UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO (PCOMP)

MESTRADO ACADÊMICO EM COMPUTAÇÃO

JÚLIO SERAFIM MARTINS

INVESTIGANDO O IMPACTO DAS COOCORRÊNCIAS DE *CODE SMELLS* NOS ATRIBUTOS INTERNOS DE QUALIDADE

QUIXADÁ

2021

JÚLIO SERAFIM MARTINS

INVESTIGANDO O IMPACTO DAS COOCORRÊNCIAS DE *CODE SMELLS* NOS ATRIBUTOS INTERNOS DE QUALIDADE

Dissertação apresentada ao Curso de Mestrado Acadêmico em Computação do Programa de Pós-Graduação em Computação (PCOMP) do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Orientadora: Profa. Dra. Carla Ilane Moreira Bezerra

QUIXADÁ

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M343i Martins, Júlio Serafim.
Investigando o impacto das coocorrências de code smells nos atributos internos de qualidade / Júlio Serafim Martins. – 2021.
118 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Campus de Quixadá, Programa de Pós-Graduação em Computação, Quixadá, 2021.
Orientação: Profa. Dra. Carla Ilane Moreira Bezerra.

1. Code Smell. 2. Software - Refatoração. 3. Software - Controle de qualidade. I. Título.
CDD 005

JÚLIO SERAFIM MARTINS

INVESTIGANDO O IMPACTO DAS COOCORRÊNCIAS DE *CODE SMELLS* NOS ATRIBUTOS INTERNOS DE QUALIDADE

Dissertação apresentada ao Curso de Mestrado Acadêmico em Computação do Programa de Pós-Graduação em Computação (PCOMP) do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Aprovada em: ___ /___/____

BANCA EXAMINADORA

Profa. Dra. Carla Ilane Moreira Bezerra (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Dr. Alessandro Fabrício Garcia
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

Prof. Dr. Lincoln Souza Rocha
Universidade Federal do Ceará (UFC)

Prof. Dr. Régis Pires Magalhães
Universidade Federal do Ceará (UFC)

À minha família, por sua capacidade de acreditar em mim e investir em mim. Mãe, seu cuidado e dedicação foi que deram, em alguns momentos, a esperança para seguir. Pai, sua presença significou segurança e certeza de que não estou sozinho nessa caminhada.

AGRADECIMENTOS

Gostaria de agradecer primeiramente a Deus que é incrível de todas as formas e sabe de todas as coisas, sempre será a vontade dele e não a minha! Obrigado Jesus por ser um companheiro que nunca me abandonou e obrigado minha mãezinha Maria Santíssima por tudo.

Aos meus pais, Raimunda Neuma e Francisco Juscelino por ter me dado toda a educação, amor e suporte necessário. Vocês são meus ídolos. Com minha mãe eu aprendi a ser forte e a estudar para mudar de vida e com meu pai eu aprendi a ser uma pessoa mais humilde e calma. Eu amo vocês! Obrigado por sempre dar o melhor para mim e para o meu irmão Jonas.

À todos meus amigos e a Clara Taely, por me apoiar nos momentos mais difíceis e me dar forças para continuar. Vocês são importantes demais pra mim.

Também gostaria de agradecer a todos os meus professores do Colégio Santa Clara e a todos os meus professores do IFCE - Campus Canindé.

À professora Carla Ilane pela paciência (e puxões de orelha também), cuidado e dedicação na execução desse trabalho. Chega ao fim um ciclo que começou lá em 2016 no "TCC 0", continuou pelo TCC 1 e TCC 2 e também no mestrado com a finalização desse trabalho. Foi um prazer enorme ter você como minha orientadora e com certeza nunca vou esquecer, meu muito obrigado! Também gostaria de agradecer ao Anderson Uchôa por ter nos ajudado bastante durante toda a pesquisa, com excelentes contribuições.

À Universidade Federal do Ceará - Campus Quixadá por ter me formado, por ser o lugar do meu primeiro emprego e mostrar que ser professor é minha verdadeira paixão, além de ser o lugar que eu pretendo obter o título de mestre. Obrigado a todos os professores e professoras desse lugar maravilhoso.

Aos professores da banca, Alessandro Garcia que é uma referência e nos ajudou com dicas valiosas. Lincoln Rocha que é um excelente pesquisador e também contribuiu nesse trabalho. Ao professor Régis Pires, que por eu o conhecer melhor, sei que além de ser um excelente profissional é uma pessoa maravilhosa e me ajudou em momentos cruciais.

Por fim, eu gostaria de agradecer a mim mesmo por nunca ter desistido, por saber de todo o meu potencial e por ter sido resiliente nos momentos mais difíceis.

“O choro pode durar uma noite, mas a alegria vem pela manhã”

(Salmos 30:5)

RESUMO

Qualidade de software é um fator crítico e essencial em diferentes tipos de organizações. Assim, os desenvolvedores devem se preocupar com anomalias conhecidas como *code smells*. *Code smells* são estruturas que podem trazer prejuízos para a qualidade e manutenibilidade do código. No entanto, trabalhos recentes da literatura vêm estudando e mostrando que levar em consideração apenas ocorrências individuais de *smells* pode não ser suficiente para avaliar o impacto real que essas anomalias podem trazer para os sistemas. Nesse contexto, surgem as coocorrências de *code smells*, que são ocorrências de mais de um *code smell* em uma mesma classe ou mesmo método. As coocorrências dessas anomalias podem ser melhores indicadores de problemas para a qualidade de software. Revisões sistemáticas da literatura apontam que a área de coocorrências de *code smells* merece mais atenção e que são necessários mais estudos empíricos para avaliar os efeitos que essas anomalias podem causar para atributos internos de qualidade. Grande parte dos estudos realizados na área de coocorrências de *code smells*, analisam o impacto dessas anomalias para a manutenibilidade, problemas de *design* e arquitetura de software. Outros fatores importantes mencionados pela literatura são que os trabalhos devem realizar mais estudos de refatoração em sistemas industriais e que devem considerar a percepção dos desenvolvedores durante esse processo. Dessa forma, o objetivo deste trabalho foi investigar o impacto de coocorrências de *code smells* para os atributos internos de qualidade, como coesão, acoplamento, complexidade, herança e tamanho e também para os desenvolvedores. Foram executados dois estudos em projetos industriais, e os principais resultados e contribuições desse trabalho, são: (i) as coocorrências *Feature Envy–God Class*, *Dispersed Coupling–God Class* e *God Class-Long Method* são extremamente prejudiciais para a qualidade de software e para os desenvolvedores; (ii) o número de coocorrências de *code smells* tende a aumentar durante o desenvolvimento do sistema; (iii) desenvolvedores têm mais dificuldade para entender códigos contendo coocorrências de *smells*; e, (iv) desenvolvedores ainda possuem inseguranças em relação a identificação e refatoração de *code smells* e suas coocorrências. A partir dos resultados deste trabalho, foi possível gerar um catálogo prático sobre a remoção das coocorrências de *code smells* mais prejudicais para os atributos interno de qualidade e também sob a perspectiva dos desenvolvedores.

Palavras-chave: Code Smells. Software - Refatoração. Software - Controle de qualidade.

ABSTRACT

Software quality is a critical and essential factor in different types of organizations. Thus, developers should be concerned about anomalies known as code smells. Code smells are structures that can harm code quality and maintainability. However, recent works in the literature have been studying and showing that taking into account only individual occurrences of smells may not be enough to assess the real impact that these anomalies can bring to systems. In this context, the co-occurrences of code smells arise, which are occurrences of more than one code smell in the same class or method. Co-occurrences of these anomalies can be better indicators of problems for software quality. Systematic reviews of the literature indicate that the area of co-occurrences of code smells deserves more attention and that more empirical studies are needed to assess the effects that these anomalies can have on internal quality attributes. A large part of the studies carried out in the area of code smells co-occurrences analyze the impact of these anomalies on maintainability, design problems and software architecture. Other important factors mentioned in the literature are that the works must carry out more studies of refactoring in industrial systems and that they must consider the perception of developers during this process. Thus, the objective of this work was to investigate the impact of code smells co-occurrences for the internal quality attributes, such as cohesion, coupling, complexity, inheritance and size, and also for the developers. Two studies were carried out on industrial projects. The main results and contributions of this work are: (i) the co-occurrences *Feature Envy–God Class*, *Dispersed Coupling– God Class* and *God Class-Long Method* are extremely detrimental to software quality and developers; (ii) the number of co-occurrences of *code smells* tends to increase during the development of the system; (iii) developers have more difficulty understanding code containing co-occurrences of *smells*; and, (iv) developers still have insecurities regarding the identification and refactoring of *code smells* and their co-occurrences. From the results of this work, it was possible to generate a practical catalog about the removal of the most harmful *code smells* co-occurrences for the internal quality attributes and also from the perspective of the developers.

Keywords: Code Smells. Software - Refactoring. Software - Quality control.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Neste capítulo são apresentadas a motivação e a contextualização desta pesquisa, a qual tem por finalidade investigar os efeitos das coocorrências de *code smells* nos atributos internos de qualidade de sistemas industriais. Na Seção 1.1 são discutidas a contextualização e a motivação desta dissertação. Na Seção 1.2, os objetivos e as questões de pesquisa são apresentados. Na Seção 1.3 é descrita a metodologia utilizada nesta pesquisa. Na Seção 1.4, são elencados os principais resultados e contribuições desta dissertação. Por fim, na Seção 1.5 detalha-se a estrutura da dissertação, com a organização dos capítulos restantes.

1.1 Motivação e contextualização do problema

O software está cada vez mais inserido no contexto de muitos produtos competitivos modernos. Portanto, a competitividade no desenvolvimento de software tem se tornado, cada vez mais, uma preocupação para empresas de todos os tamanhos e segmentos de mercado (LINDEN *et al.*, 2007). Com o crescimento e evolução da indústria do software surgiram problemas na qualidade de software relacionados ao aumento da complexidade e tamanho do software. Dessa forma, os desenvolvedores reconhecem a importância de uma boa qualidade de software, pois um dos motivos em torno do paradigma de Orientação a Objetos (OO) foi justamente possibilitar a construção de um código com alta qualidade e manutenibilidade. Além disso, um software com boa qualidade pode proporcionar menor esforço e custo em atividades de manutenção (KAUR; SINGH, 2019). A indústria do software reconhece a importância de uma alta qualidade de software, pois assim é possível proporcionar uma melhor robustez, confiabilidade e manutenibilidade (MALHOTRA; JAIN, 2019).

Al-Qutaish (2010) afirma que a qualidade de software é crítica e essencial nos diferentes tipos de organizações. Em alguns softwares, tais como, sistema de tempo-real e sistemas de controle, a baixa qualidade de software pode trazer risco à vida humana ou a perda financeira. O *Design* de sistemas OO permite maior flexibilidade em relação a outros sistemas escritos em linguagens procedurais, por meio de mecanismos como: herança, polimorfismo e encapsulamento (STROGGYLOS; SPINELLIS, 2007). (NISTALA *et al.*, 2019) realizaram um mapeamento na literatura sobre modelos de qualidade e os autores apontam que os relatórios mais recentes sobre a qualidade de software indicam uma maior necessidade da utilização de uma abordagem que priorize a qualidade do código durante o processo de desenvolvimento do

software a ser construído. Qualidade de software OO se refere a características não funcionais como usabilidade, reusabilidade, modificabilidade e facilidade de evolução (KAUR, 2019).

A qualidade de software pode ser medida através de diferentes atributos de qualidade que podem ser classificados em atributos internos de qualidade e atributos externos de qualidade (MALHOTRA; CHUG, 2016). Os atributos externos de qualidade são aqueles indicam a qualidade de uma classe baseado em fatores que utilizam somente os artefatos de software. Atributos internos de qualidade podem ser medidos utilizando apenas artefatos de software, como o próprio código fonte. Assim, realizar a medição de atributos internos de qualidade é muito fácil do que em atributos externos de qualidade. Dessa forma, são analisados neste trabalho apenas atributos internos de qualidade como coesão, acoplamento, herança e complexidade. Para isso, serão utilizadas métricas do código fonte, como por exemplo, a métrica CBO que mede o acoplamento entre objetos (MORASCA, 2009).

Ao longo da sua evolução, o software sistematicamente sofre alterações que podem levar à deterioração de sua estrutura de qualidade. Nesse contexto, surge o conceito de *code smells*, que são estruturas de código anômalas que representam sintomas que afetam a manutenibilidade de sistemas em diferentes níveis, como classes e métodos (LANZA; MARINESCU, 2007; FOWLER, 2018). Além disso, *code smells* podem indicar problemas relacionados a aspectos da qualidade do código, como legibilidade e modificabilidade, podendo causar problemas para os desenvolvedores em atividades na fase de manutenção do software (FOWLER, 2018). Segundo Fowler (2018), um exemplo de *code smell* seria o *Long Method* que é um *code smell* que indica que existe um método muito longo, complexo e com muitas variáveis. *Code smells* não são erros no código fonte, ou seja, eles não impedem do software funcionar corretamente. No entanto, essas anomalias são consideradas pontos fracos no *design* de um software que pode retardar o desenvolvimento ou aumentar a ameaça de falha ou erros no futuro. Em alguns casos, *code smells* são introduzidos no código fonte por meio de escolhas ruins de implementação ocasionada diversas vezes pela pressa dos desenvolvedores em entregarem as funcionalidades de um sistema (ABDELMOEZ *et al.*, 2014).

Alguns trabalhos têm avaliado não só as ocorrências individuais de *code smells*, mas sim as relações entre essas anomalias e o impacto que elas causam para a qualidade de software (YAMASHITA *et al.*, 2015; OIZUMI *et al.*, 2016; FERNANDES *et al.*, 2017; POLITOWSKI *et al.*, 2020). A presença de ocorrências individuais de *code smells* não afeta de forma significante a compreensão de software e nem a performance dos desenvolvedores. O oposto acontece

quando existe uma coocorrência dessas anomalias (ABBES *et al.*, 2011). Levar em consideração a análise de coocorrências de *code smells* em software pode implicar em alguns benefícios, como: uma melhor eficiência no processo de detecção de *code smells*, uma única operação de refatoração pode sugerir a remoção de várias instâncias de *code smells* de uma vez e o número de coocorrências de *smells* é bem menor do que o número de ocorrências individuais, ocasionando um menor esforço de refatoração (PIETRZAK; WALTER, 2006; YAMASHITA *et al.*, 2015; MARTINS *et al.*, 2019). Além disso, coocorrências de *code smells* podem causar problemas de manutenção de software ou intensificar os efeitos de ocorrências individuais dessas anomalias (KOKOL *et al.*, 2021). No entanto, o que se pode observar atualmente é que as definições na literatura sobre coocorrências de *code smells* podem ser diferentes das utilizadas pelos desenvolvedores durante suas atividades diárias (SINGJAI *et al.*, 2021).

Pietrzak e Walter (2006) foram os primeiros a investigar a relação entre *code smells*, que eles chamam de relações *Inter-smell*. Os autores definem seis tipos de coocorrência de *code smells* e afirmam que o estudo dessas associações e dependências entre *code smells* pode resultar em um melhor entendimento de potenciais problemas para a qualidade do software. Yamashita e Moonen (2013b) e Sjøberg *et al.* (2012) indicam que coocorrências de *smells* não são bons indicadores para uma boa manutenibilidade. Oizumi *et al.* (2014) analisaram as coocorrências de *code smells* para identificar problemas de arquitetura: os autores sugerem que coocorrências de *code smells* são indicadores significativamente melhores de problemas de código do que instâncias individuais dessas anomalias. Uma possível relação entre *Duplicated Code* e *Long Method* foi encontrada por Martins *et al.* (2019), os autores perceberam o que o número de ocorrências do *Long Method* pode aumentar o número de ocorrências do Duplicated Code. Paulo *et al.* (2018) conduziram uma revisão na literatura sobre *code smells* e identificaram que coocorrências de *code smells* podem causar problemas na manutenibilidade e problemas no *design* e que mais estudos empíricos são necessários para investigar o impacto das coocorrências dessas anomalias no código fonte.

Nesse contexto, a refatoração surge como um processo para melhorar a qualidade de sistemas através da aplicação de modificações no código, como uma forma de atingir os objetivos dos desenvolvedores (PAIXÃO *et al.*, 2020). Refatoração pode remover *code smells* e coocorrências de *code smells*, e isso pode ter um impacto direto na qualidade do código (YAMASHITA; MOONEN, 2013a). No entanto, a revisão sistemática realizada por Lacerda *et al.* (2020) indica que existem muitas oportunidades para usar a refatoração com o objetivo

de remover *code smells* e suas coocorrências. Dessa forma, um dos principais desafios de pesquisa é indicar qual ou quais estratégias de refatoração devem ser aplicadas para remoção das coocorrências.

Apesar do grande número de estudos que investigam os efeitos de ocorrências individuais de *code smells* (SANTOS *et al.*, 2018), existem poucos trabalhos que investigam os efeitos de coocorrências de *code smells*. Esse é um tópico que merece mais atenção com mais estudos empíricos para que seja feita uma análise do impacto dessas anomalias para a qualidade de software (PAULO *et al.*, 2018; LACERDA *et al.*, 2020). Outra grande lacuna apontada por revisões da literatura (KAUR; SINGH, 2019; KAUR; DHIMAN, 2019), é que a maior parte dos estudos que investiga os efeitos de *code smells* e suas coocorrências de *code smells* utilizam sistemas *open source* e que poucos trabalhos consideram sistemas industriais. Dessa forma, é importante que sejam executados mais estudos nessa área utilizando sistemas industriais, porque sistemas *open-source* são desenvolvidos com um estilo de gerenciamento diferente dos sistemas industriais e é interessante que os estudos acadêmicos sejam executados em um ambiente real de desenvolvimento (ABID *et al.*, 2020). Neste trabalho, são estudados sistemas industriais escritos em Java e também é levado em consideração o ambiente real de desenvolvimento bem, como a percepção dos desenvolvedores.

Outro problema apontado por revisões da literatura (PATE *et al.*, 2013; PAULO *et al.*, 2018; KAUR; SINGH, 2019), é que são necessários mais estudos empíricos na área de refatoração de *code smells* que levem em consideração a percepção dos desenvolvedores sobre a remoção dessas anomalias no código.

Como a área de coocorrências de *code smells* é uma área ainda muito recente, os estudos se concentram em:

- Investigar o impacto de coocorrências de *code smells* para a **manutenibilidade** (YAMASHITA; MOONEN, 2013b; FERNANDES *et al.*, 2017; MARTINS *et al.*, 2019).
- Investigar o impacto de coocorrências de *code smells* para a **arquitetura de software** (MACIA *et al.*, 2012b; FONTANA *et al.*, 2015).
- Investigar o impacto de coocorrências de *code smells* para **problemas de *design*** (OIZUMI *et al.*, 2016).
- Detectar padrões e enumerar as coocorrências de *code smells* **mais frequentes** (PALOMBA *et al.*, 2017; PALOMBA *et al.*, 2018; WALTER *et al.*, 2018).

No entanto, é necessário que sejam realizados mais estudos empíricos que façam

a investigação em sistemas industriais do impacto das coocorrências de *code smells* para os atributos internos de qualidade e para os próprios desenvolvedores, levando em consideração suas percepções (PAULO *et al.*, 2018).

1.2 Objetivos de pesquisa

Tendo estabelecido e comprovado a importância de estudar a influência das coocorrências de *code smells* para a qualidade de software no contexto de sistemas industriais OO. Este trabalho tem como objetivo principal investigar o impacto das coocorrências de *code smells* através da identificação de quais são as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade, como coesão, acoplamento, complexidade, herança, tamanho e também para os desenvolvedores. As principais questões de pesquisa bem suas subquestões dessa investigação são:

QP_1: Qual o impacto das coocorrências de *code smells* para os atributos internos de qualidade?

- ***$QP_{1.1}$:*** *Quais são as coocorrências mais frequentes de code smells em projetos industriais?*
- ***$QP_{1.2}$*** *Como a remoção dos code smells afeta os atributos internos de qualidade em projetos industriais?*
- ***$QP_{1.3}$:*** *Quais as coocorrências de code smells consideradas mais difíceis de remover do ponto de vista do desenvolvedor?*

QP_2: Quais os efeitos das coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade e para os desenvolvedores?

- ***$QP_{2.1}$:*** *Quais as coocorrências de code smells são mais prejudiciais para os atributos internos de qualidade?*
- ***$QP_{2.2}$:*** *Quais as coocorrências de code smells mais prejudiciais para os desenvolvedores?*
- ***$QP_{2.3}$:*** *Quais as principais dificuldades dos desenvolvedores durante o processo de remoção de coocorrências de code smells?*

Com isso, os objetivos específicos desta pesquisa consistem em:

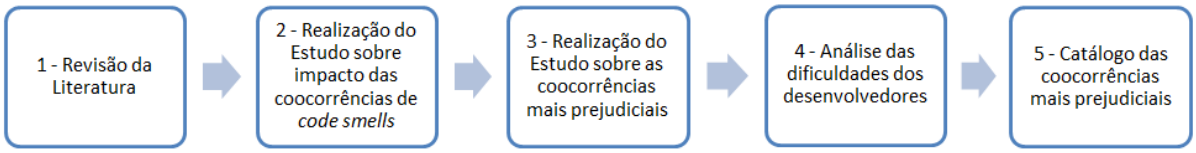
1. Identificar as coocorrências de *code smells* mais frequentes em projetos industriais.
2. Identificar as coocorrências de *code smells* mais difíceis de refatorar do ponto de vista do desenvolvedor.
3. Identificar as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade e para os desenvolvedores.

4. Identificar as principais dificuldades dos desenvolvedores durante as atividades de refatoração para remover coocorrências de *code smells*.

1.3 Metodologia

A Figura 1 apresenta a metodologia de pesquisa utilizada na construção deste trabalho. A justificativa e a descrição de cada um dos passos da metodologia são explicados abaixo.

Figura 1 – Metodologia de pesquisa



Fonte: Elaborado pelo Autor.

1. **Revisão da literatura:** Foi realizada uma revisão na literatura sobre *code smells*, coocorrências de *code smells*, retaforação de software e métricas e atributos internos de qualidade para identificar possíveis lacunas e deficiências na literatura.
2. **Realização do estudo sobre o impacto das coocorrências de *code smells*:** Uma vez que foram encontradas oportunidades de pesquisa, foi realizado um estudo inicial sobre coocorrências de *code smells*. Esse estudo é descrito no Capítulo 4, e teve como principal objetivo verificar o impacto das coocorrências de *code smells* para os atributos internos de qualidade. Neste passo, foi feita uma análise dos resultados e deficiências do estudo inicial que teve como objetivo principal verificar o impacto das coocorrências de *code smells*. Além disso, lições aprendidas foram armazenadas para a realização de um novo estudo para identificação das coocorrências mais prejudiciais.
3. **Realização do estudo sobre as coocorrências mais prejudiciais:** A partir dos resultados e deficiências encontradas no estudo anterior, foi possível realizar um novo estudo descrito no Capítulo 5. Com o novo estudo foi possível consolidar os resultados encontrados no estudo inicial e entender, sobre: (i) quais são as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade; (ii) quais as coocorrências mais prejudiciais para os desenvolvedores; e, (iii) quais foram as principais dificuldades dos

desenvolvedores no processo das remoção de coocorrências de *code smells*.

4. **Análise das dificuldades dos desenvolvedores:** A partir dos dois estudos realizados nos passos anteriores foi possível verificar as percepções dos desenvolvedores durante a remoção de *code smells* via refatoração. Assim, foi analisado as principais dificuldades dos desenvolvedores sobre a identificação e remoção de coocorrências de *code smells*.
5. **Catálogo das coocorrências mais prejudiciais:** Após a realização dos estudos, é possível ter um melhor entendimento empírico sobre as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade e para os desenvolvedores. Dessa forma, essa atividade visa catalogar os resultados dos estudos de forma a indicar um conjunto de coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade que devem ser removidas pelos desenvolvedores.

1.4 Principais Resultados e Contribuições

Os principais resultados e contribuições deste trabalho podem ser listados a seguir:

- É necessário priorizar a remoção das coocorrências *Feature Envy–God Class*, *Dispersed Coupling–God Class* e *God Class–Long Method*, pois elas são extremamente prejudiciais para os atributos internos de qualidade e para os desenvolvedores.
- O número de coocorrências de *code smells* tende a aumentar durante o desenvolvimento do sistema.
- Desenvolvedores têm mais dificuldade para entender códigos contendo coocorrências de *code smells*.
- Desenvolvedores ainda possuem inseguranças em relação a identificação e refatoração de *code smells* e suas coocorrências.
- A partir dos resultados deste trabalho e de outros estudos na literatura, foi possível gerar um catálogo prático com direcionamentos e recomendações sobre a remoção de coocorrências de *code smells*.

1.5 Organização

Este capítulo apresentou a motivação, os objetivos, as questões de pesquisa e a metodologia que foi utilizada para esta pesquisa. O restante deste documento está organizado da seguinte maneira:

No Capítulo 2 discute-se os conceitos relacionados a *code smells*, ferramentas para detecção de *code smells*, coocorrências de *code smells*, atributos internos de qualidade de software, refatoração de software, refatoração de *code smells* e refatoração e atributos de qualidade.

No Capítulo 3 são apresentados os trabalhos relacionados a este trabalho. Os trabalhos relacionados são catalogados, classificados, discutidos e comparados com a presente proposta de pesquisa.

O Capítulo 4 apresenta uma pesquisa inicial que verifica o impacto das coocorrências de *code smells* para os atributos internos de qualidade, quais as coocorrências mais difíceis de se refatorar, e quais foram as coocorrências mais frequentes em 3 sistemas industriais OO.

O Capítulo 5 aborda o segundo estudo desta pesquisa na qual foi possível investigar o impacto das coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade e para os desenvolvedores.

O Capítulo 6 oferece um catálogo prático com os principais direcionamentos e recomendações sobre remoção das coocorrências de *code smells*.

Por fim, o Capítulo 7 é dedicado às considerações finais e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo trata da fundamentação teórica utilizada neste trabalho. São abordados os conceitos que serão utilizados para alcançar os objetivos deste trabalho, como: *code smells* e ferramentas para detecção de *code smells* (Seção 2.1), coocorrências de *code smells* (Seção 2.2), atributos internos de qualidade (Seção 2.3) e refatoração de software (Seção 2.4).

2.1 *Code smells*

Code smells são anomalias que podem indicar problemas relacionados à aspectos da qualidade do código, e como consequência, os *code smells* podem causar problemas para os desenvolvedores nas atividades durante a fase de manutenção de software (FOWLER, 2018). Essas anomalias podem afetar qualquer tipo de sistema (MACIA *et al.*, 2012a; OIZUMI *et al.*, 2016). O impacto e os efeitos negativos que *code smells* podem ocasionar são diversos, como: diminuição da qualidade do *design* de software e impacto negativo nos atributos de qualidade (e.g., manutenibilidade, legibilidade e modificabilidade do código). Essa 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 *code smells* focam em sistemas orientado a objetos (OO) (KAUR; DHIMAN, 2019).

Diferente de um *bug* ou defeito no código fonte de um sistema, a presença de *code smells* 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.*, 2012a; OIZUMI *et al.*, 2016; MELLO *et al.*, 2019; UCHôA *et al.*, 2020) mostraram evidências de que *code smells* são fortes indicadores de partes do código afetadas por pobre decomposição de *features* em sistemas de software. Além disso, Palomba *et al.* (2019) concluíram que a presença de determinados *code smells* no contexto de aplicações Android pode significar um consumo elevado de energia e que a refatoração dessas anomalias mostrou uma redução da consumo de energia em todas as situações testada pelos autores.

Fowler (2018) define um catálogo de 22 tipos de *code smells* que afetam diferentes níveis do código fonte e que podem trazer problemas para a manutenibilidade do software. Outros autores definem outros tipos de *code smells* (LANZA; MARINESCU, 2007). No presente

trabalho, serão utilizados *code smells* definidos por Fowler (2018) e Lanza e Marinescu (2007) que apresentam diferentes níveis de granularidade. Estes foram escolhidos por serem os *code smells* que as ferramentas de detecção utilizadas neste trabalho suportam. A Tabela 1 apresenta os *code smells* e suas definições, bem como suas referências. Estão em negrito os *code smells* utilizados neste trabalho.

Tabela 1 – *Code smells* que as ferramentas suportam

Code Smells	**Descrição**
Data Class (FOWLER, 2018)	Classe que contém dados, mas nenhum comportamento relacionado aos dados.
Feature Envy (FOWLER, 2018)	Instruções de um método que deve ser movido para outro método, às vezes localizado em outra classe, cujas *features* são mais compartilhadas e usadas.
God Class (FOWLER, 2018)	Uma classe muito grande e complexa que geralmente concentra muitos recursos do sistema.
Long Method (FOWLER, 2018)	Método longo, complexo, com muitas variáveis e que centraliza a inteligência de uma classe.
Brain Method (LANZA; MARINESCU, 2007)	Método longo e complexo que centraliza a inteligência de uma classe.
Brain Class (LANZA; MARINESCU, 2007)	Classe complexa que acumula responsabilidade através dos *Brain Methods*.
Dispersed Coupling (FOWLER, 2018)	Método que chama um ou mais métodos de várias classes.
Intensive coupling (LANZA; MARINESCU, 2007)	Método que chama vários métodos de outras classes.
Refused parent bequest (FOWLER, 2018)	Subclasse que não usa os métodos protegidos de sua superclasse.
Shotgun surgery (FOWLER, 2018)	Método chamado por muitos métodos de outras classes.
Tradition breaker (LANZA; MARINESCU, 2007)	Subclasse que não especializa superclasse.
Duplicated Code (FOWLER, 2018)	Código duplicado em mais de um lugar no código fonte.
Type Checking (FOWLER, 2018)	Ao invés de separar um tipo de dado específico, existe um conjunto de números e *strings* que representam esse tipo de dado.

