–14. Association for Computing Machinery, 2010. ISBN 9781450300285.

Willian Oizumi, Alessandro Garcia, Leonardo Da Silva Sousa, Bruno Cafeo, and Yixue Zhao. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 440–451, 2016.

Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio SantAnna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 2017.

Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278, 2013. **DOI** 10.1109/ASE.2013.6693086.

Ghulam Rasool and Zeeshan Arshad. A review of code smell mining techniques. *J. Softw. Evol. Process*, 27(11):867–895, 2015. ISSN 2047-7473. **DOI** 10.1002/smr.1737. URL https://doi.org/10.1002/smr.1737.

Forrest Shull, Davide Falessi, Carolyn Seaman, Madeline Diep, and Lucas Layman. Technical debt: Showing the way for better transfer of empirical results. In *Perspectives on the Future of Software Engineering*, pages 179–190.

Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858—870. Association for Computing Machinery, 2016.

Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. *SIGPLAN Not.*, 34(10):47–56, 1999. ISSN 0362-1340.

Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331, 2008.

Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenilio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. How does modern code review impact software design degradation? an in-depth empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 511–522, 2020.

E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106, 2002. **DOI** 10.1109/WCRE.2002.1173068.

Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: A review of current knowledge. *J. Softw. Maint. Evol.*, 23(3):179–202, 2011. ISSN 1532-060X. **DOI** 10.1002/smr.521.