Fonte: Elaborado pelo Autor

Mantyla *et al.* (2003) apresentam uma taxonomia para *code smells*, baseada nos *code smells* definidos por Fowler (2018):

- ***Bloaters:*** representa qualquer elemento do código que se tornou muito grande e complexo de se gerenciar. Em geral, *bloaters* são difíceis de se entender e modificar. Alguns *code smells* dessa categoria são *Long Method* e *Large/God Class*.
- ***Object-Orientation Abusers:*** soluções alternativas e elementos que não são criados com as boas práticas de orientação a objetos. Um exemplo de *code smell* dessa categoria é *Refuse Parent Bequest*.

- ***The Change Preventers:*** tornam estruturas de software muito difíceis de serem modificadas, pois modificações podem levar a diferentes erros em vários lugares. Nessa categoria pode ser mencionado o *code smell Shotgun Surgery*.
- ***The Dispensables:*** *code smells* que não são necessários e que contém elementos surpléfuos que prejudicam a boa estrutura do código. Um exemplo de *code smell* dessa categoria é o *Duplicated Code*.
- ***The Couplers:*** *smells* que caracterizam um alto acoplamento. Um exemplo de *code smell* dessa categoria é o *Feature Envy*, pois essa anomalia representa alto acoplamento de elementos do código fonte como métodos e classes.

Existem diversas maneiras para realizar a detecção dessas anomalias, que vão desde a percepção humana até métricas, estratégias baseadas em regras, métodos baseados em pesquisa e visualização de software. Na prática, ferramentas também são construídas para suportar tais mecanismos de detecção (LACERDA *et al.*, 2020). A próxima Seção irá descrever as ferramentas que serão utilizadas nesse trabalho para detecção dos *code smells*.

2.1.1 Ferramentas para detecção de code smells

Atualmente existem diversas ferramentas que realizam a detecção de *code smells* (FERNANDES *et al.*, 2016). Kaur e Dhiman (2019) realizaram uma revisão na literatura para identificar as principais técnicas e ferramentas que são utilizadas para fazer a detecção de *code smells*. Os autores encontraram que a maioria dos trabalhos na literatura utilizam ferramentas automáticas que implementam métricas para realizar a detecção de *code smells*. Atualmente existem cerca de 82 ferramentas e técnicas que lidam com a detecção de *code smells* (PAULO *et al.*, 2018), algumas das ferramentas mais utilizadas são apresentadas a seguir:

- **DECOR**[1]: é uma ferramenta que permite a especificação e detecção de *code smells* e anti-padrões usando um vocabulário unificado e uma linguagem dedicada. Os mecanismos de detecção dessa ferramenta são amplamente conhecidos na literatura.
- **InFusion/IPlasma**[2]: são ferramentas desenvolvidas pela mesma empresa e apesar de utilizar os mesmos mecanismos de detecção de *code smells*, a ferramenta InFusion é proprietária e possui mais funcionalidades do que a ferramenta Iplasma que é *open source*.
- **Borland Together**[3]: é uma ferramenta proprietária que permite identificar vários *code*

[1] http://www.ptidej.net/research/designsmells/
[2] http://loose.upt.ro/iplasma
[3] http://www.borland.com/us/products/together

smells como *Data Class*, *Feature Envy* e *Shotgun Surgery*.

- **PMD**[4]: ferramenta *open source*, que possui comunidade ativa e pode ser instalada como um *plugin* na IDE Eclipse.
- **SonarQube**[5]: uma das ferramentas mais utilizadas por analistas de qualidade (TAIBI *et al.*, 2017). É uma ferramenta automática de revisão de código que detecta *bugs*, vulnerabilidades e *code smells* no código.

Além das ferramentas citadas acima, existem as ferramenta JDeodorant[6] (TSANTALIS *et al.*, 2008) e JSpIRIT[7] (VIDAL *et al.*, 2016), que são as duas ferramentas utilizadas neste trabalho. Essa ferramentas foram escolhidas pois, segundo o estudo de (PAIVA *et al.*, 2017), a ferramenta JDeodorant foi a que mais detectou a maioria das entidades afetadas por *code smells*, enquanto que a ferramenta JSpIRIT obteve uma maior precisão na detecção de *code smells* com um número menor de falsos positivos. Além disso, ambas as ferramentas possuem boa documentação *online*, são gratuitas e já foram utilizadas em diversos trabalhos na literatura (TSANTALIS *et al.*, 2008; LOZANO *et al.*, 2015; UCHÔA *et al.*, 2017; MARTINS *et al.*, 2019).

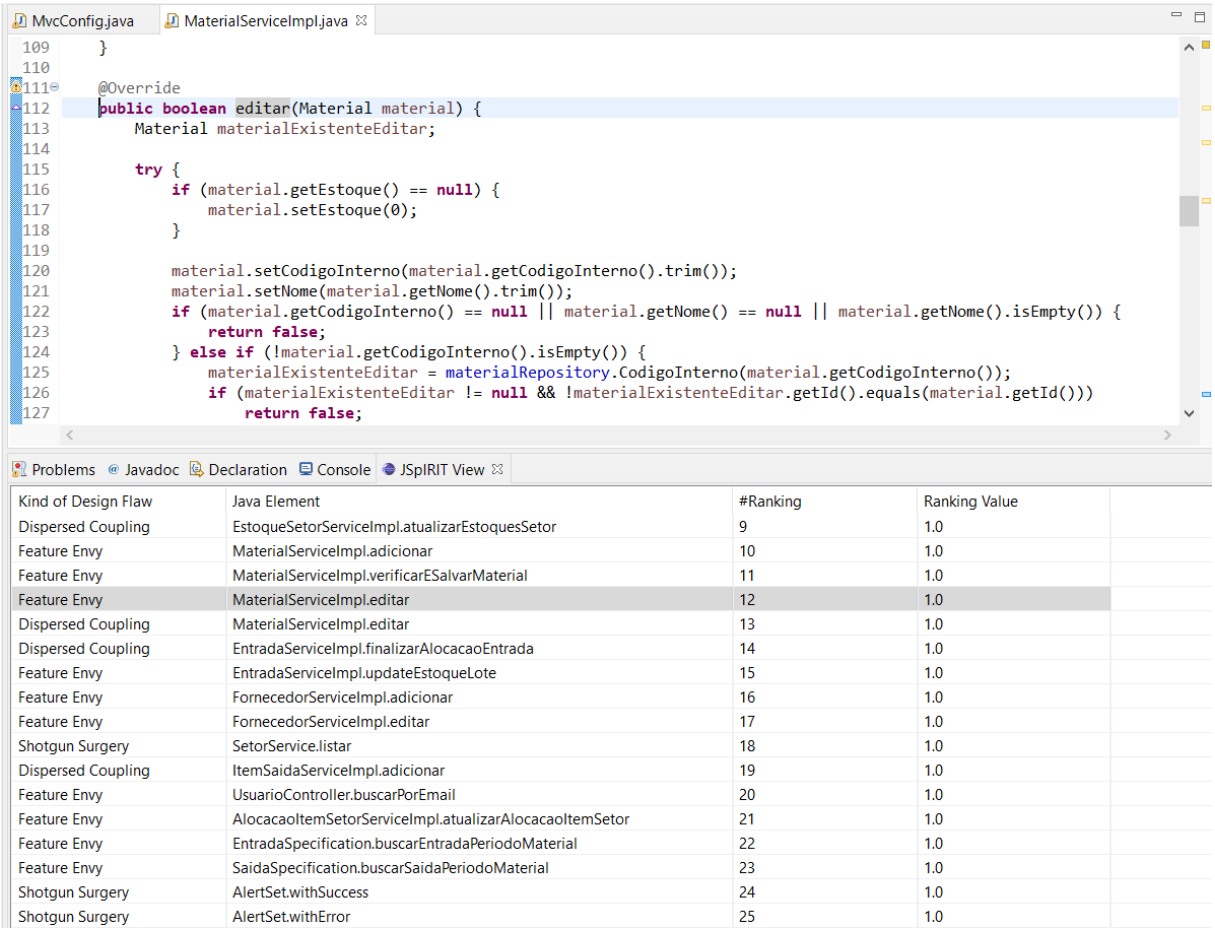
A ferramenta JSpIRIT, utiliza uma abordagem semiautomática que foca nos *code smells* mais críticos de um sistema (VIDAL *et al.*, 2015). Essa ferramenta consegue detectar 10 tipos de *code smells*, que são: *Brain Class* (BC), *Brain Method* (BM), *Data Class* (DCL), *Dispersed Coupling* (DCO), *Feature Envy* (FE), *God Class* (GC), *Intensive Coupling* (IC), *Refused Parent Bequest* (RPB), *Shotgun Surgery* (SS) e *Tradition Breaker* (TB) (VIDAL *et al.*, 2016). Neste trabalho a ferramenta JSpIRIT detecta os seguintes *code smells*: *Feature Envy*, *Dispersed Coupling*, *Intensive Coupling*, *Shotgun Surgery* e *Refused parent bequest*. JSpIRIT é uma ferramenta que suporta a identificação e a priorização dos *code smells*. O principal benefício dessa ferramenta é o fornecimento de uma funcionalidade que permite que os desenvolvedores configurem e adaptem a ferramenta para fornecer diferentes estratégias para identificar e ranquear os *smells* (VIDAL *et al.*, 2019). A ferramenta JSpIRIT pode ser utilizada para os seguintes propósitos (VIDAL *et al.*, 2015):

- **Mapeamento da informação:** É possível obter informação externa através do mapeamento do código. Um exemplo disso, é que o projeto arquitetural de uma aplicação pode ser mapeado para o código, e assim é possível indicar quais classes implementam determinado componente da arquitetura.

[4] https://pmd.github.io/
[5] https://www.sonarqube.org/
[6] https://github.com/tsantalis/JDeodorant
[7] https://sites.google.com/site/santiagoavidal/projects/jspirit

– **Detecção de *code smells:*** Todo o código da aplicação é revisado a fim de fazer identificação e apresentar os *code smells* existentes e em quais classes esses *smells* foram encontrados.
– **Priorização de *code smells:*** Os *code smells* são ranqueados de acordo com sua importância através de diferentes critérios de priorização. Um exemplo de critério de priorização pode ser o relacionamento de *code smells* e a modificabilidade do código fonte.

Figura 2 – Detectando *code smells* com a ferramenta JSpIRIT



Kind of Design Flaw	Java Element	#Ranking	Ranking Value
Dispersed Coupling	EstoqueSetorServiceImpl.atualizarEstoquesSetor	9	1.0
Feature Envy	MaterialServiceImpl.adicionar	10	1.0
Feature Envy	MaterialServiceImpl.verificarESalvarMaterial	11	1.0
Feature Envy	MaterialServiceImpl.editar	12	1.0
Dispersed Coupling	MaterialServiceImpl.editar	13	1.0
Dispersed Coupling	EntradaServiceImpl.finalizarAlocacaoEntrada	14	1.0
Feature Envy	EntradaServiceImpl.updateEstoqueLote	15	1.0
Feature Envy	FornecedorServiceImpl.adicionar	16	1.0
Feature Envy	FornecedorServiceImpl.editar	17	1.0
Shotgun Surgery	SetorService.listar	18	1.0
Dispersed Coupling	ItemSaidaServiceImpl.adicionar	19	1.0
Feature Envy	UsuarioController.buscarPorEmail	20	1.0
Feature Envy	AlocacaoItemSetorServiceImpl.atualizarAlocacaoItemSetor	21	1.0
Feature Envy	EntradaSpecification.buscarEntradaPeriodoMaterial	22	1.0
Feature Envy	SaidaSpecification.buscarSaidaPeriodoMaterial	23	1.0
Shotgun Surgery	AlertSet.withSuccess	24	1.0
Shotgun Surgery	AlertSet.withError	25	1.0

Fonte: Elaborado pelo Autor.

Como é possível visualizar na Figura 2, a ferramenta consegue realizar a detecção de *code smells* em projetos escritos na linguagem Java. Os *smells* detectados podem ser visualizados na parte inferior da Figura 2 a partir do menu *JSpIRIT View*.

Outra ferramenta utilizada neste trabalho é a JDeodorant (TSANTALIS *et al.*, 2008), que é um *plugin* da IDE Eclipse para detecção de *code smells*. JDeodorant teve seu código fonte aberto em 2014 e em junho de 2015 uma versão através de linha de comando foi lançada para permitir a execução da ferramenta sem precisar executar a IDE Eclipse. Além disso, atualmente cinco *code smells* são suportados por essa ferramenta: *Long Method*, *Feature Envy*, *God Class*,

Type/State Checking e *Duplicated Code* (TSANTALIS *et al.*, 2018). Neste trabalho, a ferramenta JDedorant detecta os seguintes *code smells*: *God Class* e *Long Method*.

Segundo Tsantalis *et al.* (2018), a ferramenta JDeodorant contribuiu na última década para realizar a detecção de *code smells* e tem sido utilizada por técnicas de recomendação de refatoração para diversos *code smells*. A ferramenta JDeodorant foi criada para fornecer uma solução holística e apoiar todas as atividades relacionadas à refatoração, como: (i) identificar onde o software deve ser refatorado; (ii) determinar quais técnicas de refatoração devem ser aplicadas nos locais identificados; (iii) garantir que após a refatoração o software continue funcionando normalmente; (iv) aplicar a refatoração; (v) avaliar o efeito da refatoração para atributos de qualidade do software como complexidade e manutenibilidade, além de avaliar atributos de processo como produtividade e custo; e, (vi) manter a consistência entre o código refatorado e outros artefatos de software como documentação, especificação de requisitos e testes (TSANTALIS *et al.*, 2018).

A Figura 3 mostra a detecção de *code smells* utilizando a ferramenta JDeodorant. Após a instalação da ferramenta é possível fazer a detecção de *code smells* através do menu *bad smells* que fica na parte superior da IDE Eclipse. A Figura 3 mostra a ocorrência do *code smell God Class* a partir do código marcado pela cor verde.

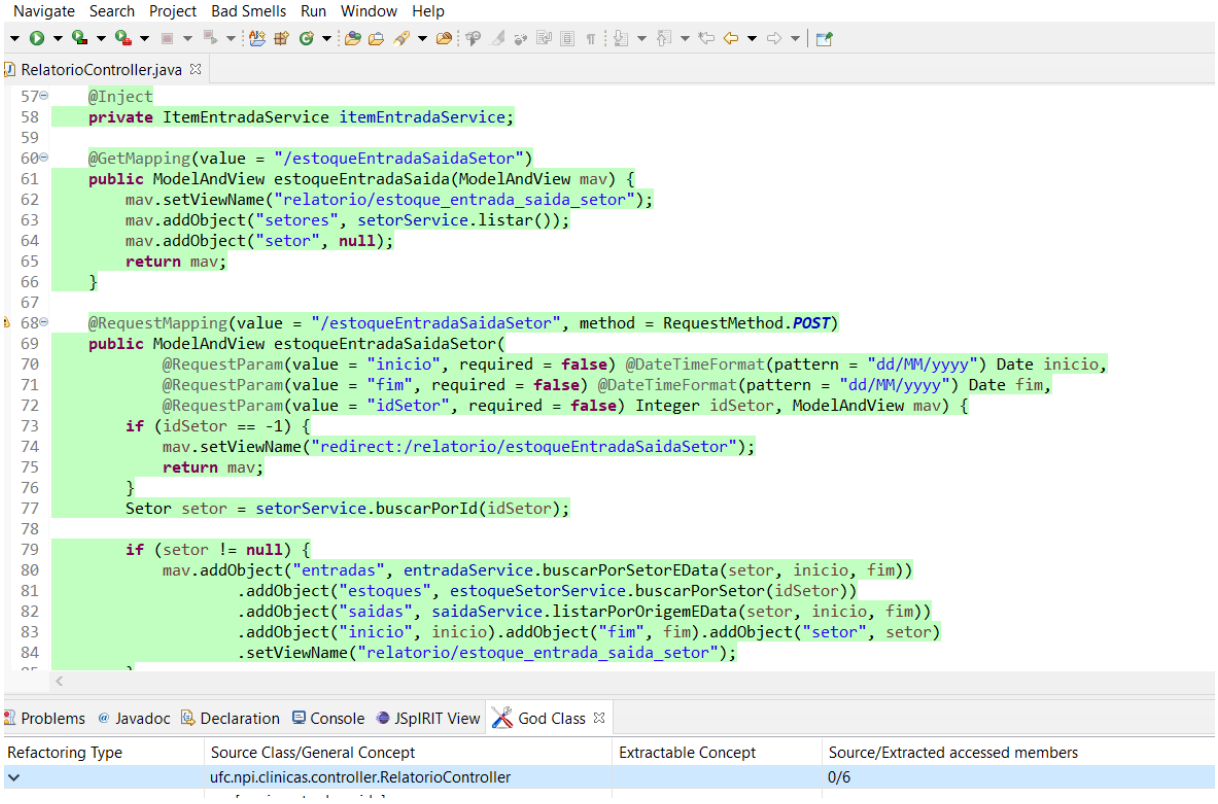
Antes de detectar as coocorrências, deve-se primeiramente fazer a detecção de ocorrências individuais de *code smells*. Isso é necessário para verificar e mapear as coocorrências dos *code smells*, que é a aparição de mais de um *code smell* em um mesmo método, ou a aparição dos *code smells* em uma mesma classe. Dessa forma, este trabalho vai verificar a coocorrência de *code smells* em sistemas Java OO utilizando as ferramenta JSpIRIT e JDeodorant.

2.2 Coocorrências de *code smells*

Coocorrências de *code smells* ocorrem quando existe um relacionamento e dependência entre dois ou mais *code smells*. Por exemplo, é possível que uma classe seja *God Class* e que possua o *Duplicated Code* (PIETRZAK; WALTER, 2006). Antes de detectar coocorrências de *code smells*, deve-se primeiro realizar a detecção de ocorrências individuais de *smells*. Isso é necessário para verificar e mapear as coocorrências de *code smells*.

Existem vários estudos na literatura que têm investigado somente ocorrências individuais de *code smells* (KAUR; SINGH, 2016; PALOMBA *et al.*, 2014; YAMASHITA; MOONEN, 2013a; TAHIR *et al.*, 2020; SANTANA *et al.*, 2021). No entanto, apenas alguns estudos (YA-

Figura 3 – Detectando *code smells* com a ferramenta JDeodorant



Fonte: Elaborado pelo Autor.

MASHITA *et al.*, 2015; MARTINS *et al.*, 2020; POLITOWSKI *et al.*, 2020; SINGJAI *et al.*, 2021; SANTANA *et al.*, 2021) abordam ou analisam coocorrências de *code smells*. Dessa forma, mais estudos empíricos que levam em consideração essas relações são necessários (YAMASHITA *et al.*, 2015). Pietrzak e Walter (2006) definem alguns tipos de coocorrências de *code smells* visando uma melhor precisão na verificação dessas anomalias e o impacto que elas podem causar nos sistemas.

Existem algumas nomenclaturas diferentes para coocorrências de *code smells* na literatura (WALTER *et al.*, 2018). Pietrzak e Walter (2006) foram os primeiros a investigar sobre relações entre *code smells* e suas dependências. Os autores categorizam 7 tipos diferentes de relações, que eles chamam de *inter-smell*, para fornecer um maior suporte na detecção dessas anomalias no código fonte. Além disso, prover um melhor entendimento do impacto e dos prejuízos que as interações de *code smells* podem causar para a qualidade de software. Lanza e Marinescu (2007) identificaram algumas relações entre *code smells* usando as palavras-chave: tem/usa. Por exemplo, *God Class* tem *Dispersed Coupling* (nível de classe) e *Intensive Coupling* usa *Shotgun Surgery* (nível de método). Oizumi *et al.* (2016) fazem uma investigação de como as relações entre *code smells* podem apoiar os desenvolvedores na localização de problemas

no *design* de software. Os autores propõem uma estratégia para grupos de coocorrências de *code smells* na qual os autores chamam de aglomerações. Yamashita *et al.* (2015) afirmam que as interações de *code smells* no mesmo arquivo (*collocated smells*) e que as interações de *code smells* em arquivos acoplados (*coupled smells*), mostraram trazer problemas em atividades de manutenção e para a qualidade do software. Fontana *et al.* (2015) estudaram diferentes relacionamentos estruturais entre 6 *code smells*, os relacionamentos são: chamando/é chamado, contém e usa. Dessa forma, os autores encontraram que *God Class* contém *Long Method* e *God Class* usa *Data Class*. O conceito de coocorrências de *code smells* utilizado nesse trabalho será o seguinte, uma vez que é realizada a detecção de ocorrências individuais de *code mells*, é possível então verificar as coocorrências de *code smells* a nível de método e a nível de classe (PALOMBA *et al.*, 2017). A nível de método uma coocorrência existe quando existem dois ou mais *code smells* em um determinado método. Por exemplo, existir em um mesmo método as ocorrências de *code smells* como *Feature Envy* e *Long Method*. Uma coocorrência a nível de classe acontece quando existe um *code smell* a nível de classe (por exemplo *God Class*) juntamente com outro *code smell* na mesma classe, como por exemplo, *Long Method* (FOWLER, 2018).

A Tabela 2 mostra exemplos de coocorrências de *code smells* a nível de classe e a nível de método. No primeiro exemplo é apresentado um exemplo de coocorrência a nível de classe (*Long Method e God Class*), ou seja, a Classe1 é uma *God Class* (GC) e possui o Método1 que é um *Long Method* (LM). No segundo exemplo, existe uma coocorrência a nível de método na qual dois *code smells Long Method* e *Feature Envy* (FE) estão "juntos" no Método2. Este exemplo mostra como é feita a identificação de coocorrências de *code smell* (PIETRZAK; WALTER, 2006) e essa identificação será feita no presente trabalho.

Tabela 2 – Exemplos de coocorrências de *code smells*

Classe	**Método**	**LM**	**FE**	**GC**
Classe1	método1()	X		X
Classe2	método2()	X	X	

Fonte: Elaborado pelo Autor

Existem poucos estudos na literatura que levam em consideração coocorrências de *code smells* (FONTANA *et al.*, 2015; LOZANO *et al.*, 2015; PALOMBA *et al.*, 2017), e os que existem não possuem fortes indícios empíricos. Assim é necessário realizar mais estudos empíricos do impacto de coocorrências de *code smells* (PAULO *et al.*, 2018; LACERDA *et al.*, 2020). Além disso, é utilizada apenas uma ferramenta para detecção de *code smells* nos

estudos (WALTER *et al.*, 2018). Nesse contexto, ocorrências individuais de *code smells* serão detectadas pelas duas ferramentas automáticas utilizadas neste trabalho e as coocorrências serão identificadas de forma manual. Com a remoção através de refatoração das coocorrências de *code smells* será possível entender e identificar quais são as coocorrências de *code smells* mais prejudiciais para atributos internos de qualidade e para os desenvolvedores.

2.3 Atributos internos de qualidade de software

Medição de software é essencial para garantir uma boa engenharia de software. Para saber se o *design* de software está em conformidade com os requisitos, se é de alta qualidade e se o código já está pronto para ser lançado, muitos desenvolvedores utilizam os resultados das métricas para tomada de decisões. Consequentemente, desenvolvedores devem ser capaz de avaliar a manutenibilidade de um produto para verificar o que pode ser melhorado (FENTON; BIEMAN, 2014).

A qualidade de software visa determinar como um *design* de software está concebido e o quão bom é o grau da estrutura do software. A qualidade de software pode ser medida por diferentes atributos de qualidade (MALHOTRA; CHUG, 2016). A ISO/IEC 25010/2011 (ISO, 2011), define a qualidade como o grau na qual um sistema satisfaz o que foi estabelecido e resolve as necessidades dos *stakeholders*.

De acordo com Malhotra e Chug (2016), atributos de qualidade podem ser classificados em: i) atributos internos de qualidade e ii) atributos externos de qualidade. Atributos externos de qualidade são aqueles que indicam a qualidade do sistema baseado em fatores que não podem ser mensurados somente utilizando artefatos de software. Por exemplo, a manutenibilidade que depende de um conjunto de fatores externos para que seja avaliada, como o tempo de vida do sistema e o ambiente para o qual o sistema está sendo modificado (DALLAL, 2013). Atributos internos de qualidade são indicadores chave para medir a qualidade da estrutura do código e assim, atributos internos de qualidade, tais como coesão e acoplamento, podem ser medidos com artefatos de software (MALHOTRA; CHUG, 2016). Quantificar atributos internos de qualidade é mais fácil do que quantificar atributos externos. Por exemplo, o tamanho de uma classe pode ser medida por meio da métrica LOC (MORASCA, 2009). Os atributos internos de qualidade utilizados neste trabalho podem ser definidos como (CHÁVEZ *et al.*, 2017):

– **Coesão:** é o grau no qual os elementos internos de um módulo estão relacionados entre si. Um baixa coesão pode ocasionar alta complexidade e *bugs* nos sistemas.

- **Acoplamento:** É o grau de interdependência entre classes e módulos de um sistema. Um alto acoplamento afeta a manutenibilidade do software.
- **Complexidade:** É a medida sobrecarga das responsabilidades e decisão de um módulo. Quanto mais alta complexidade maior é a chance de trazer problemas para a leitura e entendimento do código.
- **Herança:** representa o relacionamento entre subclasses e superclasses. Herança pode proporcionar reusabilidade do código. No entanto, um número elevado de heranças pode trazer problemas de manutenibilidade.
- **Tamanho:** representa o tamanho de um sistema a partir do número de linhas de código, classes, métodos e módulos.

Métricas de software exercem um papel fundamental para uma boa engenharia de software, pois é a partir de métricas e medições que é possível fazer uma boa avaliação da situação atual do software. Por exemplo, verificando a sua qualidade interna a partir de métricas referentes à atributos internos de qualidade (FENTON; BIEMAN, 2014). Para avaliar a qualidade dos sistemas investigados neste trabalho, serão utilizadas métricas bem conhecidas da literatura apresentadas na Tabela 3 (MCCABE, 1976; HENRY; KAFURA, 1981; NEJMEH, 1988; CHIDAMBER; KEMERER, 1994; LORENZ; KIDD, 1994; LANZA; MARINESCU, 2007). Essas métricas são suportadas pela ferramenta *Understand*[8], que será a ferramenta utilizada para coletar as métricas definidas na Tabela 3. A ferramenta *Understand* é um software proprietário e comercial desenvolvido pela empresa *SciTools*. Essa ferramenta faz a análise estática do código para verificar os valores das métricas referentes a cada atributo interno de qualidade (CHÁVEZ *et al.*, 2017).

As métricas propostas por Chidamber e Kemerer (1994), são pioneiras na área de métricas OO e apresentam uma base teórica para medir código OO. Neste trabalho a suíte *CK metrics* (CHIDAMBER; KEMERER, 1994) foi umas das escolhidas para ser utilizada na verificação da qualidade interna dos sistemas OO por já ter sido utilizada por diversos trabalhos (DYER *et al.*, 2012; MALHOTRA; CHUG, 2016; UCHÔA *et al.*, 2017) e pela ferramenta *Understand* para suportar a coleta das métricas.

Atributos internos de qualidade são geralmente mais fáceis de medir do que atributos externos. Dessa forma, utilizar atributos internos de qualidade ajudam a caracterizar a situação da estrutura e ter um melhor entendimento, pois a partir dessas medições é possível identificar

[8] https://scitools.com/

Tabela 3 – Métricas dos atributos internos de qualidade (MCCABE, 1976; CHIDAMBER; KEMERER, 1994; LORENZ; KIDD, 1994; DESTEFANIS *et al.*, 2014)

Atributos	**Métrica**	**Descrição**
Coesão	Lack of Cohesion of Methods (LCOM2) (CHIDAMBER; KEMERER, 1994)	Mede a coesão de uma classe. Quanto maior o valor dessa métrica, menos coesiva é a classe.
	Lack of Cohesion of Methods (LCOM3) (CHIDAMBER; KEMERER, 1994)	Número de componentes disjuntos no gráfico que representa cada método. Quanto maior o valor dessa métrica, menos coesiva é a classe.
Acoplamento	Coupling Between Objects (CBO) (CHIDAMBER; KEMERER, 1994)	Número de classes que uma classe está acoplada. Quanto maior o valor dessa métrica, maior o acoplamento das classes e métodos.
	Coupling Between Objects Modified (CBO Modified) (CHIDAMBER; KEMERER, 1994)	Número de classes que uma classe está acoplada. Quanto maior o valor dessa métrica, maior o acoplamento das classes e métodos.
	Fan-in (FANIN) (CHIDAMBER; KEMERER, 1994)	Número de outras classes que fazem referência a uma classe. Quanto maior o valor dessa métrica, maior o acoplamento das classes e métodos.
	(FANOUT) (CHIDAMBER; KEMERER, 1994)	Número de outras classes que fazem referência a uma classe. Quanto maior o valor dessa métrica, maior o acoplamento das classes e métodos.
Complexidade	Weighted Method Count (WMC) (MCCABE, 1976)	Soma da complexidade ciclomática de todas as funções aninhadas ou métodos. Quanto maior o valor da métrica, mais complexa são as classes e métodos.
	Average Cyclomatic Complexity (ACC) (MCCABE, 1976)	Média da complexidade ciclomática de todos os métdos aninhados. Quanto maior o valor da métrica, mais complexa são as classes e métodos.
	Sum Cyclomatic Complexity (SCC) (MCCABE, 1976)	Soma da complexidade ciclomática de todos os métodos aninhados. Quanto maior o valor da métrica, mais complexa são as classes e métodos.
	Nesting (MaxNest) (LORENZ; KIDD, 1994)	Nível máximo de aninhamento de construções de controle. Quanto maior o valor da métrica, mais complexa são as classes e métodos.
	Essential Complexity (EVG) (MCCABE, 1976)	Medida do grau em que um módulo contém construções não estruturadas. Quanto maior o valor desta métrica, mais complexas são as classes e métodos.
Herança	Number Of Children (NOC) (CHIDAMBER; KEMERER, 1994)	Número de subclasses de uma classe. Quanto maior o valor desta métrica maior é o grau de herança de um sistema.
	Depth of Inheritance Tree (DIT) (CHIDAMBER; KEMERER, 1994)	O número de níveis que uma subclasse herda de métodos e atributos de uma superclasse na árvore de herança. Quanto maior o valor dessa métrica, maior é o grau de herança de um sistema.
	Bases Classes (IFANIN) (DESTEFANIS *et al.*, 2014)	Número imediato de classes básicas. Quanto maior o valor desta métrica maior é o grau de herança de um sistema.
Tamanho	Lines of Code (LOC) (LORENZ; KIDD, 1994)	Número de linhas de código, excluindo espaços e comentários. Quanto maior o valor dessa métrica, maior será o tamanho do sistema.
	Lines with Comments (CLOC) (LORENZ; KIDD, 1994)	Número de linhas com comentário. Quanto maior o valor dessa métrica, maior será o tamanho do sistema.
	(CDL) (LORENZ; KIDD, 1994)	Número de classes. Quanto maior o valor dessa métrica, maior será o tamanho do sistema.
	Instance Methods (NIM) (LORENZ; KIDD, 1994)	Número de métodos de instância. Quanto maior for o valor desta métrica, maior será o tamanho do sistema.

Fonte: Elaborado pelo Autor

potenciais falhas em módulos, além de partes do software que poderão ser difíceis de manter ou testar (FENTON; BIEMAN, 2014). Nesse contexto, é realizada a medição da qualidade dos sistemas utilizando a ferramenta *Understand* nos sistemas antes da refatoração das coocorrências de *code smells* e é realizada novamente a medição nos sistemas refatorados a partir dos *commits de refatoração* para verificar quais as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade.

2.4 Refatoração de software

Refatoração é uma prática comum usada por desenvolvedores para eliminar *code smells* (STROGGYLOS; SPINELLIS, 2007; MURPHY-HILL *et al.*, 2012). A refatoração auxilia o código ser mais fácil de ser entendido e modificado através da eliminação de possíveis problemas e melhorando os atributos de qualidade. Além disso, refatoração também é usada para reengenharia, permitindo transformar o código legado de um sistema mais modular e organizado (LACERDA *et al.*, 2020). Refatorações podem ter finalidades diferentes. Por exemplo, *Move Method* é frequentemente aplicado por desenvolvedores para eliminar o *code smell Feature Envy*. Essa refatoração consiste em mover um método de uma classe para outra a fim de remover dependências excessivas entre as duas classes. Outro exemplo, é o *Extract Subclass* que pode ser aplicado visando eliminar o *code smell God Class*. O *God Class* é caracterizado por uma classe com tamanho e complexidade excessivos (FOWLER, 2018).

Existem outros tipos de refatoração com diferentes propósitos, tais como extrair novos elementos de código de elementos extremamentes complexos e gerenciar o grau de profundidade das heranças de classes (CHÁVEZ *et al.*, 2017). A prática de refatoração é uma importante ferramenta para melhorar continuamente a estrutura de um software, principalmente se o time de desenvolvimento é formado por desenvolvedores que se preocupam com a qualidade do código fonte que é escrito (LACERDA *et al.*, 2020). No entanto, de acordo com (PARNIN *et al.*, 2008), a atividade de refatoração costuma ser substituída devido ao seu alto custo e baixa prioridade quando comparado com outras atividades de desenvolvimento, como correção de *bugs* e implementação de novas funcionalidades.

Segundo Mens e Tourwé (2004), a prática de refatoração é considerada um dos meios mais importantes para transformar uma parte de software para melhorar a qualidade. O objetivo é diminuir a complexidade do projeto e do nível de código para diminuir o custo de manutenção e aumentar a produtividade dos desenvolvedores. Um processo básico de refatoração é ilustrado a seguir (MENS; TOURWÉ, 2004):

1. Detectar locais no código que necessitam de refatoração.
2. Determinar qual a refatoração que pode ser aplicada no código selecionado.
3. Garantir que após a refatoração o software preserve seu comportamento externo.
4. Aplicar a refatoração nos locais de código identificados.
5. Verificar o efeito da refatoração para características de qualidade de software.
6. Manter a consistência entre o artefato refatorado e outros artefatos de software.

Na prática os objetivos da refatoração podem variar e alguns desses objetivos inclui combater degradação do projeto de software, redução do esforço para realizar atividades de manutenção, facilitar a implementação de novas funcionalidades, corrigir *bugs* presentes no sistema e remover estruturas anômalas como *code smells* do código fonte (KIM *et al.*, 2012; BAVOTA *et al.*, 2015; SILVA *et al.*, 2016; MARIANI; VERGILIO, 2017). Três tipos de refatorações são empregados para atingir esses objetivos que são as refatorações *root-canal*, *floss* e *Self-Affirmed*. As definições dos tipos de refatoração são dados a seguir:

- Refatoração *root-canal*: Consiste em melhorar apenas a estrutura do código fonte (CEDRIM *et al.*, 2017).
- Refatoração *floss*: é o conceito de refatorar o código juntamente com mudanças não estruturais para atingir outros objetivos como adição de novas funcionalidades ou correção de *bugs* (CEDRIM *et al.*, 2017).
- refatoração *Self-Affirmed*: é o ato de documentar intencionalmente a atividade de refatoração. É constantemente usada em mensagens de *commits* (ALOMAR *et al.*, 2021).

Neste estudo são utilizadas as refatorações *root-canal*, pois o objetivo é melhorar a estrutura do código fonte a partir da refatoração de coocorrências de *code smells*, e também é utilizada a refatoração *Self-Affirmed* pois os desenvolvedores documentaram intencionalmente os seus *commits* de refatoração.

A refatoração consiste na mudança do código fonte de um sistema e pode melhorar a estrutura interna melhorando as medidas de atributos internos de qualidade tais como: coesão, acoplamento, complexidade, herança e tamanho (FOWLER, 2018). Com a ajuda da refatoração, projetos ruins de software podem ser convertidos em boas estruturas de código. Além disso, a refatoração pode aumentar significantemente também alguns aspectos de qualidade externos como reusabilidade, manutenibilidade e compreensibilidade (MALHOTRA; CHUG, 2016).

2.4.1 Refatoração e code smells

A refatoração de *code smells* pode melhorar a qualidade de software e assim reduzir o custo de manutenção associado ao software. Dessa forma, é importante que os desenvolvedores refatorem o código regularmente para manter a qualidade e o padrão de projeto de software a partir da remoção de anomalias como *code smells*. A refatoração pode ser feita de forma manual ou pode contar com o apoio de ferramentas (AGNIHOTRI; CHUG, 2020).

Fowler (2018) define um conjunto de métodos de refatoração para cada um dos 22

code smells. Alguns dos métodos de refatoração (FOWLER, 2018) mais utilizados para remoção dessas anomalias são (LACERDA *et al.*, 2020):

- ***Extract Class:*** esse método de refatoração é utilizado quando uma classe realiza mais operações do que o necessário. Utilizar a técnica *Extract Class* consiste em criar uma nova classe com atributos e métodos relevantes, assim um relacionamento unidirecional pode ser feito entre a antiga classe e a nova classe. Ao aplicar esse método de refatoração o código de uma classe deverá se tornar mais fácil de ser entendido e modificado.
- ***Extract Subclass:*** esse método de refatoração é utilizado quando uma classe possui métodos e atributos que são utilizados somente em alguns casos. Essa técnica de refatoração consiste em criar uma subclasse da classe antiga com métodos e atributos que fazem mais sentido estarem na subclasse. Ao aplicar esse método as classes vão se tornar mais coesas e existirá um ganho com reusabilidade.
- ***Extract Method:*** é utilizado quando um método contém muitas linhas e pode ser fragmentado em outros métodos. Essa técnica de refatoração consiste em criar um novo método e nomear esse método de acordo com seu propósito. O código do método antigo é copiado para o novo método e os campos dependentes são passados por parâmetro no novo método. Ao aplicar esse método de refatoração o código fonte deverá se tornar mais fácil de ser lido e o código duplicado diminuirá.
- ***Move Method:*** é utilizado quando um determinado método é mais utilizado por outra classe do que a sua própria classe. Essa técnica de refatoração consiste em remover o método da classe que menos o utiliza para colocá-lo na nova classe. Ao aplicar essa técnica de refatoração a coesão aumenta e dependência inter-classe diminui.
- ***Move Field:*** esse método de refatoração é utilizado quando um determinado atributo ou campo é utilizado mais em outra classe do que na sua própria classe. Essa técnica de refatoração consiste em remover o atributo/campo da classe que menos o utiliza para colocá-lo na nova classe. Ao aplicar essa técnica de refatoração as classes se tornaram mais coesas.
- ***Replace Temp with Query:*** esse método de refatoração é utilizado quando existe uma variável local que armazena o resultado de uma operação para uso posterior no código. Essa técnica consiste em mover a expressão inteira para um método separado que retorna o resultado da operação. Dessa forma, o método será consultado ao invés de armazenar o resultado em uma variável local. Ao aplicar essa técnica o código se tornará mais enxuto

por diminuir o tamanho do método, além de diminuir a possibilidade de código duplicado.

– ***Replace Inheritance with Delegation:*** esse método de refatoração é utilizado quando existe uma subclasse que usa somente uma pequena parte dos métodos de sua superclasse. Essa técnica de refatoração consiste em criar um atributo na subclasse para fazer referência ao objeto da superclasse para após isso, remover a herança. Ao aplicar essa técnica de refatoração será possível melhorar a estrutura do projeto do software. Código mais fino por meio de desduplicação, se a linha que está sendo substituída for usada em vários métodos.

Esse métodos de refatoração serão utilizados para remoção dos *code smells* estudados nesse trabalho. A Tabela 4 mostra o *code smell* e o método de refatoração que o remove (LACERDA *et al.*, 2020):

Tabela 4 – *Code smells* e métodos de refatoração associados

Code Smells	**Método de Refatoração**
Feature Envy (FOWLER, 2018)	*Move Method*, *Extract Method* e *Move Field*
God Class (FOWLER, 2018)	*Extract Class* e *Extract Subclass*
Long Method (FOWLER, 2018)	*Extract Method* e *Replace Temp with Query*
Dispersed Coupling (LANZA; MARINESCU, 2007)	Não existem métodos oficiais de refatoração desse *smell*
Intensive coupling (LANZA; MARINESCU, 2007)	Não existem métodos oficiais de refatoração desse *smell*
Refused Parent Bequest (FOWLER, 2018)	Replace Inheritance with Delegation
Shotgun surgery (FOWLER, 2018)	*Move Method* e *Move Field*

Fonte: Elaborado pelo Autor

Os métodos de refatoração acima serão utilizados neste trabalho como uma ferramenta para a remoção de *code smells* para ocasionar assim a eliminação das coocorrências de *smells*. Como pode ser visto na Tabela 4 os *code smells Dispersed Coupling* e *Intensive Coupling* não possuem métodos de refatoração "oficiais". Porém, existem *guidelines* que podem ser seguidas para remover ou evitar tanto o *Dispersed Coupling*[9] quanto para o *smell Intensive Coupling*[10].

Esses métodos foram escolhidos porque eles podem ser utilizados para remover todos os *code smells* que serão analisados neste trabalho e por serem os métodos mais utilizados para removerem esses *code smells* (LACERDA *et al.*, 2020).

[9] https://docs.embold.io/dispersed-coupling/

[10] https://docs.embold.io/intensive-coupling/

2.4.2 Refatoração e atributos de qualidade

Cada uma das operações de refatoração geralmente consiste na melhoria de vários atributos internos de qualidade (CHÁVEZ *et al.*, 2017). Atualmente existem duas abordagens principais para estudar o efeito da refatoração para a qualidade de software. A primeira abordagem consiste em identificar oportunidades de refatoração e técnicas de refatoração que serão utilizadas para refatorar *code smells* quando é aplicável e fazendo a comparação da qualidade do código antes e depois da refatoração. A segunda abordagem é analisar as mudanças implementadas no código durante a fase de manutenção detectando as mudanças devido a refatoração e fazendo a comparação da qualidade de software de antes e depois da refatoração concluída (DALLAL; ABDIN, 2017; SINGH; KAUR, 2018).

A aplicação do método *Extract Class* mostrou ter um potencial positivo em atributos como coesão e herança, no entanto, esse método pode significar um efeito negativo nos atributos complexidade e acoplamento. O método de refatoração *Extract Subclass* pode significar um impacto negativo para os atributos internos de qualidade complexidade, coesão e acoplamento porém melhora o atributo herança. A técnica de refatoração *Extract Method* pode trazer um efeito positivo para os atributos coesão, complexidade e na maioria dos casos não afeta a herança e nem o acoplamento. O método *Move Method* possui um potencial impacto positivo no atributo coesão e pode ocasionar impacto negativo para o acoplamento e complexidade. Por fim, *Move Field* ao ser aplicado pode ocasionar um impacto positivo para a coesão e um impacto negativo para o acoplamento (DALLAL; ABDIN, 2017). Nesse contexto, é possível afirmar que a refatoração não melhora todos os atributos de qualidade (LACERDA *et al.*, 2020).

Diferentes cenários de refatoração ocasionam diferentes efeitos nos atributos internos de qualidade. Por exemplo, alguns cenários de refatoração melhoram todos os atributos de qualidade, enquanto que outros cenários podem ocasionar uma combinação de efeitos como melhorar alguns atributos, piorar alguns atributos e até mesmo que alguns atributos permaneçam inalterados. Assim, é necessário uma maior demanda por estudos empíricos para explorar o impacto da refatoração nos atributos internos de qualidade (DALLAL; ABDIN, 2017).

O presente trabalho consiste em investigar quais são as coocorrências de *code smells* mais prejudiciais para atributos internos de qualidade através da análise de *commits* de refatoração da eliminação de coocorrências de *code smells*. Além disso, será verificado por meio das respostas dos questionários quais foram as coocorrências mais prejudiciais.

2.5 Percepção dos desenvolvedores sobre *code smells*

Yamashita e Moonen (2013a) realizaram um estudo exploratório com o objetivo de investigar o conhecimentos dos desenvolvedores sobre *code smells* através de perguntas perguntas para entender a familiaridade dos desenvolvedores sobre essas anomalias. Uma grande parte dos desenvolvedores não conhece o conceito de *code smells* enquanto os que conhecem, utilizam blogs técnicos, fóruns de programação, colegas de trabalho e seminários como principal fonte de informação sobre essas anomalias (YAMASHITA; MOONEN, 2013a; TAHIR *et al.*, 2018).

(PALOMBA *et al.*, 2014) conduziram um estudo com o objetivo de investigar a percepção dos desenvolvedores em relação aos *code smells*. Primeiro os autores identificaram e validaram 12 *code smells* em três projetos *open-source* e depois forneceram um questionário para os desenvolvedores onde foram apresentados exemplos de código fonte contendo *code smells* e exemplos de código sem essas anomalias. Os desenvolvedores foram perguntados se o código possuía algum problema, e em caso de resposta positiva os autores pediam para que fosse informado o tipo de problema e o motivo que levou aos participantes a identificarem o problema no código. As principais descobertas foram as seguintes (PALOMBA *et al.*, 2014):

- ***Alguns code smells geralmente não são percebidos como um problema:*** *Smells* como *Long Parameter List* e *Lazy Class* não são percebidos como "grandes ameaças", pois segundo os desenvolvedores, eles são mais relacionados a boas práticas de OO do que o código complexo. Este resultado também é confirmado por (TAIBI *et al.*, 2017).
- ***Code smells* relacionados a complexidade e tamanho longo são geralmente percebidos como uma grande ameaça pelos desenvolvedores:** Essa propriedade é valida para *smells* como *God Class* e *Long Method*. Esses resultados são confirmados por (MARTINS *et al.*, 2019; MARTINS *et al.*, 2021), na qual é encontrado que coocorrências de *code smells* contendo *God Class* são as mais prejudiciais na percepção dos desenvolvedores.
- **Experiência do desenvolvedor e conhecimento do sistema são importantes na identificação de *smells*:** É importante fazer a verificação da qualidade e análise de *code smells* para que os gerentes de projeto façam a alocação de desenvolvedores mais experientes se necessário. Isso confirma o que foi encontrado por (MARTINS *et al.*, 2021), na qual os autores chegaram a conclusão de que desenvolvedores inexperientes e com pouco conhecimento de *code smells* e métricas OO ainda possuem inseguranças durante a refatoração dessas anomalias.

Em várias discussões do *Stack Overflow* [11], os desenvolvedores usam o termo *code smell* de forma bem livre que não necessariamente se referem a definição conceitual do que é um *code smell*. Além disso, os desenvolvedores quando utilizam o termo *code smell* não estão falando de um *smell* específico e sim para expressar alguma aversão em relação ao código fonte (TAHIR *et al.*, 2018; TAHIR *et al.*, 2020). Outra importante observação feita por (TAHIR *et al.*, 2018) e confirmada por (YAMASHITA; MOONEN, 2013a) e (TAHIR *et al.*, 2020), é que desenvolvedores perguntam sobre quais são as ferramentas que auxiliam na detecção dessas anomalias, e mais do que isso, procuram por ajuda sobre como utilizar essas ferramentas durante suas atividades diárias.

Enquanto que o conceito de *code smells* tem se tornado mais conhecido nos últimos anos, a percepção dos desenvolvedores em relação aos prejuízos que essas anomalias podem causar ainda precisa ser melhor estudada (TAIBI *et al.*, 2017). A **percepção** dos desenvolvedores em relação aos ***code smells*** ainda é pouco explorada, e ainda, a percepção dos desenvolvedores durante atividades de remoção de *code smells* é raramente explorada. Dessa forma, são necessários mais estudos para abordar essa questão (KAUR; DHIMAN, 2019; KAUR, 2019; LACERDA *et al.*, 2020).

Este trabalho leva em consideração a percepção dos desenvolvedores durante as atividades de remoção das coocorrências de *code smells* com o objetivo de entender quais são as coocorrências mais prejudiciais para o código fonte.

2.6 Conclusão

Este capítulo apresentou os principais conceitos que serão utilizados por este trabalho. O objetivo deste trabalho é investigar quais são as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade e os desenvolvedores. Dessa forma, foram apresentados os *code smells* que serão detectados nos projetos e suas definições. Também foram apresentadas as duas ferramentas que serão utilizadas nos estudos para detecção dos *code smells*, JDeodorant e JSpIRIT. Foi descrita a forma como as coocorrências serão detectadas. Foram detalhados os atributos de qualidade que o trabalho irá investigar e as métricas relacionadas a esses atributos. Foram apresentados os métodos de refatoração que serão utilizados no trabalho para remoção dos *code smells*. Por fim, foram discutidas as percepções dos desenvolvedores sobre *code smells* de acordo com os estudos da literatura.

[11] https://stackoverflow.com/

3 TRABALHOS RELACIONADOS

Este capítulo apresenta os principais trabalhos relacionados a esta pesquisa. Foram agrupados os trabalhos relacionados em quatro tópicos: (i) revisões da literatura sobre *code smells* (Seção 3.1); (ii) detecção e análise das coocorrências de *code smells* (Seção 3.3); (iii) *code smells* e atributos internos de qualidade (Seção 3.2); e, (iv) coocorrências de *code smells* e atributos internos de qualidade (Seção 3.4). Na Seção 3.5 é realizada uma comparação dos trabalhos relacionados com a proposta do presente trabalho. Finalmente, na Seção 3.6 são apresentadas as conclusões do capítulo.

3.1 Revisões da literatura sobre *code smells*

Nos últimos os anos diversos estudos foram realizados para investigar *code smells*. Nesta Seção são apresentadas algumas revisões sistemáticas da literatura que analisaram os estudos primários publicados em *code smells* sob a perspectiva de diferentes problemáticas, como: (i) detecção de *code smells* (SINGH; KAUR, 2018; PAULO *et al.*, 2018; KAUR; DHIMAN, 2019); (ii) *code smells* e refatoração (SINGH; KAUR, 2018; KAUR; SINGH, 2019; ABID *et al.*, 2020; LACERDA *et al.*, 2020); e, (iii) *code smells* e atributos de qualidade (KAUR, 2019).

Paulo *et al.* (2018) conduziram uma revisão sistemática da literatura abordando 351 estudos primários para descrever o estado da arte no contexto de *code smells* entre 1990 e 2017. Os autores investigaram diversos aspectos incluindo interesse de pesquisadores na área de *smells*, diferentes grupos ou pessoas interessadas em *code smells*, diversidade de diferentes tipos de *code smells*, objetivos e insumos sobre essas anomalias. Como resultados, os autores encontraram que, apesar do *code smell Duplicated Code* ter sido estudado em quase 70% dos trabalhos, a coocorrência desse *smell* com outras anomalias é pouco explorado. Além disso, os autores também apontam que estudar coocorrências de *code smells* e suas interações em componentes de código se mostra uma área promissora e esse tipo de análise pode ajudar na localização de problemas para a manutenção de software. No entanto, até o momento poucos estudos empíricos nessa área têm sido realizados ocasionando pouca evidência do efeito dessas anomalias para a qualidade de software e por isso é necessário mais estudos sobre coocorrências de *code smells* e seus efeitos.

Singh e Kaur (2018) realizaram uma revisão na literatura analisando cerca de 238 artigos primários de conferências, *workshops* e periódicos. Os autores procuraram investigar

sobre questões relacionadas a refatoração de *code smells*, quais as principais abordagens para detecção e remoção de *code smells*, quais as principais ferramentas para identificar essas anomalias e quais os *smells* mais estudados nos trabalhos. Os resultados encontrados pelos autores mostram, que: (i) a partir de 2001 os estudos na área de refatoração começaram a surgir de forma massiva em relação a *code smells* e anti-padrões; (ii) a maioria dos estudos leva em consideração sistemas *open source* e que poucos autores têm estudado sistemas industriais; (iii) as estratégias de detecção mais utilizadas se baseiam em métricas, já que métricas são aspectos chave para mensurar atributos internos na qualidade de um sistema, utilizando abordagem de detecção automática e também semiautomática; (iv) diferentes ferramentas têm sido utilizadas para detecção e refatoração de *code smells*, como a JDeodorant por fornecer uma interface gráfica intuitiva; e, (v) os *code smells God Class* e *Blob* são os mais estudados, enquanto que *Refused Parent Bequest* é o *smell* que menos aparece nos estudos primários. Um dos problemas em aberto identificados pelos autores é que devem ser realizados mais estudos com sistemas *open source* e industriais diferentes dos que já são frequentemente estudados pelos trabalhos na literatura.

Kaur e Singh (2019) conduziram uma revisão sistemática da literatura analisando 142 estudos primários publicados até dezembro de 2017. Os autores realizaram uma investigação sobre questões relacionadas a refatoração de *code smells* e seu impacto para a qualidade de software, quais métricas e ferramentas têm sido utilizadas para avaliar o impacto das refatorações e o atual estado da arte sobre as atividades de refatoração e qualidade de software. Os resultados encontrados pelos autores apontam que: (i) os trabalhos analisados utilizam refatoração manual ou automática utilizando ferramentas; (ii) os métodos de refatoração mais utilizados são *Move Method*, *Extract Method* e *Extract Class*; (iii) os atributos internos de qualidade são mais explorados que atributos externos; (iv) os atributos internos mais estudados são coesão, acoplamento e complexidade; (v) as métricas de qualidade mais utilizadas são LOC (Número de linhas de código), NIM (Número de Métodos), LCOM2 (Falta de coesão de métodos) e COB (Acoplamento entre Objetos); (vi) existem poucas ferramentas que realizam a refatoração automática; (vii) os estudos de refatorações em ambientes acadêmicos tiveram um impacto mais positivo para a qualidade de software do que os estudos realizados na indústria; e, (viii) aplicar refatoração nem sempre significa melhorar todos os atributos de qualidade.

Kaur (2019) executou uma revisão na literatura para tentar entender a relação entre *code smells* e atributos de qualidade de software. Os autores na primeira busca conseguiram coletar 4363 estudos relevantes publicados até março de 2018, porém depois de aplicar o filtro a

partir da leitura do resumo, título e todo o artigo restaram 74 estudos primários. Eles procuraram investigar sobre questões relacionadas a *code smells* e seu impacto na qualidade de software, os principais *datasets* utilizados e suas características, atributos e métricas de qualidade para avaliar o impacto de *smells* para a qualidade. Os resultados encontrados mostram que: (i) os *code smells* mais estudados e seu impacto para a qualidade são *Duplicated Code*, *God Class*, *Long Parameter List*, *Long Method* e *Feature Envy*; (ii) a maioria dos *datasets* avaliados são *open-source* e escritos na linguagem Java; (iii) os atributos internos complexidade e coesão são os mais analisados pelos estudos primários; (iv) os *code smells God Class* e *Duplicated Code* foram os *smells* que mais demonstraram ter um impacto negativo para os atributos de qualidade. A partir dos resultados encontrados o autor sugere que sejam feitos mais estudos para validar o impacto de *code smells* para atributos de qualidade de software para se ter uma conclusão geral sobre os efeitos dessas anomalias.

Kaur e Dhiman (2019) fizeram uma revisão da literatura selecionando 32 estudos publicados até agosto de 2017. Os autores investigaram ferramentas e técnicas para identificar *code smells* em sistemas OO. Como resultados deste estudo, os autores encontraram, que: (i) existem muitas ferramentas para fazer a detecção de *code smells* e que as ferramentas JDeodorant, DECOR e iPlasma são as mais utilizadas nos trabalhos da literatura; (ii) é necessário o desenvolvimento de uma nova ferramenta para detectar e priorizar *code smells* de acordo com seu impacto para a qualidade de sofware; (iii) a maioria das ferramentas detecta *smells* em apenas uma linguagem e Java, por ser a linguagem mais comum; (iv) os *code smells Long Method*, *Duplicated Code*, *Feature Envy* e *Long Parameter List* têm mais atenção dos trabalhos; e, (v) a maioria dos estudos levam em consideração sistema *open source* e que poucos trabalhos consideram sistemas industriais.

Abid *et al.* (2020) realizaram uma revisão da literatura das últimas três décadas sobre refatoração. Os autores analisaram 3183 estudos que envolviam o conceito de refatoração publicados até maio de 2020. Os autores investigaram sobre questões envolvendo o ciclo de vida da refatoração, quais tipos de artefatos são refatorados, o motivo pelo qual os desenvolvedores e pesquisadores realizam refatoração e quais os tipos de *datasets* são utilizados para validar os estudos sobre refatoração. Os resultados encontrados mostraram que o ciclo de vida da refatoração é composto por 6 estágios que são: (i) detecção de refatoração, que é a identificação de oportunidades de refatoração; (ii) priorização de refatoração, que é priorizar determinadas refatorações baseadas em critérios; (iii) recomendação de refatoração, que utiliza ferramentas que

sugerem refatorações para os desenvolvedores; (iv) teste de refatoração, que busca garantir que o software ainda funciona corretamente; (v) documentação da refatoração, que visa documentar as refatorações; e, (vi) predição, que é localizar os locais que provavelmente deverão ser refatorados no futuro. Outros resultados encontrados pelos autores apontam que refatoração não é somente limitada ao código fonte, e que outros artefatos como arquitetura de software, esquema do banco de dados, modelos e interfaces de usuário, também podem ser refatoradas. A maioria das ferramentas apóiam a refatoração para sistemas escritos em Java. Outro resultado é que a maioria dos estudos sobre refatoração utilizam sistemas *open source* e que são necessários mais estudos empíricos em sistemas industriais.

Lacerda *et al.* (2020) realizaram uma revisão terciária e levam em consideração um conjunto de 40 estudos secundários publicados até 2018. Os autores investigaram sobre questões relacionadas aos principais tópicos de refatoração, quais os *code smells* mais analisados, quais as ferramentas para detectar *smells* e suportar refatoração e a relação entre *code smells* e refatoração. Os resultados encontrados pelos autores apontam que: (i) os *code smells* mais estudados pelos trabalhos são *God Class* e *Duplicated Code*; (ii) um tópico que merece atenção e mais estudos é o de coocorrências de *code smells* e seu impacto para a qualidade de software; (iii) os métodos de refatoração *Extract Class* e *Extract Method* são os mais analisados nos estudos; (iv) CCFinder foi a ferramenta mais citada para detecção de *code smells* e a ferramenta JDeodorant foi a mais utilizada para realizar refatorações; e, (v) atributos de qualidade afetados por *code smells* foram os mesmos afetados pelas atividades de refatoração. Alguns dos problemas em aberto identificados pelos autores são o desenvolvimento de mais ferramentas que apóiem refatorações e entender quais refatorações trazem um impacto mais significativo para a qualidade de software.

Singjai *et al.* (2021) realizaram um estudo *Grounded Theory* da literatura cinza sobre a percepção dos desenvolvedores sobre *coupling smells* que são *code smels* que caracterizam alto acoplamento. Os autores encontraram que *Feature Envy*, *Inappropriate Intimacy*, *Data Class*, *Message Chain* e *Middle Man* são os *coupling smells* mais discutidos pelos desenvolvedores. Além disso, foram identificadas que ferramentas de detecção de *coupling smells* devem levar mais em consideração o conhecimento humano sobre projeto de software. Os autores também discutiram que a correção de *coupling smells* dever ser entendido como um problema de decisão e não como um problema simples que se aplica algumas refatorações para resolver.

A partir dos resultados encontrados pelas revisões da literatura sobre *code smells*, é possível enumerar algumas deficiências e necessidades para essa área, como: (i) realizar estudos

que avaliem o impacto de *code smells* para atributos internos de qualidade; (ii) executar estudos de refatoração utilizando sistemas industriais; e (iii) analisar o impacto e desenvolver estudos empíricos sobre coocorrências de *code smells* e seus efeitos para a qualidade de software. Tendo sido estabelecida a importância de suprir essas deficiências, o presente trabalho investigará o impacto de coocorrências de *code smells* por meio da identificação de quais são as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade, como coesão, acoplamento, complexidade, herança e também para os desenvolvedores.

As próximas seções foram divididas em três tópicos com trabalhos bem relacionados a proposta desta dissertação.

3.2 Refatoração de *code smells* e atributos internos de qualidade

Na Seção anterior, foram apresentadas duas revisões sistemáticas da literatura abordando a relação entre a refatoração de *code smells* e os atributos de qualidade (KAUR; SINGH, 2019; KAUR, 2019). Os estudos foram relatados até março de 2018. Nesta Seção, foram selecionados estudos mais recentes, descrevendo o impacto da refatoração de *code smells* nos atributos internos de qualidade.

Chávez *et al.* (2017) conduziram um estudo para investigar o impacto das refatorações nos atributos internos de qualidade em 23 projetos *open source* implementados em Java. Os autores analisaram o impacto dos seguintes métodos de refatoração em cinco atributos internos de qualidade (coesão, acoplamento, herança, tamanho e complexidade): *Extract Method*, *Extract Interface*, *Extract Superclass*, *Inline Method*, *Move Field*, *Move Method*, *Rename Method*, *Pull up Field*, *Pull up Method*, *Push Down Field*, *Push Down Method*. Os resultados encontrados mostraram que em 65% dos casos os atributos internos de qualidade melhoraram enquanto que nos outros 35% esses atributos permaneceram inalterados.

AlOmar *et al.* (2019) realizaram um estudo na qual os autores analisam 1245 *commits* para verificar empiricamente o impacto dessas operações de refatoração para métricas de qualidade. Os autores fizeram uma análise manual das mensagens dos *commits* tentanto identificar qual atributo de qualidade deveria ser melhorado com a operação de refatoração. Posteriormente, os autores fizeram a clusterização dos *commits* por atributo interno de qualidade e analisaram as classes antes e depois das operações de refatoração. Com isso, foi possível verificar quais métricas foram significantemente impactadas pelas operações de refatoração e se isso refletia na intenção dos desenvolvedores na melhoria de determinado atributo de qualidade. Como

resultados os autores encontraram que a maioria das métricas que são mapeadas para os principais atributos internos de qualidade como coesão, acoplamento e complexidade, conseguiram melhorar de acordo com a intenção dos desenvolvedores nas mensagens dos *commits*.

Fernandes *et al.* (2020) realizaram um estudo quantitativo sobre o impacto da refatoração para os atributos internos de qualidade. Os autores analisaram 23 sistemas *open source* e um total de 29.303 refatorações. Os resultados encontrados pelos autores apontaram que a maioria das refatorações melhoraram um ou mais atributos internos de qualidade e que os métodos de refatoração *Extract Superclass*, *Inline Method* e *Push Down Field* melhoraram vários atributos. Além disso, eles identificaram que refatorações a nível de método são mais propensas a piorarem vários atributos. Outro resultado identificado é que o método *Extract Method* melhorou a complexidade porém piorou a coesão, o método *Move Field* melhorou a coesão, e o método *Move Method* afetou de forma negativa a coesão, enquanto que a complexidade permaneceu inalterada.

A maioria dos trabalhos na literatura avaliam o impacto da refatoração de ocorrências individuais de *code smells*. No entanto, não existem trabalhos que avaliam o impacto da refatoração das coocorrências de *code smells* para os atributos internos de qualidade. A maioria desses estudos não investigam a percepção dos desenvolvedores sobre as atividades de refatoração. Dessa forma, o presente trabalho será um estudo para investigar o impacto das coocorrências de *code smells* através da identificação das coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade, como coesão, acoplamento, complexidade, herança e também para os desenvolvedores.

3.3 Detecção e análise das coocorrências de *code smells*

Poucos estudos tem sido realizados para investigação de coocorrências de *code smells*. A maioria dos estudos que investigam coocorrências de *code smells*, analisam em sua maioria a detecção das coocorrências, e quais as coocorrências mais frequentes. Nessa Seção, são relatados 3 estudos nesse tema.

Fontana *et al.* (2015) investigaram sobre o fenômeno das coocorrências de 6 *code smells* a partir da contagem da porcentagem de *smells* presentes na mesma classe durante o ciclo de vida de 74 projetos de software. Os autores encontraram que as coocorrências mais comuns foram *Brain Method-Dispersed Coupling* e *Brain Method-Message Chains* e que os resultados encontrados sugerem que *code smells* tendem a coocorrer e interagirem de várias maneiras e

que essas anomalias podem ter uma maior impacto do que ocorrências isoladas. Além disso, dentre os 6 *code smells* estudados, os *smells* mais prejudiciais para a arquitetura de software foram *Brain Method*, *God Class*, *Shotgun Surgery* e *Dispersed Coupling*. No entanto, os autores não verificam o impacto dessas anomalias para atributos internos de qualidade.

Palomba *et al.* (2018) conduziram um estudo empírico de larga escala com o objetivo de quantificar e analisar quais os *code smells* que mais tendem a coocorrer durante o ciclo de desenvolvimento de software. Como resultados, os autores identificaram que 59% das classes são aferadas por mais de uma ocorrência de *code smells* e foram encontrados que geralmente seis pares de *code smells* específicos tendem a coocorrer frequentemente: *Message Chains-Spaghetti Code-Spaghetti Code*, *Message Chains-Complex Class*, *Message Chains-Blob*, *Message-Chains-Refused Bequest*, *Long Method-Spaghetti Code* e *Long Method-Feature Envy*. No entanto, os autores não avaliam o impacto dessas anomalias para a qualidade do software.

Walter *et al.* (2018) realizaram um análise empírica de *collocated smells*, que são interações de coocorrências de *code smells* no mesmo arquivo, e envolveu um conjunto de 92 sistemas de diferentes domínios detectando 14 diferentes *code smells* utilizando 6 ferramentas diferentes. Os relacionamentos entre *code smells* são identificados e examinados utilizando análise de correlação, análise de componentes principais e regras de associação. Como resultados, os autores identificaram em todos os domínios analisados que existe um determinado grupo de *code smells* que tendem a coocorrer como: *Brain Class*, *God Class*, *Dispersed Coupling* e *Long Method*. Além disso, foram encontrados novos relacionamentos como o de *Feature Envy-Shotgun Surgery*, *God Class-Refused Parent Bequest* e que a coocorrência *Long Parameter List-Shotgun Surgery* implica na presença de outros *code smells*: *God Class*, *Long Method* e *Feature Envy*. Os autores realizaram um importante estudo sobre os padrões de coocorrências e quais as mais frequentes levando em consideração vários sistemas. No entanto, não é feita a análise do impacto dessas anomalias para a qualidade dos sistemas.

Os trabalhos de Walter *et al.* (2018) e Palomba *et al.* (2018) levam em consideração coocorrências de *code smells*. No entanto, em ambos os trabalhos é feita uma análise de quais *smells* mais tendem a coocorrer e não verificam de alguma forma o impacto dessas anomalias. No trabalho de Fontana *et al.* (2015) os autores realizam um estudo para verificar o impacto de coocorrências para arquitetura de software e não em atributos internos de qualidade. Dessa forma, o presente trabalho visa preencher as lacunas deixadas por trabalhos anteriores na literatura, por meio de um estudo empírico para investigar o impacto de coocorrências de *code smells*.

3.4 Coocorrências de *code smells* e qualidade de software

Apesar de existirem poucos estudos analisando as coocorrências de *code smells*, alguns estudos que investigaram a relação das coocorrências e qualidade de software foram identificados, e são relatados nessa seção.

Yamashita e Moonen (2013a) analisaram o impacto de relações *inter-smells* para a manutenabilidade de quatro sistemas OO de tamanho médio escritos em Java. Os autores detectaram significantes relacionamentos entre *Feature Envy*, *God Class*,*Long Method*. Foi feita a análise de 12 *code smells* e o experimento contou com 6 engenheiros de software e durante quatro semanas foram realizadas implementações de mudança de requisitos. Além disso, os autores fizeram o registro diário dos problemas encontrados e os artefatos relacionados aos problemas durante as atividades de manutenção. Os autores observaram que relações *Inter-smell* afetam negativamente a manutenibilidade e as atividades de manutenção de software.

Oizumi *et al.* (2016) realizaram um estudo para investigar se coocorrências de *code smells*, na qual os autores chamam de aglomerações, podem significar problemas de projeto de software. Assim, os autores realizaram a identificação de coocorrências de *smells* e analisaram como essas anomaias podem afetar o *design* de 7 sistemas pertencentes a diferentes domínios. Os resultados encontrados apontam que coocorrências de *code smells* podem ocasionar problemas no *design* de software e são uma abordagem efetiva para localizar esses problemas.

Martins *et al.* (2019) investigaram o impacto da refatoração de *Inter-smell* para a manutenibilidade duas Linhas de Produto de Software (LPS), *Health Watcher* e *Mobile Media*. Os autores analisaram a relações *inter-smell* de 5 *code smells* e fizeram a comparação das LPSs antes e após a remoção das relações utilizando métricas de manutenibilidade. Os resultados encontrados sugeriram que coocorrências de *code smells* não trouxeram impacto negativo para a manutenibilidade das LPSs.

Politowski *et al.* (2020) conduziram um estudo com 133 participantes e 372 atividades de compreensão envolvendo coocorrências de dois *code smells*, os *smells Blob* e *Spaghetti Code*. O objetivo do trabalho foi verificar a compreensão dos desenvolvedores sobre o código fonte a partir das coocorrências dessas duas anomalias. Os resultados encontrados pelos autores mostraram que a legibilidade e compreensão do código pioraram, pois os desenvolvedores demoraram mais tempo para finalizar suas atividades e o esforço foi maior para completá-las. Assim, os autores apontam a necessidade dos pesquisadores estudarem mais sobre coocorrências de *code smells* e seus prejuízos para a manutenibilidade de software.

Santana *et al.* (2021) realizaram um estudo sobre coocorrências de 4 *code smells*: *Large Class*, *Long Method*, *Feature Envy* e *Refused Bequest*. Os autores construíram um *dataset* com 20 sistemas Java e encontraram ao todo 15.690 *code smells* ao todo. Além disso, os autores utilizaram regras de associação e análise estatística de dados para identificar e verificar o impacto de coocorrências de *code smells* para a modularidade dos sistemas. Os resultados encontrados apontam que a presença de dois os mais *code smells* implica na presença de *Feature Envy*. E ainda que, classes com coocorrências de *code smells* impactam nos aspectos de modularidade de um sistema.

Todos os trabalhos dessa seção analisam o impacto de coocorrências para algum fator de qualidade. Por exemplo, Yamashita e Moonen (2013a) e Martins *et al.* (2019) verificam o impacto para a manutenibilidade, Oizumi *et al.* (2016) para o *design* de software e Politowski *et al.* (2020) para a compreensão dos desenvolvedores. Porém, nenhum destes trabalhos avalia o impacto dessas anomalias para atributos internos de qualidade e excluindo Politowski *et al.* (2020), todos os outros não levam em consideração a percepção dos desenvolvedores no contexto da refatoração das coocorrências de *code smells*.

3.5 Comparação dos trabalhos relacionados

A Tabela 5 apresenta os trabalhos mais relacionados com o presente trabalho, demonstrando a relação de semelhanças e diferenças. Foram criados alguns critérios para se ter uma melhor visão geral na comparação dos trabalhos: (i) Tipo de Projeto, é um critério que informa se o projeto é *Open Source*, Industrial ou *Open Source*/Industrial; (ii) Avalia coocorrências, critério que verifica se o estudo levou em consideração o conceito de coocorrências (sim ou não); (iii) Percepção dos Desenvolvedores, que é o critério que informa se o estudo verifica a percepção dos desenvolvedores sobre as refatorações (sim ou não); e, (iv) Característica Avaliada, é o critério que informa qual fator que foi avaliado no estudo como atributos internos de qualidade, manutenibilidade, *design* de software, compreensão, arquitetura de software ou frequência das *coocorrências*. É possível perceber que nenhum dos trabalhos relacionados não verifica quais as coocorrências mais prejudiciais para atributos internos de qualidade, a maioria leva em consideração apenas projetos *open source*. Somente um estudo (POLITOWSKI *et al.*, 2020), leva em consideração a percepção dos desenvolvedores sobre a refatoração de coocorrências de *code smells*.

Tabela 5 – Comparação dos trabalhos relacionados

Trabalhos	Tipo de Projeto	Avalia coocorrências	Percepção dos Desenvolvedores	Característica Avaliada
"Proposta" do autor	Industrial	Sim	Sim	A.Internos de Qualidade
(YAMASHITA; MOONEN, 2013a)	Industrial	Sim	Não	Manutenibilidade
(FONTANA *et al.*, 2015)	*Open source*	Sim	Não	Arquitetura
(OIZUMI *et al.*, 2016)	*Open source*/ Industrial	Sim	Não	Design de Software
(CHÁVEZ *et al.*, 2017)	*Open source*	Não	Não	A.Internos de Qualidade
(PALOMBA *et al.*, 2018)	*Open source*	Sim	Não	Frequências de coocorrências
(WALTER *et al.*, 2018)	*Open source*	Sim	Não	Frequências de coocorrências
(MARTINS *et al.*, 2019)	*Open source*	Sim	Não	Manutenibilidade
(ALOMAR *et al.*, 2019)	*Open source*	Não	Não	A.Internos de Qualidade
(FERNANDES *et al.*, 2020)	*Open source*	Não	Não	A.Internos de Qualidade
(POLITOWSKI *et al.*, 2020)	*Open source*	Sim	Sim	Compreensão
(SANTANA *et al.*, 2021)	*Open source*	Sim	Não	Modularidade

Fonte: Elaborado pelo autor.

3.6 Conclusão

Neste capítulo foram apresentados os trabalhos relacionados a este trabalho. Foram identificadas lacunas em aberto e deficiências dos estudos na investigação do impacto das coocorrências de *code smells* nos atributos internos de qualidade. Os resultados encontrados pelas revisões da literatura na Seção 3.1 foram importantes para estabelecer as lacunas e deficiências nos trabalhos que abordam *code smells*, refatoração, qualidade de software e coocorrências de *code smells*. A principais necessidades são: (i) desenvolver mais estudos de refatoração no contexto de sistemas industriais; (ii) analisar a percepção dos desenvolvedores sobre as atividades de refatoração de *smells* e coocorrências de *code smells*; e (iii) verificar o real impacto das coocorrências de *code smells* nos atributos internos de qualidade. Assim, este trabalho visa realizar um estudo para analisar o impacto de coocorrências de *code smells* através da identificação de quais são as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade, como coesão, acoplamento, complexidade, herança e também na percepção dos desenvolvedores.

4 ESTUDO SOBRE O IMPACTO DAS COOCORRÊNCIAS DE *CODE SMELLS* NOS ATRIBUTOS INTERNOS DE QUALIDADE

Este capítulo apresenta um estudo investigando o impacto das coocorrências de *code smells* nos atributos internos de qualidade de três sistemas industriais OO. O estudo foi publicado na trilha principal do Simpósio Brasileiro de Engenharia de Software (SBES) 2020.

4.1 Introdução

O estudo de coocorrências e relações entre *code smells* é uma área de pesquisa relativamente nova. Embora já existam estudos na literatura sobre este tema (PIETRZAK; WALTER, 2006; YAMASHITA *et al.*, 2015; WALTER *et al.*, 2018; PALOMBA *et al.*, 2018). Nenhum dos estudos identificados analisou o impacto de coocorrências ou relações entre *code smells* nos atributos internos de qualidade. Devido ao limitado conhecimento empírico sobre este assunto, nosso estudo visa investigar o impacto das coocorrências de *code smells* nos atributos internos de qualidade.

4.2 Projeto do estudo

4.2.1 Objetivos e questões de pesquisa

O objetivo do estudo é responder qual é o impacto das coocorrências de *code smells* nos atributos internos de qualidade do software, para: identificar quais essas relações, quais são as que mais tendem a coocorrer durante o processo de desenvolvimento e quais são as coocorrências de *code smells* mais difíceis de serem refatoradas do ponto de vista dos desenvolvedores. Assim, com uma melhor compreensão das relações entre os *code smells*, pode-se gerar uma melhor recomendação para os desenvolvedores na detecção e priorização destas anomalias.

O objetivo do estudo está dividido da seguinte forma (WOHLIN *et al.*, 2012): (i) *analisar* as coocorrências de *code smells*; *com o propósito de* compreender seu impacto nos atributos internos de qualidade de software; *com relação a* (i) quais *code smells* tendem a coocorrer juntos, (ii) a remoção de coocorrências de *code smells* antes e depois da refatoração do código, e (iii) quais são as coocorrências mais difíceis de refatorar; *do ponto de vista dos* pesquisadores e desenvolvedores de software; *no contexto de* três sistemas de código-fonte fechado. As questões de pesquisa (QPs) do estudo são apresentadas a seguir:

QP*$_1$*: Quais são as coocorrências mais frequentes de code smells em projetos de código-fonte fechado? – A **QP**$_1$ visa identificar quais e com que frequência os *code smells* tendem a coocorrer juntos. Ao responder à **QP**$_1$, pode-se revelar a existência de diferentes padrões de coocorrências de *code smells* durante o processo de desenvolvimento de software. Além disso, também é possível revelar temas para novas pesquisas nas quais o estudo dessas coocorrências de *smells* ainda não tenha sido realizado.

QP*$_2$*: Como a remoção dos code smells afeta os atributos internos de qualidade em projetos de código-fonte fechado? – A **QP**$_2$ tem como objetivo fornecer evidências sobre o impacto da eliminação das coocorrências de *code smells* nos atributos internos de qualidade. Diferente de estudos anteriores (OIZUMI *et al.*, 2016; FERNANDES *et al.*, 2017) que procuram investigar a introdução das coocorrências de *code smells*, a **QP**$_2$ avalia o impacto da eliminação das coocorrências de *code smells* no que diz respeito a cinco atributos internos de qualidade (coesão, acoplamento, complexidade, herança e tamanho do software). Para atingir este objetivo, a remoção de coocorrências foi realizada na prática com os desenvolvedores dos sistemas analisados. Ao responder a **QP**$_2$, pode-se revelar como a remoção dessas coocorrências impacta nos atributos internos de qualidade.

QP*$_3$*: Quais as coocorrências de code smells consideradas mais difíceis de remover do ponto de vista do desenvolvedor? – A **QP**$_3$ avalia quais são as coocorrências mais difíceis de refatorar do ponto de vista dos desenvolvedores do projeto. O objetivo com esta questão de pesquisa é listar quais são as principais coocorrências de *code smells* que os desenvolvedores devem ter o cuidado de não inserir no código durante o processo de desenvolvimento.

4.2.2 Passos do estudo

Esta seção descreve os passos do estudo, a fim de apoiar a investigação das coocorrências de *code smells.*

Passo 1: Selecionar os sistemas para análise. Foram selecionados 3 sistemas escritos em Java, de código-fonte fechado que estão sendo desenvolvidos por parceiros industriais. Para este propósito, foi solicitado aos gerentes de projeto que indicassem projetos de acordo com os seguintes critérios: (i) sistemas com maior número de linhas de código; (ii) sistemas que não estavam em suas versões iniciais; (iii) sistemas escritos na linguagem Java; e, (iv) sistemas que já se encontravam em um ambiente de produção. A Tabela 6 apresenta as informações gerais de

cada sistema. A primeira coluna mostra o nome do sistema[1]. As demais colunas mostram: o domínio do sistema, número de classes, número de versões e número de linhas de código (LOC). Todos os dados foram coletados pela ferramenta *Understand*.

Tabela 6 – Dados gerais dos sistemas

Sistema	**Domínio**	**# de classes**	**# de versões**	# LOC
S1	Registro odontológico eletrônico	145	5	7830
S2	Oferta acadêmica	99	4	5623
S3	Almoxarifado	106	3	5447

Fonte: Elaborado pelo Autor

O sistema **S1** visa proporcionar um monitoramento integrado dos pacientes atendidos em diferentes clínicas odontológicas. **S2** visa auxiliar no processo de oferta de disciplinas necessárias no início de cada semestre acadêmico na Universidade Federal do Ceará. E finalmente, **S3** visa administrar o estoque de materiais utilizados no curso de Odontologia da Universidade Federal do Ceará, além de fazer entradas e saídas de materiais individualmente para cada clínica. Todos os sistemas em foco foram construídos para a plataforma web utilizando *Spring Boot*, *Thymeleaf* e *Jquery*.

Passo 2: Identificar code smells e suas coocorrências. Foram identificados sete tipos de *code smells* nos sistemas: *Feature Envy*, *God Class*, *Dispersed Coupling*, *Intensive Coupling*, *Refused Parent Bequest*, *Shotgun Surgery* e *Long Method*. A Tabela 1 descreve os sete *code smells* coletados. Os *code smells* foram coletados usando duas ferramentas, JDeodorant (KAUR; SINGH, 2016) e JSpIRIT (VIDAL *et al.*, 2015). Em seguida, foram identificadas as coocorrências dos *code smells* coletados. Os tipos de relacionamentos usados para identificar as coocorrências são descritos no Capítulo 2 na Seção 2.2. Este passo é replicado para todas as versões dos três sistemas selecionados. Portanto, será possível identificar quais coocorrências de *code smells* tendem a aparecer mais durante o processo de desenvolvimento e o número total dessas relações.

Passo 3: Medir os atributos internos de qualidade. A Tabela 3 apresenta as 13 métricas de código que foram utilizadas para medir os atributos internos de qualidade neste trabalho (CHIDAMBER; KEMERER, 1994; LORENZ; KIDD, 1994; DESTEFANIS *et al.*, 2014). As métricas utilizadas neste trabalho foram as seguintes:

– **Coesão**: *LCOM2*.

– **Acoplamento**: *CBO*.

[1] Foram omitidos seus nomes devido a restrições de propriedade intelectual.

- **Complexidade**: *ACC, SCC, MaxNest e EVG*.
- **Herança**: *NOC, DIT e IFANIN.*
- **Tamanho**: *LOC, CLOC, CDL e NIM*.

Para calcular cada métrica, foi utilizada uma licença não comercial da ferramenta *Understand*. Essas métricas foram selecionadas porque nos permitem avaliar as diferentes propriedades de cada atributo (BIEMAN; KANG, 1995; CHIDAMBER; KEMERER, 1994), tais como LOC e CBO que medem o tamanho e o acoplamento, respectivamente. Dessa forma, as métricas podem revelar o efeito das coocorrências de *code smells* sobre os atributos internos de qualidade. Optou-se por realizar a análise em um escopo de classe. No total, foram verificados cinco atributos internos de qualidade: coesão, acoplamento, complexidade, tamanho e herança.

Passo 4: Remoção das coocorrências de code smells por parte dos desenvolvedores. Este passo visa efetuar a remoção das coocorrências dos *code smells* identificados no Passo 2. Para isso, foram convidados os desenvolvedores que contribuíram para o desenvolvimento de cada sistema a participar como sujeitos do estudo. Assim, foi enviado um *Formulário de Identificação* para cada desenvolvedor. O *Formulário de Identificação* é fornecido no Apêndice B. Este formulário tem por objetivo classificar o desenvolvedor quanto à escolaridade, experiência com desenvolvimento de software e seus projetos. Suas respostas foram analisadas para determinar quais deles eram elegíveis para participar do estudo. A Tabela 7 sintetiza as características de cada desenvolvedor selecionado para o experimento. Todos os desenvolvedores são da mesma empresa, mas nem todos tinham conhecimento de todos os sistemas, 5 tinham conhecimento prévio de S1, 4 de S2, e 5 de S3. A empresa liberou os desenvolvedores como uma parte regular do trabalho.

Tabela 7 – Identificação dos desenvolvedores

ID	Experiência em anos	Nível Educacional	Métricas de Qualidade	*Code Smells*	Java
P1	5	Graduação	Avançado	Intermediário	Intermediário
P2	1	Graduação	Básico	Básico	Intermediário
P3	2	Graduação	Avançado	Avançado	Avançado
P4	5	Graduação	Intermediário	Básico	Avançado
P5	2	Graduação	Intermediário	Intermediário	Intermediário
P6	3	Graduação	Avançado	Avançado	Avançado
P7	5	Mestrado	Intermediário	Intermediário	Intermediário

Fonte: Elaborado pelo Autor

Após a seleção os desenvolvedores, solicitou-se a eles que realizassem a remoção de coocorrências de *code smells* (nível de método e nível de classe) em seus sistemas através da refatoração manual do software. O procedimento experimental usado para remover as

coocorrências de *code smells* é explicado na Seção 4.2.3.

Passo 5: Realizar uma nova medição dos atributos internos de qualidade. Após a remoção das coocorrências de *code smell*, realizou-se novas medições dos atributos internos de qualidade. O objetivo era comparar o valor das métricas para cada versão do sistema antes e depois da remoção das coocorrências de *code smells*, através da refatoração manual do software. O conjunto de métricas de qualidade utilizadas para medir os atributos internos de qualidade está descrito na Tabela 3. A análise comparativa foi realizada utilizando os resultados das métricas de qualidade. A comparação foi feita com a versão mais atual de cada sistema antes de remover as coocorrências e com esse mesmo sistema após o procedimento de remoção de coocorrências de *code smells*.

Para avaliar se a qualidade dos sistemas melhorou ou piorou após a remoção de coocorrências de *code smells*, foi adotada a mesma abordagem de (TARWANI; CHUG, 2016), em que os autores utilizam a soma das métricas para comparar a qualidade dos sistemas. Isso significa que se o valor da soma das métricas de cada atributo interno de qualidade aumentasse, esse atributo interno de qualidade teria piorado. Por exemplo, utilizou-se quatro métricas para avaliar o atributo de complexidade como mostrado na Tabela 3, foi medido e calculado a soma dos valores de cada métrica deste atributo antes de remover as coocorrências e após a refatoração são considerados três cenários:

1. Se a soma dessas métricas diminuiu, então a complexidade diminuiu.
2. Se a soma dessas métricas aumentou, então a complexidade aumentou.
3. Se não houve diferença entre as somas, então a complexidade não mudou.

Desta forma, foi adotada esta abordagem para todas as outras métricas e atributos internos de qualidade. Detalhes sobre a detecção de coocorrências de *code smells* e sistemas de medição antes e depois da remoção de coocorrências dos *code smells* são encontrados no site[2].

4.2.3 Procedimentos experimentais

O estudo foi composto por um conjunto de quatro atividades, detalhadas a seguir.

Atividade 1: Sessão de treinamento. Foi conduzida uma *sessão de treinamento* com todos os participantes sobre os conceitos essenciais para o estudo, tais como *code smells*, atributos internos de qualidade e refatoração. Também foram treinados os participantes do estudo para identificar as coocorrências de *code smells*. O treinamento teve duração de 1 hora e meia.

[2] https://julioserafim.github.io/SBES2020/

Foram apresentados um conjunto de exemplos práticos que ilustram as operações de refatoração que poderiam ser aplicadas em cada *code smell* apresentado na primeira parte do treinamento. Em seguida, foi fornecido um conjunto de exercícios para que os desenvolvedores pudessem aplicar métodos de refatoração para remover os *code smells*. Essa etapa teve duas horas de duração. Foi decidido proporcionar uma sessão de treinamento para nivelar seus conhecimentos sobre os principais conceitos relativos ao nosso estudo. Dessa forma, tenta-se reduzir o viés, concentrando-nos nos principais conceitos e apresentando exemplos teóricos e práticos.

Atividade 2: Remoção das coocorrências de *code smells* através de refatoração manual. Foi solicitado aos desenvolvedores que realizassem a remoção de coocorrências de *code smells* (nível de método e nível de classe) em seus sistemas através da refatoração manual de software. Por exemplo, se houver uma ocorrência do *Long Method* e *Feature Envy* no mesmo método, o desenvolvedor pode escolher apenas um destes *code smells* para remover. Esta remoção resulta na eliminação da coocorrência a nível do método. Por outro lado, uma coocorrência em nível de classe acontece quando há uma *God Class* ou um *Refused Parent Bequest* junto com algum outro *code smell*, caso em que os desenvolvedores também poderiam escolher qual o *code smell* a ser removido, eliminando assim a coocorrência em nível de classe.

Para facilitar a remoção de coocorrências de *code smells*, foi fornecido aos participantes uma lista que resumia o nome dos métodos ou classes em que as coocorrências de *code smells* foram identificadas a partir do Passo 2. Além disso, para cada coocorrência de *code smell*, foram criadas *issues* no Github relacionadas às atividades de refatoração. Cada *issue* continha informações sobre a classe e o método afetado por um *code smell*. Assim, os desenvolvedores eram livres para escolher as *issues* e, consequentemente, qual *code smell* a ser refatorado para remover a coocorrência. Foram realizadas reuniões semanais para verificar o progresso das atividades e se os desenvolvedores encontraram algum tipo de dificuldade ou obstáculo no processo de refatoração. Os desenvolvedores foram instruídos a deixar claro quais *commits* estavam relacionados a uma atividade de refatoração. Assim, cada *commit* foi marcado com a etiqueta que representa o nome do *code smell* a ser refatorado para eliminar a coocorrência (por exemplo *God Class*). Além disso, foram criadas *branches* separadas para cada uma das atividades de refatoração.

Atividade 3: Validação da remoção das coocorrências de *code smells*. Os *commits* foram analisados para verificar se a coocorrência foi eliminada pelo programador. A cada *commit* de refatoração, a refatoração da coocorrência foi validada. Se validado, o código

Tabela 8 – Número de *commits* de refatoração e número de coocorrências removidas

Sistema	**Coocorrências removidas**	***Commits* de refatoração**	**Total de *commits***
S1	37	95	2993
S2	33	51	1045
S3	24	37	1217

Fonte: Elaborado pelo Autor

da *branch* deve ser incorporado ao repositório principal. Se a atividade não foi validada, o desenvolvedor deve refatorar mais uma vez até que a coocorrência não ocorra mais.

A análise dos projetos ocorreu em momentos diferentes, ou seja, não foram analisados os três sistemas ao mesmo tempo. O primeiro projeto (S1) foi o primeiro a ser estudado, o segundo foi o sistema S2 e finalmente o sistema S3. Todo o processo de refatoração, análise e estudo dos três projetos levou três meses e incluiu sete desenvolvedores diferentes.

E para realizar a remoção das coocorrências de *code smells* em cada sistema, foram necessários vários *commits*. Como pode ser visto na Tabela 8. Nesta tabela, apresenta-se o número de *commits* de refatoração e o número de coocorrências removidas em cada um dos sistemas.

Atividade 4: Aplicação do questionário de acompanhamento. Após as atividades de refatoração, foi aplicado um questionário para verificar a percepção dos desenvolvedores sobre essas atividades. Por exemplo, foi perguntado a eles se já haviam utilizado os conceitos de *code smell*, refatoração e métricas de qualidade. Também foi solicitado para que eles dissessem quais eram os *code smells* mais difíceis e fáceis e as coocorrências mais difíceis e fáceis de refatorar e as razões de suas respostas.

4.3 Resultados e Discussão

4.3.1 A frequência das coocorrências de code smells (QP_1)

A $\mathbf{QP}_1$ foi respondida identificando as coocorrências mais frequentes de *code smells* nas versões dos três sistemas estudados. Os procedimentos que foram utilizados para identificar as coocorrências de *code smells* são descritos na Seção 4.2.3. A Tabela 9 apresenta a frequência de cada coocorrência agrupada por sistema e versão. A primeira e a segunda colunas listam cada sistema e versão. As demais colunas apresentam cada coocorrência de *code smell*.

As coocorrências mais frequentes de *code smells*. Na Tabela 9 observa-se que existem pelo menos cinco tipos de *code smells* que tendem a formar uma coocorrência: *Feature*

Tabela 9 – Coocorrências de *code smells* nos três sistemas

Sistema	Versão	FE e LM	DCO e LM	DCO e FE	GC e LM	IC e LM	GC e SS	FE e GC	FE e RPB	DCO e GC	FE e SS
S1	v1.0	9	1	5	7	0	1	2	2	0	0
	v1.1	15	6	4	7	1	1	5	2	0	0
	v1.2	10	8	3	9	1	1	4	2	0	0
	v1.3	12	8	3	9	1	1	3	0	0	0
	v1.3.1	11	8	3	7	1	1	3	3	0	0
	Total	**57**	**31**	**18**	**39**	**4**	**5**	**17**	**9**	**0**	**0**
S2	v0.1	2	2	3	1	1	6	4	0	5	0
	v0.2	3	4	4	6	0	0	1	0	5	0
	v0.2.1	3	5	4	8	1	4	2	0	6	0
	Total	**8**	**11**	**11**	**15**	**2**	**10**	**7**	**0**	**16**	**0**
S3	v0.1	1	2	2	5	0	0	1	0	2	0
	v0.2	1	1	0	2	0	0	2	0	1	0
	v1.0	3	4	1	6	0	2	5	0	2	1
	Total	**5**	**7**	**3**	**13**	**0**	**2**	**8**	**0**	**5**	**1**

Fonte: Elaborado pelo Autor

Tabela 10 – Coocorrências que mais tendem a coocorrer

Coocorrências	**# de sistemas em que coocorreram**	**Sistemas**
God Class–Long Method	3	S1,S2,S3
Dispersed Coupling–Long Method	3	S1,S2,S3
Feature Envy–Long Method	2	S1,S3
Dispersed Coupling–Feature Envy	2	S1,S2
Feature Envy–God Class	2	S1,S3

Fonte: Elaborado pelo Autor

Envy, *Long Method*, *Dispersed Coupling*, *God Class* e *Shotgun Surgery*. Com relação aos tipos mais frequentes de coocorrências de *code smells*, tem-se algumas observações interessantes. O *ranking* dos 5 principais tipos de coocorrência de *code smells* por sistema, desde os mais frequentes até os menos frequentes, indica que *God Class–Long Method* e *Dispersed Coupling–Long Method*, são as coocorrências que mais tendem a coocorrer juntas.

Além disso, também foram encontradas coocorrências de *code smells* que mais são detectadas em pelo menos dois sistemas, tais como: *Feature Envy–Long Method* nos sistemas S1 e S3, *Dispersed Coupling–Feature Envy* nos sistemas S1 e S2 e *Feature Envy–God Class* nos sistemas S1 e S3, indicando um padrão em que essas coocorrências tendem a coocorrer. A Tabela 10 apresenta as coocorrências que mais tendem a coocorrer nos sistemas estudados. Estes resultados confirmam o que foi encontrado em trabalhos anteriores na literatura sobre coocorrências de *code smells* (LANZA; MARINESCU, 2007; YAMASHITA; MOONEN, 2013b; LOZANO *et al.*, 2015; PALOMBA *et al.*, 2018).

Lição 1: *As coocorrências mais frequentes de code smells entre os projetos são God Class–Long Method e Dispersed Coupling–Long Method.*

As coocorrências de *code smells* tendem a aumentar durante a evolução do software. Comparando a primeira e a última versão de cada sistema, pode-se observar um

aumento no número de coocorrências de *code smells* na maioria dos sistemas analisados. Mais precisamente, para o sistema S1, observa-se um aumento em cinco das oito (**62,5**%) coocorrências: *Feature Envy–Long Method*, *Dispersed Coupling–Long Method*, *Intensive Coupling–Long Method*, *Feature Envy–God Class*, e *Feature Envy–Refused Parent Bequest*. No caso do sistema S2, observa-se também um aumento de (**62,5**%) nas seguintes coocorrências: *Feature Envy–Long Method*, *Dispersed Coupling–Long Method*, *Dispersed Coupling–Feature Envy*, *God Class–Long Method* e *Dispersed Coupling–God Class*. Finalmente, no sistema S3, um aumento em 6 das 8 (**75%**) coocorrências de *code smells* foi observado: *Feature Envy–Long Method*, *Dispersed Coupling–Long Method*, *Dispersed Coupling–Feature*, *God Class–Long Method*, *God Class–Shotgun Surgery*, *Feature Envy–God Class*, e *Feature Envy–Shotgun Surgery*.

Estes resultados sugerem que as coocorrências tendem a aumentar com o tempo. Um dos fatores que pode explicar este fenômeno é o número de funcionalidades em cada versão. As últimas versões têm mais funcionalidades do que as primeiras. Entende-se que os desenvolvedores também devem se preocupar com o número de instâncias individuais de *code smells*, como *Long Method*, *God Class* e *Feature Envy* porque estes *code smells* tendem a coocorrer com algum outro *code smell*. Na Tabela 9, é possível notar que cada um desses *code smells* está presente em pelo menos quatro relações de coocorrência. Entretanto, são necessários mais estudos empíricos para verificar a relação entre o número de funcionalidades e o número de coocorrências de *code smells*.

Lição 2: *A maioria das coocorrências aumentou ao longo do desenvolvimento dos sistemas.*

4.3.2 O impacto da remoção das coocorrências de code smells (QP$_2$)

A **QP**$_2$ avalia o impacto da remoção das coocorrências de *code smells* em cinco atributos internos de qualidade: coesão, herança, tamanho, acoplamento e complexidade. Enfatiza-se que a remoção de coocorrências foi realizada na prática através de refatorações manuais aplicadas por desenvolvedores familiarizados com os sistemas (ver a Seção 4.2.2). A Tabela 16 apresenta o impacto da remoção das coocorrências de *code smells* para os atributos internos de qualidade, considerando as versões mais recentes dos três sistemas.

Inclui-se os atributos de qualidade e suas respectivas métricas. Analisou-se a qualidade dos três sistemas usando a ferramenta *Understand* antes do processo de remoção das coocorrências de *code smells* e o valor computado para cada métrica pode ser visto na Tabela 16.

Identificou-se que após a remoção das coocorrências de *code smells*, a coesão

Tabela 11 – Impacto da remoção das coocorrências de *code smells* por sistema e atributo interno de qualidade

Sistema		Coesão	Complexidade				Herança			Acoplamento	Tamanho			
		LCOM2	ACC	SCC	EVG	MaxNet	DIT	NOC	IFANIN	CBO	LOC	CLOC	NIM	CDL
S1 com coocorrências		3596	103	1120	867	111	172	34	138	551	7830	166	919	146
	Total	3596				2201			344	551				9061
S1 sem coocorrências		3878	110	1223	247	116	186	35	151	536	8416	199	1004	163
	Total	3878				1696			372	536				9782
Resultados		**↓7,75%**			**↓22,94%**			**↑ 8,13%**		**↑3,2%**		**↑7,95%**		
S2 com coocorrências		3300	101	881	177	74	111	15	119	332	7094	112	719	106
	Total	3300				1233			245	332				8031
S2 sem coocorrências		3438	102	913	175	74	113	16	124	333	5623	109	748	112
	Total	3438				1264			253	333				6592
Resultados		**↓4,1%**			**↑2,5%**			**↑3,2%**		**↑0,3%**		**↓17,91%**		
S3 com coocorrências		3634	86	770	102	62	101	12	109	313	5082	151	640	99
	Total	3634				1020			222	313				5972
S3 sem coocorrências		3856	78	579	93	53	108	11	119	334	5447	126	696	104
	Total	3856				803			238	334				6373
Resultados		**↓6,1%**			**↓21,27%**			**↑ 7,2%**		**↑ 9,9%**		**↑6,71%**		

Fonte: Elaborado pelo Autor

diminuiu nos três sistemas estudados neste trabalho. O acoplamento e a herança aumentaram nos três sistemas. A complexidade diminuiu significativamente em dois sistemas (S1 e S3) e aumentou em um sistema (S2). O tamanho dos sistemas S1 e S3 aumentou ligeiramente e o tamanho do sistema S2 foi reduzido.

Identificou-se que a coesão do sistema S1 piorou em **7,75%** após a remoção de coocorrências de *code smells*, isto pode ser visto através do valor da métrica LCOM2 em que com coocorrências de *code smells* era 3596 e após o processo de remoção de coocorrências passou para 3878. Quanto maior o valor desta métrica, pior é a coesão de um sistema (CHIDAMBER; KEMERER, 1994). Por outro lado, o valor da métrica de complexidade diminuiu após a remoção das coocorrências. Para avaliar o impacto da remoção de *code smells* em atributos com mais de uma métrica, comparou-se a soma do valor da métrica (TARWANI; CHUG, 2016) antes da remoção dos *code smells* e depois da remoção dos *code smells*, como mostrado na Tabela 16. Pode-se verificar na Tabela 16 que a soma dos atributos de complexidade diminuiu de 2021 para 1696, indicando uma diminuição de **22,94%** do valor total da complexidade (CHIDAMBER; KEMERER, 1994; NEAMTIU *et al.*, 2013).

No sistema S1, os atributos de herança e tamanho obtiveram um aumento de **8,13%** e **7,95%** respectivamente no valor de suas métricas. O acoplamento aumentou em **3,2%**. Espera-se que haja uma redução no acoplamento percentual (CHOWDHURY; ZULKERNINE, 2010). Uma possível explicação para o aumento do acoplamento, apesar da remoção das coocorrências, é o aumento do valor das métricas de herança, uma vez que se descobriu anteriormente que o aumento na herança pode significar um aumento no acoplamento de classes (KRISHNAPRIYA; RAMAR, 2010).

O sistema S2, foi o único sistema estudado neste trabalho que obteve uma grande redução em seu tamanho após a remoção de coocorrências de *code smells*. Pode-se ver na Tabela 16 que o tamanho diminuiu em **17,91%**. O número de linhas de código (LOC) diminuiu de 7094 para 5623, mas o número de métodos (NIM) e o número de classes (CDL) aumentaram. Embora o atributo de complexidade não tenha diminuído neste sistema, houve um pequeno aumento de **2,5%** em complexidade e também produziu a menor redução na coesão e menores métricas de acoplamento e aumento de valor na herança de atributos. Finalmente, no sistema S3, identificou-se uma diminuição de **6,3%** no atributo de coesão, um aumento na herança de **7,2%** e no acoplamento de **9,9%**, indicando mais uma vez que o aumento da herança no sistema pode sugerir um aumento no acoplamento. O tamanho também aumentou em **6,71%**. No entanto, houve uma diminuição significativa da complexidade em **21,27%**.

O impacto negativo da remoção das coocorrências de *code smells*. A remoção de coocorrências de *code smells* teve um impacto negativo nos atributos internos de qualidade, tais como coesão e acoplamento. Após remover estas anomalias, descobriu-se que estes atributos pioraram em todos os três sistemas estudados neste trabalho.

Lição 3: *A remoção de coocorrências de code smells não teve um impacto positivo em atributos tais como coesão e acoplamento.*

O impacto positivo da remoção das coocorrências de *code smells*. Por outro lado, a remoção de coocorrências de *code smells* conseguiu reduzir significativamente a complexidade nos sistemas S1 e S3. Vários trabalhos na literatura já estudaram o impacto da complexidade nos sistemas OO. A maioria destes estudos associa a complexidade a problemas como a diminuição da capacidade de manutenção do software, maior propensão a erros e redução da qualidade. (SUBRAMANYAM; KRISHNAN, 2003; DARCY *et al.*, 2005; NEAMTIU *et al.*, 2013; ALENEZI; ALMUSTAFA, 2015). No sistema S1, a complexidade foi reduzida em 22,94% e no sistema S3 para 21,97%.

Lição 4: *Depois de remover as coocorrências, a complexidade diminuiu 22,94% e 21,27% respectivamente em dois sistemas dos três sistemas alvo estudados. Indicando que a complexidade pode diminuir com a remoção das coocorrências.*

Os dados obtidos nesta questão de pesquisa sugerem evidências de que a remoção de coocorrências de *code smells* pode acarretar uma redução no atributo de complexidade. Entretanto, mais estudos empíricos precisam ser realizados para se obter uma melhor compreensão.

Tabela 12 – Coocorrências mais difíceis de serem refatoradas

Posição	Coocorrência de *code smell*	Pontos
1	*God Class–Long Method*	61
	Dispersed Coupling–Long Method	61
2	*Feature Envy–Long Method*	57
	Feature Envy–God Class	57
3	*Dispersed Coupling–Feature Envy*	53
4	*God Class–Shotgun Surgery*	50
	Feature Envy–Shotgun Surgery	50
5	*Feature Envy–Refused Parent Bequest*	49
	Dispersed Coupling–God Class	49
6	*Intensive Coupling–Long Method*	47

Fonte: Elaborado pelo Autor

4.3.3 As coocorrências de code smells mais difíceis de remover (QP$_3$)

A **QP$_3$** foi respondida solicitando aos desenvolvedores que respondessem um questionário sobre as atividades de refatoração. Esse questionário é fornecido no Apêndice A. Neste questionário, os desenvolvedores que informaram em uma escala de 1 a 5 quais eram as coocorrências de *code smells* mais difíceis de refatorar, onde 1 é o mais fácil e 5 o mais difícil, se o desenvolvedor não tivesse refatorado uma certa coocorrência ele marcou a opção "*Eu não refatorei esta coocorrência de code smell*". Além disso, também foi solicitado aos desenvolvedores que escrevessem em um campo as razões dessas escolhas para ter uma melhor compreensão das suas percepções.

Os dados foram organizados em um *ranking* utilizando a técnica de *Contagem de Borda* (REILLY, 2002). *Contagem de Borda* é uma técnica de classificação projetada para obter um consenso em vez de uma maioria. Utilizou-se esta técnica da seguinte forma. Se eles tiverem **n** candidatos, o primeiro no *ranking* tem **n** pontos, o segundo **n-1** pontos, o terceiro tem **n-2** pontos, e assim por diante. Esta técnica também foi utilizada em um estudo anterior que classificou os *code smells* mais populares entre os desenvolvedores (YAMASHITA; MOONEN, 2013a). A Tabela 12 apresenta o *ranking* das coocorrências dos *code smells* mais difíceis de refatorar.

A perspectiva do desenvolvedor sobre a coocorrência de *code smells*. Observa-se que, sob a perspectiva do desenvolvedor, as coocorrências mais difíceis de serem refatoradas, das mais frequentes para as menos frequentes, são: (1) *God Class–Long Method* e *Dispersed Coupling–Long Method*; (2) *Feature Envy–Long Method* e *Feature Envy–God Class*; (3) *Dispersed Coupling–Feature Envy*; e *Intensive Coupling–Long Method* com uma diferença significativa

de pontos em relação as coocorrências que estão no topo.

Resultado 5: *As coocorrências God Class–Long Method e Dispersed Coupling–Long Method foram as mais difíceis de remover do ponto de vista dos desenvolvedores.*

Com este resultado, identificou-se um problema potencial envolvendo coocorrências de *code smells*. Na Seção 4.3.1, constatou-se que as coocorrências *God Class–Long Method* e *Dispersed Coupling–Long Method* eram as mais propensas a coocorrerem. Entretanto, como mostrado na Tabela 9, estas foram também são as coocorrências mais difíceis de serem refatoradas. Esta descoberta sugere que os desenvolvedores devem tomar cuidado para não inserir estas coocorrências no código fonte.

Lição 6: *Os desenvolvedores devem ter o cuidado de não inserir as coocorrências God Class–Long Method e Dispersed Coupling–Long Method.*

Além disso, as coocorrências *Feature Envy–Long Method* e *Feature Envy–God Class* também devem receber atenção especial dos desenvolvedores, pois encontram-se na segunda posição na Tabela 12 e na Seção 4.3.1 elas também ocorreram com certa frequência, já que estavam em 2 dos 3 sistemas estudados neste trabalho. Também obteve-se algumas explicações interessantes sobre por que os desenvolvedores refatoraram uma certa coocorrência de *code smell*:

P1:"Sempre considerei o smell mais fácil de refatorar em cada coocorrência e a dificuldade de refatorar esse smell como sendo a dificuldade de refatorar a coocorrência."

P3: "Eu escolhi as dificuldades com base na dificuldade de cada code smell envolvido. Assim, os casos que têm God Class e Shotgun Surgery são os que apresentam maiores dificuldades e os mais fáceis são os que têm Feature envy, Long method e Refused Parent Bequest."

P4: "O fato é que God Class é mais complexa devido ao fato de que temos que criar outra classe e ainda não a refatoramos, e Long Method é o mais fácil porque já tínhamos experiência."

P5:"Devido ao fato de que para corrigir a coocorrência era necessário corrigir o code smell Feature Envy, não sendo necessárias muitas mudanças."

Uma descoberta do estudo foi que os desenvolvedores tinham mais dificuldade e não gostavam de remover os *code smells* como *God Class*, *Shotgun Surgery*, *Dispersed Coupling* e

Intensive Coupling das coocorrências. De acordo com os desenvolvedores, estes *smells* "*envolvem muitas funções e variáveis*", "*exigem mudanças em vários lugares no código,*" e "*exigem muitas operações para removê-los".* Enquanto os *smells* como *Feature Envy*, *Long Method* e *Refused Parent Bequest* foram mais fáceis de refatorar porque, de acordo com os desenvolvedores, *"as refatorações são mais simples e mais rápidas*" e "*foi necessário refatorar somente um ou alguns métodos".*

Essas respostas indicam que os desenvolvedores não consideraram a coocorrência como um todo, mas quais eram os *code smells* que faziam parte dessa coocorrência.

4.4 Ameaças à validade

Esta seção discute as ameaças à validade do estudo, de acordo com a classificação de Wholin (WOHLIN *et al.*, 2012).

Validade interna. Uma ameaça interna deste estudo é o baixo número de sistemas analisados. Entretanto, os sistemas utilizados neste estudo são sistemas de código fechado, que são pouco analisados na literatura. Outra questão identificada é que as classes analisadas são entidades de produção, ou seja, não considerou-se as entidades de teste utilizadas para testar as classes de produção ao medir os atributos internos de qualidade. No entanto, considerou-se que os desenvolvedores estão mais preocupados com classes que fornecem determinadas funcionalidades para o sistema e não com classes de teste.

Validade de construção. Os *code smells* foram identificados automaticamente pelas ferramentas JSpIRIT e JDeodorant, reduzindo a chance de erros na detecção. Mesmo assim, as estratégias implementadas por estas ferramentas podem ser uma ameaça potencial à validade. Desta forma, outras ferramentas de detecção poderiam utilizar estratégias diferentes das ferramentas utilizadas neste estudo. Assim, isto poderia causar uma variação no conjunto de *code smells* identificados e consequentemente afetar a detecção de coocorrências de *code smells*. Existem vários tipos de relações entre *code smells* encontrados em outros estudos, tais como *coupled smells* e *colocated smells* (YAMASHITA; MOONEN, 2013a). Em nosso estudo, consideramos as coocorrências de *smells* como ocorrendo nos níveis de classe e método.

Validade externa. Os resultados só podem ser utilizados para sistemas OO escritos em Java. Uma limitação é o domínio dos sistemas. A partir de outros domínios é possível ter resultados diferentes. Outro problema identificado é que existem desenvolvedores com pouca experiência de desenvolvimento ou pouco conhecimento sobre *code smells*, refatoração ou

métricas de qualidade. Para mitigar este problema, foram realizado um treinamento com todos os desenvolvedores.

4.5 Conclusão

O estudo executado neste trabalho considerou 7 tipos de *code smells* e suas coocorrências em 3 sistemas Java OO de código fechado, e 5 atributos internos de qualidade (coesão, herança, tamanho, acoplamento e complexidade). Como objetivo principal do estudo: (i) análise das coocorrências que mais tendem a coocorrer nestes sistemas; (ii) investigação do impacto da remoção destas anomalias nos atributos internos de qualidade; e, (iii) identificação de quais são as coocorrências a serem removidas de acordo com a perspectiva dos desenvolvedores. O processo de remoção dessas coocorrências de *code smells* levou 3 meses e aconteceu em momentos diferentes para cada sistema, um total de 183 *commits* foram feitos e 94 coocorrências foram removidas.

As principais conclusões deste estudo, foram: (i) *God Class–Long Method* e *Dispersed Coupling–Long Method* são as coocorrências mais frequentes nos três sistemas e também as coocorrências mais difíceis de refatorar na perspectiva dos desenvolvedores; (ii) as coocorrências aumentam durante o desenvolvimento do sistema; (iii) a remoção dessas anomalias tem um impacto negativo nos atributos de coesão e acoplamento; e, (iv) a remoção das coocorrências de *code smells* sugere uma diminuição significativa na complexidade dos sistemas. A descoberta (iv) é interessante porque vários estudos na literatura apontam os danos de alta complexidade para os sistemas OO (SUBRAMANYAM; KRISHNAN, 2003; DARCY *et al.*, 2005; NEAMTIU *et al.*, 2013).

5 ESTUDO SOBRE AS COOCORRÊNCIAS MAIS PREJUDICIAIS PARA ATRIBUTOS INTERNOS DE QUALIDADE E PERCEPÇÕES DOS DESENVOLVEDORES

Este capítulo apresenta o estudo realizado para verificar quais as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade a partir da análise de *commits* de refatoração dessas anomalias, e para os desenvolvedores a partir da análise de respostas de suas percepções durante as atividades de refatoração das coocorrências de *code smells*. O estudo foi aceito para publicação na trilha principal do Simpósio Brasileiro de Engenharia de Software (SBES) 2021.

5.1 Introdução

Através do estudo realizado no Capítulo 4, foi possível ter um entendimento inicial de qual o impacto e como as coocorrências de *code smells* podem afetar os atributos internos de qualidade. Também foi possível identificar quais as coocorrências mais frequentes e como essas anomalias se formam durante as *releases* de um determinado software. Além disso, foi possível verificar a opinião e o ponto de vista dos desenvolvedores sobre quais foram os *code smells* e as coocorrências de *code smells* mais difíceis e fáceis de se refatorar no código fonte. Isso implica em um melhor direcionamento e suporte para que os desenvolvedores tomem cuidado em inserir determinadas coocorrências no código fonte.

Os resultados do estudo do Capítulo 4 forneceram uma base para entender coocorrências de *code smells* para a qualidade de software. No entanto, foram estudados apenas 3 sistemas e por esse motivo, é necessário analisar e avaliar o impacto dessas anomalias nos atributos internos de qualidade para mais sistemas com o objetivo de ter um conhecimento mais sólido e empírico. Além disso, também é necessário entender melhor a experiência dos desenvolvedores e seu ponto de vista sobre a remoção de coocorrências de *code smells* através de refatoração. Outra questão importante que o estudo anterior não abordou e que foi abordado no neste novo estudo, refere-se a quais coocorrências de *code smells* são mais prejudiciais para os atributos internos de qualidade e para os próprios desenvolvedores.

Dessa forma, foi realizado um novo estudo levemente controlado que tem como objetivo investigar quais são as coocorrências de *code smells* mais prejudiciais para quatro atributos internos de qualidade – coesão, acoplamento, complexidade e herança. Com o objetivo de priorizar uma análise mais profunda dos atributos coesão, acoplamento, complexidade e herança

o atributo tamanho não foi considerado neste estudo, além disso, o tamanho é o atributo interno menos considerado e estudado na literatura (DALLAL; ABDIN, 2017). Para esse propósito, foram analisados dados de 5 sistemas OO de código fechado. Foi realizada a identificação das coocorrências mais prejudiciais para os atributos internos de qualidade e também na percepção dos desenvolvedores. Além disso, foi analisada a percepção dos desenvolvedores durante a remoção de coocorrências de *code smells* através da refatoração. As percepções foram coletadas a partir da técnica de diário, na qual os desenvolvedores documentaram suas percepções durante todo o processo de remoção de cada coocorrência de *code smell*. Por fim, foi feita a análise das principais dificuldades enfrentadas pelos desenvolvedores durante a remoção dessas anomalias. Nas próximas seções são apresentados o planejamento e os resultados do estudo.

5.2 Projeto do estudo

5.2.1 Objetivos e questões de Pesquisa

Baseado em (WOHLIN *et al.*, 2012), foi feita a definição do objetivo deste estudo: verificar o impacto da refatoração das coocorrências de *code smells* para os desenvolvedores e os atributos internos de qualidade. Para alcançar esse objetivo o estudo do Capítulo 4 será ampliado para ter um melhor entendimento do (i) impacto de coocorrências das *code smells* para os atributos internos de qualidade; (ii) percepções dos desenvolvedores durante as atividades de refatoração de cooocorrências; (iii) verificar quais a coocorrências são mais prejudiciais para os atributos internos de qualidade; e, (iv) verificar se no ponto de vista dos desenvolvedores essas coocorrências são as mais prejudiciais.

Dessa forma, foram definidas as seguintes questões de pesquisa para o estudo:

QP$_1$***:*** *Quais as coocorrências de code smells são mais prejudiciais para os atributos internos de qualidade?* – **A QP**$_3$ tem como objetivo verificar quais são as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade com base nos resultados das métricas da Tabela 3. Analisar essa questão é importante pois conhecer e entender quais as coocorrências mais prejudiciais para a qualidade de software pode fornecer um bom direcionamento sobre quais coocorrências devem ser evitadas. Dessa forma, o objetivo em responder essa questão de pesquisa é entender que coocorrências trazem maior prejuízo para a qualidade de um software.

QP$_2$***:*** *Quais as coocorrências de code smells mais prejudiciais para os desenvolve-*

dores? – A **QP**$_2$ avalia quais são as coocorrências que são prejudiciais levando em consideração o ponto de vista dos desenvolvedores. Essa questão é importante, pois estudos anteriores têm considerado o impacto de coocorrências sobre a manutenibilidade (MALHOTRA; CHUG, 2016; YAMASHITA; MOONEN, 2013a; MARTINS *et al.*, 2019), mas não levam em consideração as percepções dos desenvolvedores sobre as coocorrências (apenas utilizam formulários genéricos). Dessa forma, o objetivo em responder essa questão de pesquisa é entender a percepção dos desenvolvedores sobre as atividades de refatoração e sobre as coocorrências de *code smells*.

QP$_3$***: Quais são as principais dificuldades enfrentadas pelos desenvolvedores durante a remoção de coocorrências de code smells?*** – A **QP**$_3$ tem como objetivo identificar as principais dificuldades enfrentadas pelos desenvolvedores durante a remoção de coocorrências de *code smells* via refatoração. Ao responder a **QP**$_3$, é possível entender quais são as principais dificuldades e os critérios que os desenvolvedores levam em consideração para explicar a dificuldade de remover as coocorrências de *code smells*.

5.2.2 Passos do estudo

Conforme discutido no começo deste capítulo, o objetivo deste trabalho é investigar o impacto de coocorrências de *code smells* através da identificação de quais são as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade, como coesão, acoplamento, complexidade, herança e também para os desenvolvedores. Para isso, serão definidos alguns passos para que o objetivo seja alcançado. Os passos do estudo são descritos a seguir.

Passo 1: Selecionar sistemas industriais OO escritos em Java. Foram selecionados cinco sistemas OO de código fechado escritos em Java. Para selecionar os sistemas, foram usados os seguintes critérios: (i) o sistema deve ter como linguagem base, pois a maioria das ferramentas para detecção de *code smells* e métricas de software utilizam o Java; (ii) sistemas que já estão um ambiente de produção; e (iii) sistemas com o maior número de linhas de código.

A Tabela 13 apresenta os dados de cada sistema. A primeira coluna se refere a numeração do sistema. As colunas restastes apresentam: nome do sistema, número de classes, número de métodos e número de linhas de código. Todos os dados foram coletados usando a ferramenta *Understand*.

O S1 permite gerenciar evento públicos e privados. S2 armazena e gerencia ações de extensão e projetos de pesquisa desenvolvidos pela comunidade acadêmica. S3 permite o gerenciamento de riscos na universidade. S4 permite gerenciar as competências dos funcionários

Tabela 13 – Dados dos sistemas analisados

Sistema	Nome	# de classes	# de métodos	# de linhas de código
S1	Contest	78	612	4790
S2	GPA Extensão	110	706	6464
S3	Gestão de Riscos	106	698	4296
S4	Gestão de Competências	183	1247	8394
S5	Atividade Complementar	48	250	1910

Fonte: Elaborado pelo Autor

da instituição. S5 facilita o gerenciamento das atividades complementares dos estudantes da universidade. Todos esses sistemas são WEB e foram desenvolvidos utilizando o *Spring framework* como tecnologia de *back-end* e utilizando Thymeleaf, Vue.js e JQuery como tecnologias de f*ront-end*.

Passo 2: Fazer a detecção de code smells e suas coocorrências. Antes da detecção de coocorrências de *code smells* foi necessário fazer primeiramente a identificação das ocorrências individuais de seis tipos de *smells*: *Feature Envy*, *God Class*, *Dispersed Coupling*, *Intensive Coupling*. *Shotgun Surgery* e *Long Method*. Os *code smells* foram coletados através do uso das ferramentas JDeodorant (KAUR; SINGH, 2016) e JSpIRIT (VIDAL *et al.*, 2016). Uma vez que as ocorrências individuais de *code smells* foram encontradas, foi possível detectar as coocorrências dessas anomalias em cada sistema. Os tipos de relacionamentos usados para identificar as coocorrências são descritos na Seção 2.2.

Passo 3: Realizar a medição dos atributos internos de qualidade. Antes de realizar a refatoração das coocorrências de *code smells* foi feita a medição da qualidade de todos os sistemas a partir da medição dos atributos internos de qualidade. Para computador o valor de cada métrica, foi utilizada a ferramenta *Understand* com licença para pesquisadores. Ao todo foram utilizadas 13 métricas de software para mensurar quatro atributos internos de qualidade: coesão, acoplamento, complexidade e herança. A Tabela 3 mostra as métricas utilizadas, bem como sua definição e a qual atributo interno de qualidade se refere. Essas métricas foram selecionadas porque permitem a avaliação de diferentes propriedades de cada atributo (BIEMAN; KANG, 1995; CHIDAMBER; KEMERER, 1994), tal como CBO e WMC que medem respectivamente o acoplamento e a complexidade. As métricas utilizadas neste trabalho foram as seguintes:

- **Coesão**: *LCOM2 e LCOM3*.
- **Acoplamento**: *CBO, CBO Modified, FANIN, FANOUT*.
- **Complexidade**: *WMC, SCC, MaxNest e EVG*.
- **Herança**: *NOC, DIT e IFANIN*.

Assim, essas métricas de código podem revelar a qualidade dos sistemas antes e depois da remoção de coocorrências de *code smells*.

Passo 4: Realizar a refatoração das coocorrências de code smells. Esse passo tem como objetivo conduzir a remoção de coocorrências de *code smells* identificadas no Passo 2. Dessa forma, foi feito o recrutamento de desenvolvedores que contribuíram para o desenvolvimento de cada sistema de software durante a participação do experimento. Assim, foi enviado um formulário de caracterização para cada desenvolvedor. O formulário é fornecido no Apêndice A. Esse formulário serviu para caracterizar cada participante em relação a educação, experiência com desenvolvimento de software, conhecimento sobre *code smells* e métricas de qualidade e o grau de domínio da linguagem de programação Java. As respostas dos desenvolvedores foram importantes para analisar quais participantes estavam aptos a participar de fato do experimento desse trabalho. A Tabela 7 sumariza as características de cada desenvolvedor do experimento. Todos os desenvolvedores são da mesma empresa (NPI), mas nem todos tinham conhecimento prévio dos sistemas; 3 tinha conhecimento prévio do S1, 4 do S2, 4 do S3, 2 do S4 e 2 do S5. Primeiramente foram alocados os participantes que já conheciam determinado sistema, depois o restante dos participantes foram alocados de modo que o nível técnico (Métricas de Qualidade, *Code Smells* e Java) estivesse equilibrado levando em consideração todos os sistemas.

Tabela 14 – Caracterização dos desenvolvedores

ID	Anos Experiência	Nível Educacional	Métricas Qualidade	Code Smells	Java
P1	3 anos	Graduação	Básico	Básico	Intermediário
P2	6 anos	Graduação	Básico	Básico	Intermediário
P3	4 anos	Graduação	Avançado	Avançado	Avançado
P4	2 anos	Graduação	Intermediário	Intermediário	Avançado
P5	4 anos	Mestrado	Básico	Intermediário	Avançado
P6	4 anos	Graduação	Básico	Básico	Intermediário
P7	2 anos	Graduação	Intermediário	Intermediário	Intermediário
P8	2 anos	Graduação	Básico	Básico	Intermediário
P9	2 anos	Graduação	Intermediário	Intermediário	Avançado
P10	3 anos	Graduação	Básico	Básico	Intermediário
P11	3 anos	Graduação	Básico	Básico	Intermediário
P12	5 anos	Graduação	Intermediário	Intermediário	Avançado
P13	8 anos	Mestrado	Avançado	Avançado	Avançado
P14	4 anos	Graduação	Basico	Básico	Intermediário

Fonte: Elaborado pelo Autor

Após a seleção dos participantes, foi solicitados que eles realizassem a remoção de coocorrências de *code smells* (a nível de classe e a nível de método) nos sistemas através de

refatoração manual. Os desenvolvedores tiveram liberdade para escolher quais coocorrências refatorar. No entanto, eles deveriam informar quais eram os *commits* relacionados a remoção das anomalias. Isso permitiu a análise de código por *commit* e a identificação do número de dias que foram necessário para remover uma determinada coocorrências. Na Seção 5.3 é explicado com mais detalhes como foi feita a remoção das coocorrências.

Passo 5: Documentar as perspectivas dos desenvolvedores durante a remoção das coocorrências de code smells. Nesse passo foi utilizada a técnica do diário para documentar a percepção dos desenvolvedores em relação a remoção de coocorrências de *code smells*. A técnica do diário consiste de um método de coleção de dados na qual os participantes escrevem em um *template* suas atividades diárias sobre qualquer evento que o tenha afetado tanto positivamente quanto negativamente durante a remoção dos *smells*. Esta técnica representa uma maneira de entender o comportamento dos participantes, minimizando a influência dos pesquisadores (FRANÇA *et al.*, 2018). Neste trabalho, os desenvolvedores utilizaram a técnica do diário durante a remoção de coocorrências de *code smells* para gravar informações sobre as seguintes sentenças: (1) *Eu estou atualmente trabalhando na refatoração das seguintes coocorrências:*; (2) *Minhas principais dificuldades na remoção dessas coocorrências são:*; (3) *As coocorrências mais prejudiciais para os sistemas são:*; (4) *Eu escolhi as coocorrências mais prejudiciais acima porque:*; e (5) *Eu estou usando os seguintes métodos de refatoração para remover as coocorrências*. Dessa forma, é possível utilizar a técnica do diário para capturar a percepção dos desenvolvedores no momento da remoção de coocorrências. O template é fornecido no Apêndice D.

Passo 6: Analisar a remoção das coocorrências de code smells e a percepção dos desenvolvedores Após a remoção das coocorrências de *code smells*, foram realizadas novas medições nos atributos internos de qualidade. O objetivo foi verificar se a qualidade do sistema melhorou ou piorou após a remoção das coocorrências via refatoração. Para isso, foi utilizada a mesma estratégia utilizada por (TARWANI; CHUG, 2016), na qual é utilizado o somatório de métricas para cada atributo. Assim, se o valor do somatório das métricas de um determinado atributo **aumenta**, isso significa que o atributo **piora**. Foi feita a medição calculando o valor de cada métrica de cada classe antes e depois dos *commits* de refatoração, relacionados a completa remoção de uma determinada coocorrência.

Por exemplo, foram utilizadas duas métricas para calcular a coesão (ver Tabela 3), foi feita a medição e cálculo do somatório de cada métrica desse atributo na classe que continha

a coocorrência antes da remoção e depois da remoção completa da coocorrência. Após isso foi analisado três possíveis cenários:

1. Se o valor do somatório das duas métricas diminuiu, então a coesão **aumentou/melhorou**.
2. Se o valor do somatório das duas métricas aumentou, então a coesão **diminuiu/piorou**.
3. Se o valor do somatório das duas métricas permaneceu inalterado, então a coesão **não foi alterada**.

Foi utilizada essa mesma abordagem para todas as outras métricas e atributos internos de qualidade.

Finalmente, foram analisas as respostas dos desenvolvedores. Para isso, foram utilizados procedimentos de *Grounded Theory* (STOL *et al.*, 2016). Foram utilizados os procedimentos *open* e codificação axial para analisar os tipos de coocorrências consideradas mais prejudiciais e as principais dificuldades enfrentadas pelos desenvolvedores durante o processo de remoção de coocorrências através da refatoração manual.

5.3 Atividades do Experimento

O estudo experimental foi composto por 4 atividades.

Atividade 1: Seção de Treinamento. Foi conduzido uma sessão de treinamento com todos os participantes sobre conceitos essenciais necessários para participar do estudo como: *code smells* e suas coocorrências, atributos internos de qualidade e métodos de refatoração. Os participantes também foram ensinados a identificar coocorrências de *code smells*. Foram, no total, cerca de 4 horas de treinamento divididas em duas partes:

- **1ª parte do treinamento:** Foram apresentados exemplos práticos de refatoração que poderiam ser aplicadas na remoção de coocorrências de *code smells* e foram apresentadas as definições teóricas de cada um dos conceitos abordados no trabalho.
- **2ª parte do treinamento:** Foram mostrados exemplos brinquedo para que os desenvolvedores realizasse a remoção de *code smells*. Depois foi explicada a técnica do diário e como os desenvolvedores iriam utilizá-la durante as operações de refatoração e remoção de cada coocorrência de *code smell*.

A decisão de fornecer uma sessão de treinamento teve como objetivo aumentar e nivelar o conhecimento dos participantes para tentar reduzir o viés de conhecimento.

Atividade 2: Remoção das coocorrências de code smells através de refatoração.

Foi solicitado aos desenvolvedores para que eles realizassem a completa remoção

das coocorrências de *code smells* através de refatoração manual. Para ajudar os desenvolvedores, foi apresentada uma lista com o nome dos métodos e classes que continham as coocorrências que foram identificadas no Passo 2. Adicionalmente, para cada coocorrência de *code smell* foram criadas *issues* no Github relacionadas às atividades de refatoração. Cada *issue* continha informação sobre a classe e método afetado pela coocorrência. Foram criadas *branches* para cada uma das atividades de refatoração.

Dessa forma, os desenvolvedores tiveram liberdade para escolher suas *issues* e quais coocorrências de *code smell* refatorar. Foram conduzidas reuniões semanais para verificar o progresso das atividades e se os desenvolvedores estavam encontrando algum tipo de dificuldade ou obstáculo durante o processo de refatoração. Além disso, foram criados grupos no aplicativo de mensagens instantâneas *Whatsapp* de cada time alocado a um sistema e isso facilitou o suporte dos pesquisadores quanto ajudou na comunicação das equipes de desenvolvimento de cada sistema analisado.

Apesar da liberdade dos desenvolvedores em escolher as *issues*, foram tomados cuidados para diminuir o viés de escolha dos programadores e no treinamento eles foram alterados para escolherem tipos diferentes de coocorrências para remover. Além disso, foi instruído para que os participantes deixassem claro os *commits* relacionados às atividades de refatoração. Assim, foi possível identificar em cada *commit* qual a coocorrência que estava sendo removida.

Atividade 3: Documentação da perpepção durante a remoção de coocorrências de code smells. Durante todo o processo de remoção de coocorrências de *code smells* via refatoração, os desenvolvedores foram instruídos a documentar suas percepções usando a técnica do diário (FRANÇA *et al.*, 2018). Assim, cada desenvolvedor documentou qual coocorrência ele estava trabalhando naquele momento, qual a coocorrência mais prejudicial e o motivo dessa escolha. Além disso, os desenvolvedores explicaram as principais dificuldades relacionadas a remoção das coocorrências.

Atividade 4: Validação da completa remoção das coocorrências de code smells. Os *commits* foram analisados para verificar se as coocorrências das *code smells* foram completamente removidas pelos desenvolvedores. A análise e revisão dos resultados foram feitas pelos pesquisadores até achar um consenso. Foram analisados os seguintes itens:

1. O impacto da remoção das coocorrências para atributos internos de qualidade (coesão, acoplamento, herança e complexidade).

2. O número de dias que os desenvolvedores demoraram para concluir cada *issue*.

A análise dos sistemas foi realizada projeto por projeto, começando com o sistema S1 e finalizando com o sistema S5. Todo o processo de remoção das coocorrências de *code smells* via refatoração para todos os sistemas durou cerca de 3 meses e envolveu 14 desenvolvedores. Toda análise dos resultados durou cerca de 2 meses para ser finalizada. A Tabela 15 mostra o número de coocorrências de *code smells*, o número de *commits* de refatoração e o número total de *commits* dos sistemas.

Tabela 15 – Número de *commits* de refatoração e número de coocorrêcias removidas

Sistema	# coocorrências	# commits de refatoração	# total de commits
S1	16	92	1597
S2	30	132	1056
S3	12	70	1196
S4	20	106	2471
S5	4	20	111

Fonte: Elaborado pelo Autor

5.4 Resultados e Discussão

Nesta seção são apresentados os principais resultados deste trabalho. Na Seção 5.4.1 é apresentado as coocorrências mais prejudiciais para os atributos internos de qualidade. Na Seção 5.4.2 são apresentadas as coocorrências mais prejudiciais na percepção dos desenvolvedores. Na Seção 5.4.3 são apresentadas as principais dificuldades dos desenvolvedores durante o processo de remoção.

5.4.1 Coocorrências de code smells que são mais prejudiciais para atributos internos de qualidade

A $\mathbf{QP}_1$ foi abordada através da análise do impacto da remoção das coocorrências de *code smells* nos quatro atributos internos de qualidade: acoplamento, coesão, complexidade e herança. A remoção de coocorrências foi realizada através da refatoração manual por 14 desenvolvedores de software. A Tabela 16 mostra a **média** do impacto da remoção de cada uma das coocorrências levando em consideração todos os 5 sistemas estudados neste trabalho. A primeira coluna representa o tipo de coocorrência. As colunas restantes apresentam o impacto que remoção de coocorrências de *code smells* para cada um dos atributos coesão, complexidade, acoplamento e herança, respectivamente. O símbolo ↑ indica um **aumento** no valor do atributo

depois da remoção das coocorrências. O símbolo ↓ indica uma **diminuição** no valor do atributo após a remoção das coocorrências. Por fim, o símbolo – indica que o valor do atributo permaneceu **inalterado**.

É importante observar que se a coesão aumenta (através da diminuição do valor das métricas LCOM e LCOM3), significa que esse atributo foi melhorado porque quanto maior a coesão de uma classe ou método, maior é a qualidade de um sistema. De forma contrária, atributos como complexidade e acoplamento devem se manter o menor possível para indicar melhoria na qualidade do código. De fato, uma alta complexidade pode indicar um código mais difícil de entender e um alto acoplamento pode indicar um código difícil de modificar. Assim, para esses dois atributos, a diminuição significa melhoria na qualidade. Finalmente, um aumento no atributo herança pode significar maior reusabilidade no código e consequentemente uma melhor qualidade. No entanto, é necessário ter cuidado com a herança excessiva. Esse tipo de situação pode levar um alto acoplamento e ser prejudicial para o software (FOWLER, 2018).

Tabela 16 – Impacto da remoção das coocorrências de *code smells* para os atributos de qualidade levando em consideração todos os sistemas

Tipo de Coocorrência	**Coesão**	**Complexidade**	**Acoplamento**	**Herança**
Feature Envy–God Class	↑ 10,51%	↓30,98%	↓21,52%	-
God Class–Shotgun Surgery	↑ 2,59%	↓27,61%	↓19,14%	-
Dispersed Coupling–God Class	↑ 3,16%	↓24,59%	↓20,00%	↑ 3,57%
Feature Envy–Long Method	↓16,17%	↑ 2,95%	↑ 4,99%	-
Intensive Coupling–Long Method	↓19,90%	↑ 12,63%	↓6,81%	-
Dispersed Coupling–Long Method	↓30,80%	↓9,62%	↓15,48%	↑ 5,56%
Dispersed Coupling–Feature Envy	↓39,76%	↓22,12%	↓13,50%	-
Feature Envy–Intensive Coupling	-	-	↓2,59%	-
God Class–Long Method	↑ 19,97%	↓41,59%	↓33,98%	↑ 11,00%

Fonte: Elaborado pelo Autor

Coocorrências que foram removidas e melhoram todos os atributos internos de qualidade. Resultados da Tabela 16 revelam algumas observações interessantes. A remoção do *Dispersed Coupling–God Class* e *God Class–Long Method* melhorou **todos** os quatro atributos internos. De forma mais precisa, a remoção de *Dispersed Coupling–God Class* melhorou a coesão *3,16%*, diminuiu a complexidade em **24,59%** e aumentou a herança em **3,57%**. A remoção de *God Class–Long Method* provocou uma melhoria na coesão por cerca de **19,97%**, a complexidade e o acoplamento tiveram uma diminuição de **41,59%** e **33,98%** respectivamente. A herança obteve um aumento de **11,0%**. É possível perceber que em ambos os casos a herança não obteve um aumento significativo a ponto de existir um caso de herança excessiva.

Essas observações indicam que a remoção dessas coocorrências trazem uma significante melhora da qualidade do sistema ou que a **presença** dessas coocorrências indicam prejuízo para a qualidade do software. Assim, é possível concluir que é conveniente para os desenvolvedores, analistas de qualidade, gerentes de projeto e outros profissionais focarem na remoção dessas coocorrências específicas para a melhoria na qualidade dos sistemas.

Lição 1: A remoção de ***Dispersed Coupling-God Class*** e ***God Class-Long Method*** melhorou todos os quatro atributos (complexidade, acoplamento, herança e coesão), sugerindo que a presença dessas anomalias são bem prejudiciais para a qualidade dos sistemas.

Coocorrências que foram removidas e melhoraram pelo menos três atributos internos de qualidade Ao analisar a Tabela 16 é possível observar que a remoção de algumas coocorrências, como *Feature Envy-God Class* e *God Class–Shotgun Surgery* melhoraram a coesão, complexidade e acoplamento. Com exceção da herança, que permaneceu inalterado. De forma mais específica, a remoção de *Feature Envy-God Class* causou uma melhoria na coesão de **10,51%**, um diminuição de **30,98 %** na complexidade e uma diminuição de **21,52%** no acoplamento. Por outro lado, a remoção de *God Class-Shotgun Surgery* aumentou a coesão em **2,59%**, diminuiu em **27,61%** a complexidade e diminuiu em **19,14%** o acoplamento.

Finalmente, a remoção de *Dispersed Coupling-Long Method* resultou em uma diminuição da complexidade em **9,62%**, do acoplamento em **15,48%** e aumentou a herança em **5,56%**, no entanto, piorou a coesão em cerca de **30,80%**.

É possível perceber que a da remoção de *Feature Envy-God Class*, *God Class–Shotgun Surgery* não melhorou todos os atributos. No entanto, a remoção dessas anomalias trouxe benefícios para coesão, complexidade, acoplamento e não afetou de forma negativa a herança.

Lição 2: A remoção de ***Feature Envy-God Class*** e ***God Class-Shotgun Surgery*** melhorou três atributos internos de qualidade (coesão, acoplamento e complexidade). Isso também indica que essas coocorrências precisam de mais atenção e cuidado por parte dos desenvolvedores.

Dispersed Coupling-Long Method, melhorou três atributos (acoplamento, complexidade e herança), no entanto, piorou a coesão de forma significativa. Essa é uma situação de interessante discussão, pois as pessoas envolvidas no desenvolvimento do sistema devem decidir se é conveniente melhorar três atributos e em consequência piorar a coesão.

Lição 3: A remoção de ***Dispersed Coupling-Long Method*** melhorou a complexidade, acoplamento e a herança. No entanto, piorou a coesão de forma significativa. As pessoas envolvidas no desenvolvimento do sistema devem tomar a decisão se essa coocorrência deve ser removida por completo.

Coocorrências que foram removidas e melhoraram 2 atributos internos de qualidade. A remoção de *Dispersed Coupling–Feature Envy* melhorou 2 atributos. A refatoração dessa coocorrência diminuiu os atributos complexidade em **22,12%** e o acoplamento em **13,50%**, porém piorou a coesão em **39,76%**. Esse caso também é uma situação na qual as pessoas envolvidas no desenvolvimento do sistema devem tomar uma decisão (e.g, analisar a melhoria dos atributos complexidade e acoplamento em detrimento da coesão).

Lição 4: A remoção de ***Dispersed Coupling–Feature Envy*** provocou efeito positivo na complexidade e no acoplamento, porém piorou na coesão. Profissionais da área devem verificar se é conveniente remover essa coocorrência para melhorar a complexidade e acoplamento em detrimento da coesão.

Coocorrências que foram removidas e melhorou 1 atributo interno de qualidade. A refatoração de *Intensive Coupling–Long Method* piorou a coesão em **19,90%**, piorou a complexidade em **12,63%**, porém o acoplamento diminuiu **6,81%**.

Lição 5: A remoção de ***Intensive Coupling–Long Method*** provocou efeito negativo para a coesão e a complexidade. No entanto, a remoção dessa coocorrência trouxe efeito positivo para o acoplamento.

Enquanto que a refatoração de *Feature Envy–Intensive Coupling* trouxe uma diminuição para o acoplamento em **2,59%** e não teve modificação significativa em nenhum outro atributo. Nesse mesmo sentido, a remoção da coocorrência *Feature Envy–Long Method* também não melhorou nenhum atributo interno de qualidade, pois piorou a coesão em **16,17%**, a complexidade em **2,95%** e o acoplamento aumentou em **4,99%**.

Lição 6: A remoção de ***Feature Envy–Long Method*** não trouxe efeito positivo para nenhum atributo de qualidade. Além disso, a remoção de ***Feature Envy–Long Method*** sugere impacto negativo para coesão, complexidade e acoplamento.

A remoção dessa coocorrência não trouxe efeito positivo para nenhum atributo interno de qualidade. No entanto, é necessário considerar que a remoção dessa anomalia pode

trazer um benefício prático que é melhorar o entendimento do código por parte do desenvolvedor. Isso por ser observado a partir da fala de um dos desenvolvedores:

P12: "Tive mais dificuldade e mais trabalho nas coocorrências que possuíam Feature Envy pois tive que mexer em diversos métodos."

Implicações da QP_1. Os resultados encontrados sugerem que as seguintes coocorrências: *Dispersed Coupling-God Class*, *God Class-Long Method*, *Feature Envy-God Class* e *God Class-Shotgun Surgery* são **extremamente prejudiciais** para a qualidade de software e que a remoção dessas coocorrências resulta em uma melhoria na qualidade dos atributos internos de qualidade. Os resultados encontrados também sugerem que a remoção de certas coocorrências de *code smell* melhoram a qualidade de software e confirma os resultados de estudos anteriores na literatura sobre a remoção de coocorrências de *code smells* (YAMASHITA; MOONEN, 2013b; MARTINS *et al.*, 2020; FERNANDES *et al.*, 2017). Além disso, a remoção de coocorrências de tais como *Dispersed Coupling-Feature Envy* e *Intensive Coupling-Long Method* melhoram certos atributos e pioram outros. Isso acontece porque as operações de refatoração de *code smells* podem levar a melhora de determinados atributos internos de qualidade e a piora de outros atributos (LACERDA *et al.*, 2020; ABID *et al.*, 2020). Por exemplo, a utilização do método *Extract Method* pode melhorar a complexidade enquanto piora a coesão (FERNANDES *et al.*, 2020). Assim, os gerentes de projetos e desenvolvedores podem escolher remover ou não essas coocorrências com o objetivo de melhorar determinado atributo interno de qualidade mesmo sabendo que a utilização de determinado método de refatoração pode ocasionar na piora de um outro atributo de qualidade. Um dos critérios de tomada de decisão pode ser a grau de importância que uma determinada classe possui para a equipe de desenvolvimento e assim, escolher melhorar a qualidade do código de classe mesmo que cause certo prejuízo para outras classes do software. Por fim, a remoção de *Feature Envy-Long Method* não melhorou nenhum dos atributos internos, sugerindo que a remoção de determinadas coocorrências podem ter efeitos negativos para a qualidade (MARTINS *et al.*, 2019).

5.4.2 Coocorrências de code smells mais prejudiciais no ponto de vista dos desenvolvedores

A QP_2 foi abordada a partir da análise de *commits* de refatoração das coocorrências e avaliação das respostas dos desenvolvedores que foram escritas utilizando a técnica de diário.

Essas respostas foram escritas diariamente durante o processo de remoção das coocorrências de *code smells* via refatoração. As respostas são fornecidas no Apêndice C.

A Tabela 17 apresenta as coocorrências mais prejudiciais de acordo com a percepção dos desenvolvedores. A primeira coluna mostra o tipo de coocorrência, a segunda coluna lista os desenvolvedores que consideram a coocorrência como mais prejudicial. Por fim, a última coluna sumariza o número de desenvolvedores que consideraram a coocorrência como mais prejudicial.

Tabela 17 – Coocorrências mais prejudiciais de acordo com a percepção dos desenvolvedores

Tipo de Coocorrência	**Desenvolvedor**	**# Desenvolvedores**
Feature–Envy	P1,P5,P6,P11,P12,P13	6
Dispersed Coupling–God Class	P3,P8,P9,P14,P7	5
God Class–Long Method	P2,P4,P10	3

Fonte: Elaborado pelo Autor

As coocorrências mais prejudiciais na percepção dos desenvolvedores. A Tabela 17 permite-nos a observar que as coocorrências consideradas mais prejudiciais para os desenvolvedores foram: *Feature Envy-God Class*, mencionada por cinco desenvolvedores; *Dispersed Coupling–God Class* mencionada por quatro desenvolvedores; e *God Class–Long Method* mencionada por três desenvolvedores. Também é possível observar que o *code smell God Class* está presente nas três coocorrências mencionadas pelos desenvolvedores. Isso sugere que uma coocorrência contendo esse *smell* pode ser considerada prejudicial pelos desenvolvedores. Essa observação foi mencionada por alguns desenvolvedores:

P11: "Feature Envy e God Class é a mais prejudicial e a minha opinião o smell God Class é o mais prejudicial devido a sua dificuldade de remoção."

P5: "A classe que tem God Class precisa de muitas refatoração para diminuir seu tamanho consideravelmente."

Cada vez fica mais claro que o *smell God Class* deve ser evitado em sistemas OO. Isso fica ainda mais evidente a partir dos resultados encontrados por (PALOMBA *et al.*, 2014) e por (TAIBI *et al.*, 2017), na qual os autores sugerem que o *code smell God Class* além de ser considerado o mais prejudicial também é percebido como uma grande ameaça pelos desenvolvedores.

Lição 7: A presença do *smell God Class* em uma coocorrência sugere que essa coocorrência tem mais chance de ser considerada prejudicial pelos desenvolvedores.

Além das coocorrências da Tabela 17 terem sido consideradas como as mais prejudiciais na percepção dos desenvolvedores, a remoção dessas coocorrências trouxe um impacto positivo para os atributos de qualidade. De fato, na Tabela 16 é possível observar que a remoção de *Dispersed Coupling–God Class*, e *God Class–Long Method* melhorou todos os atributos internos de qualidade e a remoção de *Feature Envy–God Class* melhorou os atributos coesão, complexidade e acoplamento (atributo herança permaneceu inalterado). Essas observações sugerem que a presença dessas coocorrências é prejudicial tanto para os atributos internos de qualidade quanto para os desenvolvedores. Assim, essas coocorrências devem ser removidas de um sistema. Além disso, desenvolvedores de software devem prestar atenção para não introduzir essas coocorrências durante o processo de desenvolvimento, já que essas coocorrências tendem a aparecer frequentemente em sistema de software (YAMASHITA *et al.*, 2015; LOZANO *et al.*, 2015; PALOMBA *et al.*, 2018; MARTINS *et al.*, 2020).

Lição 8: Desenvolvedores devem remover e prestar atenção para evitar introduzir as coocorrências *Dispersed Coupling–God Class*, *Feature Envy–God Class* e *God Class–Long Method* durante o desenvolvimento de software.

Inspeção da qualidade de software é prática importante no contexto de dívida técnica para evitar a inserção de *code smells*, por exemplo, é possível garantir que o novo código não terá nenhum *code smell* ou que não terá determinados tipos de *code smells* (FALESSI *et al.*, 2017). Além disso, em Liu *et al.* (2011) os autores sugerem uma estratégia para evitar a influência de *code smells* na qualidade que consiste em fazer a detecção dessas anomalias no tempo certo e em estágios iniciais.

Desenvolvedores de software necessitam realizar monitoramento contínuo de *code smells* afim de evitar que essas anomalias aumentem de forma crítica o esforço de manutenção do software (AVERSANO *et al.*, 2020).

Dessa forma, a inspeção da qualidade, detecção de *code smells* em fases iniciais e monitoramento contínuo são estratégias que podem ser utilizadas pelos desenvolvedores para evitar introduzir coocorrências de *code smells* durante o desenvolvimento de software.

Coocorrências que necessitam de mais tempo para serem removidas. A Tabela 18 mostra o número de dias que foram necessários para remover cada tipo de coocorrência.

A primeira coluna se refere ao tipo de coocorrência. A segunda coluna mostra o número de ocorrências por tipo de coocorrência encontrados ao todo nos sistemas estudados. A terceira coluna mostra o número de *commits* de refatoração para remover determinada coocorrência. A quarta coluna mostra o número total de dias para remover um tipo de coocorrência. Por fim, a quinta coluna mostra a média de dias que levou para remover uma ocorrência por tipo de coocorrência. Por exemplo, levando em consideração todos os 5 sistemas, a coocorrência *Feature Envy-God Class* apareceu 24 vezes e necessitou de 106 *commits de refatoração* e 50 dias para ser completamente removida, e para remover um instância desse tipo de coocorrência, os desenvolvedores demoraram em média cerca de 2,08 dias.

Ao observar a Tabela 18, é possível visualizar que a coocorrência *Feature-Envy–God Class* foi a que levou mais dias para sua remoção, mas é importante destacar que ela foi a que teve mais ocorrências (24) ao longo dos 5 sistemas. Assim, foi utilizada a **média de dias** como uma medida mais precisa para verificar qual tipo de coocorrência demorou mais tempo para ser removida.

O cálculo para computar a média foi o seguinte:

$$\textbf{Média de dias} = \frac{\#Dias}{\#Coocorrencias}$$

Ou seja, o número total de dias para remover por completo um tipo de coocorrência pelo número total de ocorrências de uma determina coocorrência. Por exemplo, a coocorrência *Feature Envy–God Class* demorou 50 dias e apareceu 24 vezes. Colocando na fórmula fica 50/24 = 2,08 dias.

Tabela 18 – Coocorrências mais prejudiciais de acordo com a percepção dos desenvolvedores

Coocorrência	**# Coocorrências**	**# Commits**	**# Dias**	**Média Dias**
Feature Envy–God Class	24	106	50	2,08
God Class–Shotgun Surgery	6	12	12	2
Dispersed Coupling–God Class	13	84	30	2,3
Feature Envy–Long Method	11	50	13	1,18
Intensive Coupling–Long Method	8	46	25	3,12
Dispersed Coupling–Long Method	11	69	17	1,54
Dispersed Coupling–Feature Envy	2	7	2	1
Feature Envy–Intensive Coupling	1	4	1	1
God Class–Long Method	6	42	12	2

Fonte: Elaborado pelo Autor

Em relação ao tempo, foi analisado a média de dias que demorou para cada tipo coocorrência ser removida por completo. A coocorrência com a maior média de dias foi

Intensive Coupling-Long Method com **3,12 dias**, ou seja, foi necessário 3,12 dias para remover 1 das 8 ocorrências de *Intensive Coupling-Long Method*. Isso pode ser considerado como um resultado interessante pois apesar dessa coocorrência ter tido a maior média de dias para ser removida, nenhum desenvolvedor citou essa coocorrência como a mais prejudicial. Outras coocorrência que tiveram altas médias de dias foram: *Dispersed Coupling-God Class* com **2,3 dias** de média, *Feature Envy-God Class* com **2,08 dias** e *God Class-Long Method* com **2 dias** de média. Esses resultados vão ao encontro da percepção dos desenvolvedores, já que essas são as três coocorrências que foram mencionadas como sendo as mais prejudiciais.

Lição 9: As coocorrências *Intensive Coupling-Long Method*, *Dispersed Coupling-God Class*, *Feature Envy-God Class* e *God Class-Long Method* foram as que demoram mais tempo para serem removidas pelos desenvolvedores.

É muito importante observar essas três coocorrências: ***Dispersed Coupling-God Class***, ***Feature Envy-God Class***, ***God Class-Long Method***. Elas tiveram as seguintes características encontradas nesse trabalho:

1. ***Dispersed Coupling-God Class***: A remoção total desse tipo de coocorrência melhorou **todos** os atributos internos de qualidade; foi considerada a **segunda** mais prejudicial na percepção dos desenvolvedores; e foi a **segunda** coocorrência que demorou mais tempo para ser removida.
2. ***Feature Envy-God Class***: A remoção total desse tipo de coocorrência melhorou **três** atributos internos de qualidade; foi considerada como sendo a **mais prejudicial** para os sistemas na opinião dos desenvolvedores; e foi a **terceira** que mais demorou a ser removida.
3. ***God Class-Long Method***: A remoção total desse tipo de coocorrência melhorou **todos** os atributos internos de qualidade; foi considerada a **terceira** coocorrência mais prejudicial na percepção dos desenvolvedores; e a **quarta** que mais demorou para ser removida.

Dessa forma, fica mais claro que é necessário remover ou evitar ao máximo a inserção desses três tipos de coocorrências pelos desenvolvedores.

Implicações da QP_2. Os resultados encontrados sugerem que a presença do *code smell* **God Class** tende a deixar uma coocorrência mais prejudicial para a qualidade e na percepção dos

desenvolvedores. Além disso, desenvolvedores deveriam priorizar a remoção de coocorrências como *Dispersed Coupling-God Class*, *Feature Envy-God Class* and *God Class-Long Method*. Caso o desenvolvimento de software estiver no começo, é recomendado evitar a introdução dessas coocorrências, pois elas são as que mais demoram para serem removidas, juntamente com a coocorrência *Intensive Coupling-Long Method*.

5.4.3 As principais dificuldades enfrentadas pelos desenvolvedores na remoção de coocorrências

A questão $\mathbf{QP}_3$ foi abordada através da análise das respostas dos desenvolvedores que escritas utilizando a técnica de diário. As principais respostas e suas codificações são fornecidas no Apêndice E. Assim, com as respostas coletadas, foi realizada uma análise qualitativa e quatro categorias foram identificadas:

1. *Dificuldade de Entendimento do Código.*
2. *Complexidade de Métodos ou Funções.*
3. *Esforço de Refatoração.*
4. *Grande Quantidade de Código.*

A Tabela 19 descreve as categorias identificadas durante a análise das respostas dos desenvolvedores. A primeira coluna se refere a categoria e a segunda coluna se refere a descrição da categoria.

Tabela 19 – Descrição das categorias

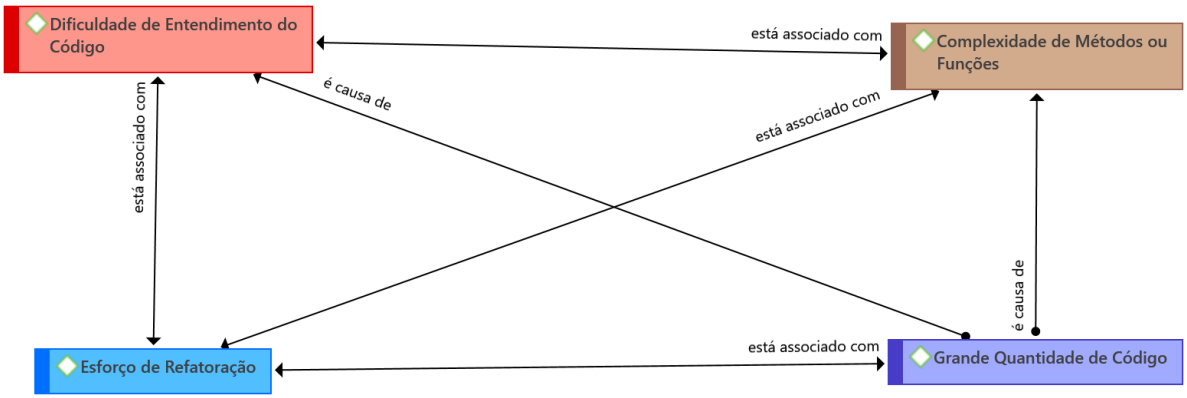
Categoria	**Descrição**
Dificuldade de Entendimento do Código	Se refere a dificuldade de entendimento do código fonte pelo desenvolvedor
Complexidade de Métodos ou Funções	É a definição de métodos ou funções que fazem muitas chamadas a outros métodos no código fonte
Esforço de Refatoração	Se refere ao alto nível de trabalho e retrabalho nas atividades de refatoração do código fonte realizadas pelo desenvolvedor
Grande Quantidade de Código	Se refere a enorme quantidade de código fonte escrito no software seja em uma classe ou em um método

Fonte: Elaborado pelo Autor

A Figura 5.4.3 mostra as categorias e seus relacionamentos com as principais dificuldades enfrentadas pelos desenvolvedores durante a remoção das coocorrências.

Observando a Figura 5.4.3, é possível identificar as categorias e relacionamentos associados com as principais dificuldades enfrentadas pelos desenvolvedores durante a remoção de coocorrências de *code smells*. A associação entre *Dificuldade de Entendimento do Código* e

Figura 4 – Categorias e relacionamentos identificados nas dificuldades dos desenvolvedores



Fonte: Elaborado pelo Autor

a *Complexidade de Métodos ou Funções* é um aspecto a ser destacado neste trabalho, e isso é reforçado por:

P4: "Um método fazendo trabalho de outro, deixa um pouco complicado entender o que está acontecendo."

P8: "As coocorrências que eu estou removendo usa vários métodos e funções de outros classes e afeta a legibilidade da classe e do método e diminui o entendimento do código."

P6: "Elas afetam o entendimento inicial de como o código funciona dado o excesso de métodos chamados em uma classe ou método."

A análise realizada sugere que as coocorrências de *code smells* podem ser causa e aumentar a complexidade de métodos. Assim, um determinado método é chamado ou começa a chamar vários outros métodos. O método afetado pela coocorrência possui estruturas de controle complexas para o desenvolvedor. Todos esses fatores levam a dificuldade de entendimento do código pelos desenvolvedores, e isso é reforçado ao analisar algumas respostas:

P8: "As coocorrências que estou removendo faz a utilização de vários métodos e funções de outras classes e afeta tanto a legibilidade da classe quanto do método afetado e diminui a capacidade de compreensão do código."

P6: "Afetam a compreensão inicial sobre como funciona o código dado o excesso de métodos chamados em uma única classe ou método."

P4:"A maior dificuldade é entender o código, porque para remover as anomalias foi necessário estudar a lógica do código, algumas vezes eu tive dificuldade para identificar onde a anomalia estava acontecendo."

P7:"minha principal dificuldade foi entender o código inicialmente, e o que ele fazia em uma determinada classe que eu tava trabalhando, porque é um grande projeto pra organizar e fazer funcionar corretamente."

P2:"É mais difícil entender por causa que a coocorrência aumenta o acoplamento."

Dessa forma, é possível perceber que a dificuldade de entendimento do código é um fator crucial no momento do desenvolvedor remover coocorrências. Nossas análises sugerem que quantos menos o desenvolvedor entender o código com a coocorrência, maior trabalho ele terá para removê-la. Adicionalmente, o desenvolvedor não se sentirá seguro o suficiente para saber se a solução proposta realmente resolveu o problema. Esse resultado reforça o que já havia sido encontrado previamente por (POLITOWSKI *et al.*, 2020), na qual os autores encontraram que o menor entendimento em códigos com coocorrências levou a um aumento no tempo gasto em tarefas diárias e no esforço.

Lição 10: Uma das principais dificuldades para os desenvolvedores é entender o código fonte que contém as coocorrências de *code smells*.

A categoria de *Esforço de Refatoração* se relaciona com outras duas categorias que são: *Complexidade de Métodos ou Funções* e *Grande Quantidade de Código*. Um exemplo de como Esforço de Refatoração se associa com Grande Quantidade de Código é destacado a seguir:

P11:"As coocorrências fizeram o código ficar muito grande e isso deixou o código complexo para futuras manutenções."

A conexão entre *Esforço de Refatoração* e *Complexidade de Métodos ou Funções* foi destacada no comentário de:

P11:"A coocorrência é prejudicial por chamar vários métodos de outras funções e assim requer muito cuidado na remoção, desde que não interfira no funcionamento do sistema."

Nesse contexto, é possível observar que métodos complexos e grande quantidade de código estão diretamente relacionados a um grande esforço de refatoração durante a remoção de coocorrências de *code smells*. As análises sugerem que a presença de coocorrências de *code smells* pode significar um aumento de linhas no código fonte, ocasionando dessa forma, métodos complexos e um crescimento do esforço de refatoração pelos desenvolvedores (SOARES *et al.*, 2020). Também foi observado o relacionamento entre *Grande Quantidade de Código* e *Complexidade de Métodos e Funções*. Foi encontrado que uma grande quantidade de código pode significar uma maior complexidade de métodos e funções e esse resultado é reforçado por (PALOMBA *et al.*, 2014) e (TAIBI *et al.*, 2017). Assim, os desenvolvedores devem se preocupar com o tamanho do código fonte que eles produzem, pois esse código com coocorrências pode gerar métodos complexos. Essa observação pode ser explicada a partir do seguinte comentário:

P9:"As coocorrências fazem o código ficar grande e desorganizado, fazendo as classes e métodos ficarem muito complexos."

Lição 11: Quanto maior e mais complexo for um método, maior o esforço de refatoração para remover coocorrências.

Desenvolvedores ainda não se sentem seguros para identificar e remover coocorrências de *code smells*.

Um dos resultados encontrados foi que os desenvolvedores ainda possuem inseguranças na remoção e identificação das coocorrências de *code smells*. Em determinados momentos alguns desenvolvedores não tem certeza se removeram a coocorrência completamente ou de forma correta. Isso pode ser observado nas seguintes respostas:

P1:"Eu tenho dificuldade na remoção de coocorrências e verificar se a solução que eu apliquei foi adequada e se realmente resolveu o code smell."

P6:"Em alguns momentos foi difícil identificar onde a anomalia estava acontecendo."

P8:"Eu tenho dificuldade em analisar o que fazer para refatorar a anomalia bem. Decidir qual o melhor método de refatoração em cada situação."

Essas observações sugerem que o desenvolvedor não se sente seguro para aplicar as operações de refatoração e remover completamente a coocorrência de *code smell*. Além disso,

algumas vezes o desenvolvedor não sabe ou não está certo se a solução usada para remover foi a melhor naquele momento, e isso pode afetar negativamente a qualidade do software ao invés de melhorá-la.

Esses resultados confirmam o que já havia sido encontrado por (YAMASHITA; MOONEN, 2013a), na qual os autores fizeram um *survey* com 85 desenvolvedores sobre *code smells* e descobriram o seguinte:

1. 32% dos desenvolvedores não conheciam o conceito de *code smells*.
2. Do grupo restante, 37,50% já tinha ouvindo falar mas não sabiam ao certo do que se tratava.

(TAHIR *et al.*, 2018) também sugere que:

1. Desenvolvedores não sabem ao certo o conceito de *code smells* e nem conhecem o nome de *code smells* específicos.
2. Para os desenvolvedores ainda é confuso a diferença entre *code smells* e antipadrões. Desenvolvedores tendem a achar que são definições semelhantes.

E por fim, (PALOMBA *et al.*, 2014) sugere que:

1. Desenvolvedores inexperientes ainda não conseguem identificar *code smells* e nem se sentem seguros para tomar decisões em relação a melhora da qualidade do software.

Assim, é possível perceber que os resultados encontrados neste trabalho confirmam os resultados encontrados por outros estudos na literatura sobre a falta de conhecimento e a insegurança dos desenvolvedores em relação a *code smells*.

Implicações da QP_3. Os resultados encontrados sugerem que métodos complexos podem atrapalhar o entendimento do código pelo desenvolvedores durante a remoção de coocorrências, significando um maior esforço durante as operações de refatoração. De fato, uma das características das coocorrências é tornar mais difícil o entendimento do código e deixar as atividades de refatoração mais difíceis de serem completadas (POLITOWSKI *et al.*, 2020). Uma grande quantidade de código pode gerar métodos complexos e isso indica um grande esforço de refatoração. Dessa forma, é interessante que os desenvolvedores otimizem seu código o máximo possível durante suas atividades diárias. Além disso, foi possível observar que os desenvolvedores ainda possuem inseguranças na remoção e identificação de *code smells* bem como suas coocorrências, isso confirma os resultados encontrados em

(YAMASHITA; MOONEN, 2013a; PALOMBA *et al.*, 2014; TAHIR *et al.*, 2018). Apesar dos resultados encontrados sobre a percepção dos desenvolvedores durante o processo de remoção das coocorrências de *code smells*, ainda são necessários mais estudos para entender melhor o desenvolvedores durante a remoção das coocorrências de *code smells* (KAUR; DHIMAN, 2019; MELLO *et al.*, 2019; LACERDA *et al.*, 2020)

5.5 Ameaças à validade

Essa Seção discute ameaças à validade do estudo de acordo com a classificação de (WOHLIN *et al.*, 2012).

Validade Interna. Uma das ameaças é o pequeno número de sistemas e linhas de código analisadas nesse estudo. No entanto, o sistemas são de código fechado e foi possível ter um entendimento mais profundo de cada um dos sistemas. Outro problema identificado é que foi feita análise e medição de qualidade em classes de produção. Entretanto, os desenvolvedores estavam mais preocupados com classes de produção do que classes de teste.

Validade de Construção. Foram utilizadas as ferramentas JSpIRIT e JDeodorant para detectar os *code smells* e suas coocorrências. Cada uma das ferramentas possuem suas estratégias de detecção e isso pode ser uma potencial ameaça à validade já que outras ferramentas com outras estratégias de detecção poderiam encontrar diferentes *code smells*. Além disso, foi utilizada a ferramenta Understand que pode coletar um número finito de métricas, logo outra ferramenta pode coletar métricas diferentes da ferramenta Understand. No entanto, essas ferramentas foram escolhidas por causa de sua precisão e por serem amplamente utilizadas. Outra ameaça à validade é que os participantes preencheram o diário de forma subjetiva. Para mitigar essa ameaça, foi explicado em detalhes o objetivo de cada questão durante a sessão de treinamento.

Validade Externa. Os resultados encontrados neste estudo são utilizados para sistemas implementados utilizando a linguagem Java. Outra limitação é que alguns desenvolvedores possuem pouco conhecimento sobre a técnica do diário, *code smells*, métricas de qualidade e refatoração. Para mitigar esse problema, foi conduzido um treinamento sobre esses assuntos para todos os desenvolvedores.

5.6 Conclusão

Com o estudo do Capítulo 4 foi possível ter um entendimento inicial sobre o impacto de coocorrências. No entanto, com o novo estudo deste Capítulo foi possível ampliar o estudo anterior para consolidar os resultados encontrados e obter um novo entendimento sobre quais são as coocorrências de *code smells* mais prejudiciais para atributos internos de qualidade durante a etapa de remoção dessas anomalias através de refatoração e também para os desenvolvedores, levando em consideração suas percepções através de respostas gravadas durante as atividades de refatoração utilizando a técnica do diário.

O presente estudo considerou 6 tipos de *code smells* e suas coocorrências em 5 sistemas OO de código fonte fechado e 4 atributos internos de qualidade (coesão, herança, acoplamento e complexidade). Como principal objetivo do estudo: (i) foi investigado o impacto da remoção dessas coocorrências de *code smells* para atributos internos de qualidade e quais foram as coocorrências mais prejudiciais para os sistemas; (ii) foi identificado as coocorrências mais prejudiciais a partir da perspectiva dos desenvolvedores; e (iii) foi analisada as principais dificuldades e percepções dos desenvolvedores durante a remoção dessas coocorrências. Todo o processo de remoção dessas coocorrências durou 3 meses e ao todo foram feitos 420 *commits* de refatoração e 14 desenvolvedores removerão cerca de 82 coocorrências. Durante todo o processo de remoção de coocorrências de coocorrências via refatoração os desenvolvedores foram instruídos a documentar suas percepções utilizando a técnica do diário.

6 CATÁLOGO DAS COOCORRÊNCIAS DE *CODE SMELLS* MAIS PREJUDICIAS

Com os resultados encontrados neste trabalho e em estudos anteriores da literatura, foi possível gerar e apresentar um catálogo sobre as coocorrências de *code smells*. Esse catálogo é uma abordagem prática direcionado para desenvolvedores de software. No entanto, profissionais da área de computação e pesquisadores também podem utilizar esse catálogo.

6.1 Introdução

O estudo realizado no Capítulo 4 foi importante para fornecer um entendimento inicial sobre o impacto da remoção de coocorrências de *code smells* para atributos internos de qualidade. Em outras palavras, o objetivo do estudo do Capítulo 4 foi entender qual era o tipo de impacto que a remoção de *code smells* causava nos atributos internos de qualidade e se esse impacto se refletia de forma positiva ou negativa para esses atributos. Além disso, com esse estudo foi possível verificar quais as coocorrências de *code smells* mais apareceram durante ciclo de desenvolvimento dos sistemas e quais coocorrências eram mais difíceis de se remover para os desenvolvedores via refatoração. Com os resultados encontrados, foi possível entender que a remoção de coocorrências de *code smells* pode trazer algum benefício na qualidade dos sistemas de software.

O estudo do Capítulo 5 foi realizado com o objetivo de entender melhor os resultados encontrados pelo estudo do Capítulo 4. Assim, no estudo do Capítulo 5 foi feita uma análise mais detalhada do impacto da remoção de cada tipo de coocorrência de *code smell* para os atributos de qualidade. Além de entender melhor a percepção dos desenvolvedores sobre a remoção dessas anomalias. Dessa forma, foi possível entender: (i) quais as coocorrências mais prejudiciais para os atributos internos de qualidade; (ii) quais as coocorrências mais prejudiciais para os desenvolvedores; e, (iii) quais foram as principais dificuldades e percepções dos desenvolvedores durante todo o processo de remoção das coocorrências de *code smells*.

Com os resultados encontrados nos dois trabalhos e em outros trabalhos na literatura, foi possível criar um catálogo sobre as coocorrências de *code smells* com uma abordagem prática e recomendações específicas sobre a remoção dessas anomalias. Nas próximas seções são apresentadas por categoria as recomendações a cerca de da remoção das coocorrências de *code smells*.

6.1.1 Coocorrências de code smells extremamente prejudiciais que devem ser removidas ou evitadas

É fortemente recomendado que os desenvolvedores **priorizem** a remoção ou tenham atenção para não inserir as coocorrências abaixo, por ordem de prioridade:

1. *God Class–Long Method*
2. *Feature Envy–God Class*
3. *Dispersed Coupling–God Class*

Primeiramente, essas coocorrências são consideradas extremamente prejudiciais. Como foi percebido no Capítulo 5, a remoção dessas anomalias trouxe melhorias para pelo menos 3 atributos de qualidade (*Feature Envy–God Class*; para todos os atributos de qualidade (*Dispersed Coupling–God Class*, *God Class–Long Method*; e foram as únicas votadas pelos desenvolvedores como sendo as mais prejudiciais. É possível perceber que o *code smell God Class* está presente nas três coocorrências consideradas mais prejudiciais, isso pode ser explicado porque a presença de *God Class* significa uma grande ameaça para os desenvolvedores (PALOMBA *et al.*, 2014; TAIBI *et al.*, 2017) e esse *smell* impacta de forma negativa a coesão, acoplamento e complexidade (KAUR, 2019).

God Class–Long Method foi considerada como a **mais prejudicial**, por sua remoção ter melhorado todos atributos internos e por ter sido votada como a terceira mais prejudicial pelos desenvolvedores no estudo do Capítulo 5. No estudo do Capítulo 4, *God Class–Long Method* foi a que mais apareceu nos sistemas e a mais difícil de ser removida. Além disso, a presença do *code smell Long Method* pode indicar que o código fonte é difícil de ser entendido, levando a constantes mudanças e introdução de falhas nos sistemas (KAUR, 2019). As anomalias *God Class* e *Long Method* estão entre os *code smells* que mais tendem a aparecer juntos e implicar em diversos prejuízos para a qualidade de software (MARTINS *et al.*, 2019; KAUR; DHIMAN, 2019).

Essas coocorrências estão classificadas em uma ordem de prioridade. Logo, *Feature Envy–God Class* foi considerada como sendo a **segunda** coocorrência mais prejudicial, porque no estudo do Capítulo 5 ela foi a mais votada pelos desenvolvedores confirmando os resultados encontrados por Kaur (2019), que afirmam que a existência de *Feature Envy* e *God Class* juntos merecem maior atenção e prioridade de remoção porque são prejudiciais para desenvolvedores de software. Além disso, foi a coocorrência que mais apareceu em todos os sistemas (24 ocorrências

no total) e no estudo do Capítulo 4 essa coocorrência foi considerada como a segunda mais difícil de refatorar. O *code smell Feature Envy* é difícil de ser identificado pelos desenvolvedores, quebra os princípios de orientação a objetos, causa métodos longos e é muito prejudicial quando acompanhado de outro *code smell* (SINGJAI *et al.*, 2021).

Por fim, a coocorrência *Dispersed Coupling–God Class* foi considerada a **terceira** mais prejudicial, porque no estudo do Capítulo 5 foi a segunda mais votada pelos desenvolvedores e a segunda com mais ocorrências nos sistemas (13 no total). O número de ocorrências de *Dispersed Coupling* em classes ou métodos é bem alto e quando *Dispersed Coupling* aparece juntamente com *God Class* pode ocasionar diversos problemas, especialmente na arquitetura de software (FONTANA *et al.*, 2015).

6.1.2 Coocorrências de code smells prejudiciais que podem ser removidas ou evitadas

A recomendação é que os desenvolvedores tentem remover ou tomem cuidado para não inserir as coocorrências abaixo:

1. *God Class–Shotgun Surgery*
2. *Feature–Envy–Intensive Coupling*

Apesar dessa coocorrência não ter sido citada por nenhum desenvolvedor como a mais prejudicial, foi encontrado no trabalho do Capítulo 5 que a remoção de *God Class–Shotgun Surgery* nos sistemas trouxe benefício para a coesão, complexidade e acoplamento. Além disso, no trabalho do Capítulo 4, *God Class–Shotgun Surgery* foi considerada como a quarta coocorrência mais difícil de se refatorar. Dessa forma, é aconselhável a remoção dessa coocorrência ou cuidado para não inserir essa anomalia no código. Além disso, coocorrências com o *smell God Class* tendem a ser mais prejudiciais. Isso confirma o que já foi encontrado na revisão sistemática realizada por Kaur (2019) que aponta que classes com *Shotgun Surgery* e *God Class* são mais propensas à mudanças, indicando assim, um maior esforço de manutenção. Também foi encontrado por Yamashita e Moonen (2013b), Walter *et al.* (2018) que os *code smells God Class* e *Shotgun Surgery* tendem a aparecer juntos e essas anomalias causam a violação do Princípio de Segregação de Interface.

No trabalho do Capítulo 5, a remoção de *Feature Envy–Intensive Coupling* provocou uma melhoria no acoplamento e não piorou nenhum outro atributo. Também já foi encontrado na literatura por Yamashita *et al.* (2015), que a presença de *Feature Envy* pode provocar o

aparecimento de *Intensive Coupling* e aumentar o acoplamento das classes.

6.1.3 Coocorrências de code smells prejudiciais que quando removidas melhoram os atributos de qualidade em detrimento de outro

A recomendação para as coocorrências dessa categoria é que seja feita uma análise para verificar se é conveniente remover uma coocorrência em busca de melhorar um ou mais atributos em detrimento de outro. De acordo com os resultados do Capítulo 5, foi considerado o número de atributos melhorados para realizar a classificação de prioridade das coocorrências abaixo:

1. *Dispersed Coupling–Long Method*
2. *Dispersed Coupling–Feature Envy*
3. *Intensive Coupling–Long Method*

Por exemplo, a remoção de *Dispersed Coupling–Long Method* tem prioridade nesse categoria porque trouxe melhoria para 3 atributos (complexidade, acoplamento, herança) e piorou a coesão. No estudo do Capítulo 4, ela foi considerada a mais difícil de se remover pelos desenvolvedores. Além disso, a coocorrência *Dispersed Coupling* e *Long Method* tendem a aparecer de forma frequente nos sistemas (WALTER *et al.*, 2018).

A remoção de *Dispersed Coupling–Feature Envy* provocou a melhoria de 2 atributos internos de qualidade (complexidade e acoplamento), porém piorou a coesão. A remoção de *Intensive Coupling–Long Method* piorou a coesão e complexidade, porém melhorou o acoplamento.

Alguns atributos tiveram alguma piora após as operações de refatoração. No entanto, como encontrado por Kaur e Singh (2019), as operações em sistemas industriais possuem menor efeito positivo do que as operações de refatoração empregadas em sistemas acadêmicos. Além disso, operações de refatoração de *code smells* podem levar a melhoria de determinados atributos internos de qualidade e a piora de outros atributos (LACERDA *et al.*, 2020; ABID *et al.*, 2020).

6.1.4 Coocorrências de code smells que podem ser prejudiciais para os desenvolvedores

A recomendação nesse caso, é que seja avaliado se é conveniente fazer a remoção, já que a inserção de coocorrências pode prejudicar o entendimento do código por parte dos

desenvolvedores (POLITOWSKI *et al.*, 2020). Além disso, no Capítulo 4, essa coocorrência foi considerada juntamente com *Feature Envy–God Class* a segunda mais difícil de se remover:

1. *Feature Envy–Long Method*

A remoção dessa coocorrência não trouxe nenhum efeito positivo para nenhum atributo. A remoção dessa anomalia trouxe problemas para a coesão, complexidade e acoplamento. No entanto, a remoção de *Feature Envy* e *Long Method* pode trazer benefícios para outros aspectos relacionados ao desenvolvimento de software, como a redução da propensão à mudanças, diminuição dos erros e melhoria no entendimento do código fonte (KAUR, 2019).

Dessa forma, a decisão de remoção ou evitar inserir essa anomalia no código pode ser uma boa alternativa para melhorar outros aspectos inerentes ao desenvolvimento de software.

A seguir, são feitas recomendações de quais coocorrências de *code smells* remover para melhorar determinado atributo interno de qualidade. As recomendações levam em consideração os resultados encontrados e fornecidos na Tabela 16 do Capítulo 5, além de resultados encontrados em estudos da literatura. As coocorrências estão classificadas em ordem de prioridade de remoção.

6.1.5 Coocorrências de code smells para se remover com o objetivo de melhorar a coesão

A lista das coocorrências que ao serem removidas, melhora (aumenta) a coesão, são as seguintes:

1. *God Class–Long Method*
2. *Feature Envy–God Class*
3. *Dispersed Coupling–God Class*
4. *God Class–Shotgun Surgery*

A coesão deve ser melhorada quando uma classe faz mais do que ela deveria fazer, ou seja, tem muitas responsabilidades que não são inerentes a própria classe (MARTIN, 2000). A remoção das anomalias acima, podem ajudar para que os sistemas tenham classes e métodos mais coesos. É interessante observar que o *code smell God Class* está presente em todas as coocorrências que melhoraram a coesão após serem removidas. Esse resultado reforça o que AlOmar *et al.* (2019) já havia encontrado, ou seja, quando uma classe possui baixa coesão é necessária separá-la em uma ou mais classes para melhorar esse atributo de qualidade. Um dos

principais métodos utilizados para fazer essa operação é o *Extract Method*, que pode separar uma *God Class* em uma ou mais classes (FOWLER, 2018).

6.1.6 Coocorrências de code smells para se remover com o objetivo de melhorar a complexidade

Abaixo é apresentada a lista de coocorrências que ao serem removidas, melhora (diminui) a complexidade:

1. *God Class–Long Method*
2. *Feature Envy–God Class*
3. *God Class–Shotgun Surgery*
4. *Dispersed Coupling–God Class*
5. *Dispersed Coupling–Feature Envy*
6. *Dispersed Coupling–Long Method*

A complexidade do software geralmente é associada a problemas como: decaimento da manutenibilidade do software, maior propensão a erros e redução da qualidade (NEAMTIU *et al.*, 2013; ALENEZI; ALMUSTAFA, 2015). Dessa forma, a remoção das coocorrências acima pode trazer uma diminuição no grau de complexidade dos sistemas. Esses resultados confirmam o que foi encontrado por Lacerda *et al.* (2020), na qual os autores apontam que os *code smells God Class*, *Long Method* e *Feature Envy* afetam a complexidade. Além disso, o *code smell God Class* sugere a presença de *Dispersed Coupling* e *Shotgun Surgery* (YAMASHITA *et al.*, 2015).

6.1.7 Coocorrências de code smells para se remover com o objetivo de melhorar o acoplamento

Abaixo são apresentadas as coocorrências que ao serem removidas, melhora (diminui) o acoplamento:

1. *God Class–Long Method*
2. *Feature Envy–God Class*
3. *Dispersed Coupling–God Class*
4. *God Class–Shotgun Surgery*
5. *Dispersed Coupling–Long Method*

6. *Dispersed Coupling–Feature Envy*
7. *Intensive Coupling–Long Method*
8. *Feature Envy–Intensive Coupling*

Um alto acoplamento pode significar um projeto ruim de software, e pode levar a problemas para modificar e compreender o software, ocasionando uma maior dificuldade em atividades de desenvolvimento (TAUBE-SCHOCK *et al.*, 2011). A remoção das coocorrências acima, pode ajudar a diminuir o acoplamento do software. É possível perceber que a remoção de 8 tipos de coocorrências (9 no total) provocou melhoria nesse atributo de qualidade. Isso confirma o que foi encontrado por Kaur e Singh (2019), que em geral, as operações de refatoração resultam em uma melhoria do acoplamento. É possível perceber o *code smell God Class* está presente nas primeiras quatro coocorrências. Essas coocorrências possuem maior prioridade de remoção pois *God Class* é uma anomalia que impacta diretamente de forma negativa o acoplamento (KAUR; SINGH, 2019). Além disso, os *code smells Intensive Coupling* e *Dispersed Coupling* prejudicam o acoplamento, o que é natural levando em consideração suas definições (LANZA; MARINESCU, 2007; YAMASHITA *et al.*, 2015).

6.1.8 Coocorrências de code smells para se remover com o objetivo de melhorar a herança

Abaixo é apresentada a lista de coocorrências que ao serem removidas, melhora (aumenta) a herança:

1. *God Class–Long Method*
2. *Dispersed Coupling–Long Method*
3. *Dispersed Coupling–God Class*

O aumento da herança provoca uma melhoria na reusabilidade, mas é necessário ter cuidado com a herança excessiva. Apenas a remoção dos *code smells* acima, provocou uma melhoria nesse atributo em até 11%. A remoção de somente três tipos de coocorrências, impactou de forma direta na herança. Isso confirma o que já foi encontrado por Lacerda *et al.* (2020), na qual os autores apontam que é uma tendência que o atributo herança permaneça inalterado após as operações de refatoração em estudos que levam em consideração sistemas industriais. Além disso, a relação da maioria dos *code smells* com o atributo herança, ainda permanece incerto na prática (SINGJAI *et al.*, 2021).

6.2 Implicações práticas do catálogo

As principais implicações práticas do catálogo são as seguintes:

1. Listagem das coocorrências mais prejudiciais para os sistemas e que devem ser evitadas.
2. Direcionamento e recomendação sobre quais coocorrências remover.
3. Apresentação dos efeitos positivos e negativos na remoção de determinado tipo de coocorrência.

Com esse catálogo, é possível verificar quais os principais benefícios na remoção de determinadas coocorrências, como: (i) quais atributos são melhorados ou piorados; (ii) entender quais são as coocorrências mais prejudiciais para a qualidade do software e que devem ser evitadas; e, (iii) direcionamento que auxilia na tomada de decisão sobre qual tipo de coocorrência e mais conveniente de remover através da refatoração.

6.3 Conclusão do Capítulo

Nos Capítulos 4 e 5, foram descritos os estudos com objetivo de entender melhor o impacto que coocorrências de *code smells* podem trazer para a qualidade dos sistemas e para os próprios desenvolvedores. Dessa forma, o catálogo fornece um direcionamento e recomendações que podem ser seguidas sobre a remoção dessas anomalias. Além de levar em consideração resultados de estudos anteriores na literatura, o catálogo foi construído a partir dos seguintes resultados encontrados neste trabalho: (i) as coocorrências *Feature Envy–God Class*; *Dispersed Coupling–God Class* e *God Class-Long Method* são extremamente prejudiciais para a qualidade de software e para os desenvolvedores; (ii) o número de coocorrências de *code smells* tende a aumentar durante o desenvolvimento do sistema; (iii) desenvolvedores têm mais dificuldade para entender códigos contendo coocorrências de *smells*; e, (iv) desenvolvedores ainda possuem inseguranças em relação a identificação e refatoração de *code smells* e suas coocorrências.

7 CONSIDERAÇÕES FINAIS

Ocorrências individuais de *code smells* podem indicar problemas no código fonte. No entanto, coocorrências de *code smells* podem ser melhores indicadores de problemas no código do que ocorrências individuais de *smells* (PAULO *et al.*, 2018). Uma coocorrência de *code smell* consiste em ter um ou mais *code smells* em um determinado método ou em determinada classe (PIETRZAK; WALTER, 2006). Nesse contexto, estudos anteriores na literatura (YAMASHITA; MOONEN, 2013a; OIZUMI *et al.*, 2016; FERNANDES *et al.*, 2017; POLITOWSKI *et al.*, 2020) vêm mostrando que coocorrências de *code smells* trazem prejuízos para a manutenibilidade, *design* de software e compreensão do código pelos desenvolvedores. No entanto, existe uma lacuna na literatura que é a análise da remoção através de refatoração das coocorrências de *code smells* para os atributos internos de qualidade (PAULO *et al.*, 2018) e para os desenvolvedores (PATE *et al.*, 2013; PAULO *et al.*, 2018).

Assim, neste trabalho foi realizada uma análise do impacto da remoção das coocorrências de *code smells* para os atributos internos de qualidade em projetos de código fechado, e quais as coocorrências mais difíceis se refatorar pelos desenvolvedores. Também foi realizada a identificação de quais são as coocorrências de *code smells* mais prejudiciais para os atributos internos de qualidade, como coesão, acoplamento, complexidade, herança e também para os desenvolvedores através da análise das suas percepções durante as atividades de refatoração.

A partir do entendimento inicial da importância de estudar as coocorrências de *code smells*, foi possível responder algumas questões de pesquisa relevantes.

Considerando a primeira questão de pesquisa deste trabalho: "*Qual o impacto das coocorrências de code smells para os atributos internos de qualidade?*", foi possível entender que as coocorrências de *code smells* tendem a aumentar durante o processo de desenvolvimento, ou seja, desenvolvedores tendem a inserir mais *code smells* e consequentemente suas coocorrências desde o início do desenvolvimento. Além disso, também foi encontrado que ao analisar a remoção de coocorrências como um todo, a remoção dessas anomalias trouxe resultado positivo para o atributo complexidade, ou seja, a remoção das coocorrências diminuiu a complexidade nos sistemas de software. Nesse sentindo, também foi feita uma análise sobre quais coocorrências de *code smells* foram mais difíceis de se refatorar ou remover pelos desenvolvedores, esse entendimento é importante para evitar inserir determinados tipos de coocorrências durante o processo de desenvolvimento de software.

No entanto, foi necessário estender e fazer uma análise mais profunda sobre o im-

pacto de cada um dos tipos de coocorrências encontradas em um sistema. Dessa forma, ao responder segunda questão de pesquisa *"Quais os efeitos das coocorrências de code smells mais prejudiciais para os atributos internos de qualidade e para os desenvolvedores?"* é possível entender de uma forma mais detalhada o impacto que um determinado tipo de coocorrência pode trazer para os atributos internos de qualidade de um sistema e para o próprios desenvolvedores. Assim, foi encontrado que existem determinados tipos de coocorrências que são prejudiciais para atributos internos de qualidade, bem como para os próprios desenvolvedores. Esses resultados ajudam a reforçar alguns resultados da primeira questão de pesquisa e a trazer um novo entendimento sobre quais coocorrências de *code smells* remover para trazer benefícios a qualidade do sistema e para os desenvolvedores de software.

Dessa forma, foi possível construir um catálogo prático com direcionamentos e recomendações sobre a remoção de coocorrências de *code smells* além dos principais resultados que foram: (i) as coocorrências *Feature Envy–God Class*; *Dispersed Coupling–God Class* e *God Class-Long Method* são extremamente prejudiciais para a qualidade de software e para os desenvolvedores; (ii) o número de coocorrências de *code smells* tende a aumentar durante o desenvolvimento do sistema; (iii) desenvolvedores têm mais dificuldade para entender códigos contendo coocorrências de *smells*; e, (iv) desenvolvedores ainda possuem inseguranças em relação a identificação e refatoração de *code smells* e suas coocorrências.

7.1 Publicações

Como resultados desse trabalho de mestrado foram publicados três artigos em conferências, apresentados na Tabela 20.

7.2 Trabalhos Futuros

Como trabalhos futuros para continuação deste trabalho, pode-se destacar os seguintes:

– A pesquisa realizada neste trabalho permite a possibilidade de trabalhos futuros que abordem outras tecnologias, plataformas e outras abordagens. É possível ampliar e reproduzir essa pesquisa para um maior número de sistemas com o objetivo de possuir uma maior quantidade de dados para análise do impacto da remoção de coocorrências de *code smells* para os atributos internos de qualidade.

Tabela 20 – Publicações

Publicação	Descrição
MARTINS, Júlio; BEZERRA, Carla; UCHÔA, Anderson. Analyzing the Impact of Inter-smell Relations on Software Maintainability: An Empirical Study with Software Product Lines. In: Proceedings of the XV Brazilian Symposium on Information Systems. 2019. p. 1-8.	Resultado do Trabalho de Conclusão de Curso do autor, publicado durante o mestrado. Esse foi um estudo inicial na identificação de coocorrências de *code smells* em versões de uma Linha de Produto de Software, e análise do impacto dessas coocorrências nos atributos internos de qualidade ao longo da evolução da linha.
MARTINS, Júlio; BEZERRA, Carla; UCHÔA, Anderson; GARCIA, Alessandro. Are Code Smell Co-occurrences Harmful to Internal Quality Attributes? A Mixed-Method Study. In: Proceedings of the XXXIV Brazilian Symposium on Software Engineering (SBES). 2020.	Resultado direto do trabalho da proposta desta dissertação. Esse estudo consistiu em um investigação do impacto das coocorrências de *code smells* nos atributos internos de qualidade de 3 sistemas de código fechado, com a participação de 7 desenvolvedores dos projetos. Esse estudo está descrito no Capítulo 4.
MARTINS, Júlio; BEZERRA, Carla; UCHÔA, Anderson; GARCIA, Alessandro. How do Code Smell Co-occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective. In: Proceedings of the 35th Brazilian Symposium on Software Engineering (SBES). 2021.	Resultado direto do trabalho desta dissertação. Esse estudo consistiu em uma análise das coocorrências de *code smells* mais prejudiciais para atributos internos de qualidade e para os desenvolvedores. Este estudo contou com a participação de 14 desenvolvedores em 5 projetos. Esse estudo está descrito no Capítulo 5.

Fonte: Elaborado pelo Autor

- Outro trabalho futuro importante a se destacar, é a avaliação do impacto da remoção das coocorrências de *code smells* para outros aspectos de qualidade de software, como: manutenibilidade, arquitetura de software, modularidade e compreensão. Ainda existem poucos trabalhos que avaliem o impacto da remoção das coocorrências de *code smells* para aspectos de qualidade (LACERDA *et al.*, 2020).
- Pode-se também reproduzir o estudo com diferentes ferramentas que detectam outros *code smells*. As ferramentas de identificação de *code smells* utilizam estratégias de detecção diferentes (PAIVA *et al.*, 2015), assim é possível encontrar novos tipos de coocorrências de *code smells*.
- Outra possibilidade, é avaliar o impacto da remoção das coocorrências de *code smells* para outras plataformas, como a plataforma de desenvolvimento *mobile*. Já existem trabalhos que avaliam o impacto de *code smells* para o consumo de energia de dispositivos Android (CARETTE *et al.*, 2017; OLIVEIRA *et al.*, 2018; IANNONE *et al.*, 2020). No entanto, até a escrita deste trabalho, não existem estudos que verifiquem o impacto das coocorrências de *code smells* para os dispositivos Android.
- Outro trabalho futuro identificado, seria o desenvolvimento de uma ferramenta que faça a detecção das coocorrências de *code smells* de forma automática. Podendo também fornecer também a opção de refatoração automática dessas anomalias.
- Também é possível entender melhor a partir de técnicas de *machine learning*, como foram inseridos os *code smells* e suas *coocorrências* no código fonte.
- Em relação a percepção dos desenvolvedores, também são necessários mais estudos

que entendam de forma mais profunda o ponto de vista do desenvolvedor durante as atividades de refatoração. Além disso, é possível estender o estudo de (YAMASHITA; MOONEN, 2013a) e verificar se os desenvolvedores estão familiarizados com o conceito de coocorrências de *code smells*.

– Por fim, vislumbra-se como trabalho futuro uma melhor investigação do impacto das coocorrências de *code smells* de uma maneira geral para o desenvolvimento de software, na perspectiva de código aberto.

REFERÊNCIAS

ABBES, M.; KHOMH, F.; GUEHENEUC, Y.-G.; ANTONIOL, G. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: IEEE. **15th CSMR**. [*S. l.*], 2011. p. 181–190.

ABDELMOEZ, W.; KOSBA, E.; IESA, A. F. Risk-based code smells detection tool. In: SOCIETY OF DIGITAL INFORMATION AND WIRELESS COMMUNICATION. **The International Conference on Computing Technology and Information Management (ICCTIM)**. [*S. l.*], 2014. p. 148.

ABID, C.; ALIZADEH, V.; KESSENTINI, M.; FERREIRA, T. d. N.; DIG, D. 30 years of software refactoring research: A systematic literature review. **arXiv preprint arXiv:2007.02194**, 2020.

AGNIHOTRI, M.; CHUG, A. A systematic literature survey of software metrics, code smells and refactoring techniques. **Journal of Information Processing Systems**, v. 16, n. 4, 2020.

AL-QUTAISH, R. E. Quality models in software engineering literature: an analytical and comparative study. **Journal of American Science**, v. 6, n. 3, p. 166–175, 2010.

ALENEZI, M.; ALMUSTAFA, K. Empirical analysis of the complexity evolution in open-source software systems. **International Journal of Hybrid Information Technology**, v. 8, n. 2, p. 257–266, 2015.

ALOMAR, E. A.; MKAOUER, M. W.; OUNI, A. Toward the automatic classification of self-affirmed refactoring. **Journal of Systems and Software**, Elsevier, v. 171, p. 110821, 2021.

ALOMAR, E. A.; MKAOUER, M. W.; OUNI, A.; KESSENTINI, M. On the impact of refactoring on the relationship between quality attributes and design metrics. In: IEEE. **2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [*S. l.*], 2019. p. 1–11.

AVERSANO, L.; CARPENITO, U.; IAMMARINO, M. An empirical study on the evolution of design smells. **Information**, Multidisciplinary Digital Publishing Institute, v. 11, n. 7, p. 348, 2020.

BAVOTA, G.; LUCIA, A. D.; PENTA, M. D.; OLIVETO, R.; PALOMBA, F. An experimental investigation on the innate relationship between quality and refactoring. **Journal of Systems and Software**, Elsevier, v. 107, p. 1–14, 2015.

BIEMAN, J. M.; KANG, B.-K. Cohesion and reuse in an object-oriented system. **ACM SIGSOFT Software Engineering Notes**, ACM New York, NY, USA, v. 20, n. SI, p. 259–262, 1995.

CARETTE, A.; YOUNES, M. A. A.; HECHT, G.; MOHA, N.; ROUVOY, R. Investigating the energy impact of android smells. In: IEEE. **2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [*S. l.*], 2017. p. 115–126.

CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; MELLO, R. de; FONSECA, B.; RIBEIRO, M.; CHÁVEZ, A. 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**. [*S. l.*: *s. n.*], 2017. p. 465–475.

CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D.; GARCIA, A. How does refactoring affect internal quality attributes?: A multi-project study. In: ACM. **31st SBES**. [*S. l.*], 2017. p. 74–83.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Trans. Softw. Eng.**, IEEE, v. 20, n. 6, p. 476–493, 1994.

CHOWDHURY, I.; ZULKERNINE, M. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In: ACM. **Proceedings of the 2010 ACM Symposium on Applied Computing**. [*S. l.*], 2010. p. 1963–1969.

DALLAL, J. A. Object-oriented class maintainability prediction using internal quality attributes. **Inf. Softw. Technol.**, Elsevier, v. 55, n. 11, p. 2028–2048, 2013.

DALLAL, J. A.; ABDIN, A. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. **IEEE Transactions on Software Engineering**, IEEE, v. 44, n. 1, p. 44–69, 2017.

DARCY, D. P.; KEMERER, C. F.; SLAUGHTER, S. A.; TOMAYKO, J. E. The structural complexity of software an experimental test. **IEEE Trans. Softw. Eng.**, IEEE, v. 31, n. 11, p. 982–995, 2005.

DESTEFANIS, G.; COUNSELL, S.; CONCAS, G.; TONELLI, R. Software metrics in agile software: An empirical study. In: SPRINGER. **International Conference on Agile Software Development**. [*S. l.*], 2014. p. 157–170.

DYER, R.; RAJAN, H.; CAI, Y. An exploratory study of the design impact of language features for aspect-oriented interfaces. In: ACM. **11th AOSD**. [*S. l.*], 2012. p. 143–154.

FALESSI, D.; RUSSO, B.; MULLEN, K. What if i had no smells? In: IEEE. **2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [*S. l.*], 2017. p. 78–84.

FENTON, N.; BIEMAN, J. **Software metrics:** a rigorous and practical approach. [*S. l.*]: CRC Press, 2014.

FERNANDES, E.; CHÁVEZ, A.; GARCIA, A.; FERREIRA, I.; CEDRIM, D.; SOUSA, L.; OIZUMI, W. Refactoring effect on internal quality attributes: What haven't they told you yet? **Inf. Softw. Technol.**, Elsevier, p. 106347, 2020.

FERNANDES, E.; OLIVEIRA, J.; VALE, G.; PAIVA, T.; FIGUEIREDO, E. A review-based comparative study of bad smell detection tools. In: ACM. **20th EASE**. [*S. l.*], 2016. p. 18.

FERNANDES, E.; VALE, G.; SOUSA, L.; FIGUEIREDO, E.; GARCIA, A.; LEE, J. No code anomaly is an island. In: SPRINGER. **16th ICSR**. [*S. l.*], 2017. p. 48–64.

FONTANA, F. A.; FERME, V.; ZANONI, M. Towards assessing software architecture quality by exploiting code smell relations. In: IEEE. **2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics**. [*S. l.*], 2015. p. 1–7.

FOWLER, M. **Refactoring:** improving the design of existing code. [*S. l.*]: Addison-Wesley Professional, 2018.

FRANÇA, C.; SILVA, F. Q. D.; SHARP, H. Motivation and satisfaction of software engineers. **IEEE Transactions on Software Engineering**, IEEE, v. 46, n. 2, p. 118–140, 2018.

HENRY, S.; KAFURA, D. Software structure metrics based on information flow. **IEEE Trans. Softw. Eng.**, IEEE, n. 5, p. 510–518, 1981.

IANNONE, E.; PECORELLI, F.; NUCCI, D. D.; PALOMBA, F.; LUCIA, A. D. Refactoring android-specific energy smells: A plugin for android studio. In: **Proceedings of the 28th International Conference on Program Comprehension**. [*S. l.*: *s. n.*], 2020. p. 451–455.

ISO. Iec 25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models. **International Organization for Standardization**, v. 34, p. 2910, 2011.

KAUR, A. A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. **Archives of Computational Methods in Engineering**, Springer, p. 1–30, 2019.

KAUR, A.; DHIMAN, G. A review on search-based tools and techniques to identify bad code smells in object-oriented systems. In: **Harmony search and nature inspired optimization algorithms**. [*S. l.*]: Springer, 2019. p. 909–921.

KAUR, S.; SINGH, P. How does object-oriented code refactoring influence software quality? research landscape and challenges. **Journal of Systems and Software**, Elsevier, v. 157, p. 110394, 2019.

KAUR, S.; SINGH, S. Spotting & eliminating type checking code smells using eclipse plug-in: Jdeodorant. **International Journal of Computer Science and Communication Engineering**, v. 5, n. 1, 2016.

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. A field study of refactoring challenges and benefits. In: **Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering**. [*S. l.*: *s. n.*], 2012. p. 1–11.

KOKOL, P.; KOKOL, M.; ZAGORANSKI, S. Code smells: A synthetic narrative review. **arXiv preprint arXiv:2103.01088**, 2021.

KRISHNAPRIYA, V.; RAMAR, K. Exploring the difference between object oriented class inheritance and interfaces using coupling measures. In: IEEE. **2010 International Conference on Advances in Computer Engineering**. [*S. l.*], 2010. p. 207–211.

LACERDA, G.; PETRILLO, F.; PIMENTA, M.; GUÉHÉNEUC, Y. G. Code smells and refactoring: a tertiary systematic review of challenges and observations. **J. Syst. Softw.**, Elsevier, p. 110610, 2020.

LANZA, M.; MARINESCU, R. **Object-oriented metrics in practice:** using software metrics to characterize, evaluate, and improve the design of object-oriented systems. [*S. l.*]: Springer Science & Business Media, 2007.

LINDEN, F. J. Van der; SCHMID, K.; ROMMES, E. **Software product lines in action:** the best industrial practice in product line engineering. [*S. l.*]: Springer Science & Business Media, 2007.

LIU, H.; MA, Z.; SHAO, W.; NIU, Z. Schedule of bad smell detection and resolution: A new way to save effort. **IEEE transactions on Software Engineering**, IEEE, v. 38, n. 1, p. 220–235, 2011.

LORENZ, M.; KIDD, J. **Object-oriented software metrics:** a practical guide. [*S. l.*]: Prentice-Hall, Inc., 1994.

LOZANO, A.; MENS, K.; PORTUGAL, J. Analyzing code evolution to uncover relations. In: IEEE. **2nd PPAP**. [*S. l.*], 2015. p. 1–4.

MACIA, I.; GARCIA, J.; POPESCU, D.; GARCIA, A.; MEDVIDOVIC, N.; STAA, A. von. Are automatically-detected code anomalies relevant to architectural modularity? In: **11th AOSD**. [*S. l.*: *s. n.*], 2012. p. 167–178.

MACIA, I.; GARCIA, J.; POPESCU, D.; GARCIA, A.; MEDVIDOVIC, N.; STAA, A. von. Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. In: ACM. **11th AOSD**. [*S. l.*], 2012. p. 167–178.

MALHOTRA, R.; CHUG, A. An empirical study to assess the effects of refactoring on software maintainability. In: IEEE. **International Conference on Advances in Computing, Communications and Informatics (ICACCI)**. [*S. l.*], 2016. p. 110–117.

MALHOTRA, R.; JAIN, J. Analysis of refactoring effect on software quality of object-oriented systems. In: SPRINGER. **International Conference on Innovative Computing and Communications**. [*S. l.*], 2019. p. 197–212.

MANTYLA, M.; VANHANEN, J.; LASSENIUS, C. A taxonomy and an initial empirical study of bad smells in code. In: **International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.** [*S. l.*: *s. n.*], 2003. p. 381–384.

MARIANI, T.; VERGILIO, S. R. A systematic review on search-based refactoring. **Information and Software Technology**, Elsevier, v. 83, p. 14–34, 2017.

MARTIN, R. C. Design principles and design patterns. **Object Mentor**, v. 1, n. 34, p. 597, 2000.

MARTINS, J.; BEZERRA, C.; UCHÔA, A.; GARCIA, A. Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. [*S. l.*: *s. n.*], 2020. p. 52–61.

MARTINS, J.; BEZERRA, C.; UCHÔA, A.; GARCIA, A. How do code smell co-occurrences removal impact internal quality attributes? a developers' perspective. 2021.

MARTINS, J.; BEZERRA, C. I. M.; UCHÔA, A. Analyzing the impact of inter-smell relations on software maintainability: An empirical study with software product lines. In: **Proceedings of the XV Brazilian Symposium on Information Systems**. [*S. l.*: *s. n.*], 2019. p. 1–8.

MCCABE, T. J. A complexity measure. **IEEE Trans. Softw. Eng.**, IEEE, n. 4, p. 308–320, 1976.

MELLO, R. de; UCHÔA, A.; OLIVEIRA, R.; OIZUMI, W.; SOUZA, J.; MENDES, K.; OLIVEIRA, D.; FONSECA, B.; GARCIA, A. Do research and practice of code smell identification walk together? a social representations analysis. In: **13th ESEM**. [*S. l.*: *s. n.*], 2019. p. 1–6.

MENS, T.; TOURWÉ, T. A survey of software refactoring. **IEEE Trans. Softw. Eng.**, IEEE, v. 30, n. 2, p. 126–139, 2004.

MORASCA, S. A probability-based approach for measuring external attributes of software artifacts. In: IEEE COMPUTER SOCIETY. **3rd ESEM**. [*S. l.*], 2009. p. 44–55.

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

NEAMTIU, I.; XIE, G.; CHEN, J. Towards a better understanding of software evolution: an empirical study on open-source software. **J. Softw.: Evol. Process**, Wiley Online Library, v. 25, n. 3, p. 193–218, 2013.

NEJMEH, B. A. Npath: a measure of execution path complexity and its applications. **Communications of the ACM**, Association for Computing Machinery, Inc., v. 31, n. 2, p. 188–201, 1988.

NISTALA, P.; NORI, K. V.; REDDY, R. Software quality models: A systematic mapping study. In: IEEE. **2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)**. [*S. l.*], 2019. p. 125–134.

OIZUMI, W.; GARCIA, A.; SOUSA, L. da S.; CAFEO, B.; ZHAO, Y. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: IEEE. **38th ICSE**. [*S. l.*], 2016. p. 440–451.

OIZUMI, W. N.; GARCIA, A. F.; COLANZI, T. E.; FERREIRA, M.; STAA, A. von. When code-anomaly agglomerations represent architectural problems? an exploratory study. In: IEEE. **28th SBES**. [*S. l.*], 2014. p. 91–100.

OLIVEIRA, J.; VIGGIATO, M.; SANTOS, M. F.; FIGUEIREDO, E.; MARQUES-NETO, H. An empirical study on the impact of android code smells on resource usage. In: **SEKE**. [*S. l.*: *s. n.*], 2018. p. 314–313.

PAIVA, T.; DAMASCENO, A.; FIGUEIREDO, E.; SANT'ANNA, C. On the evaluation of code smells and detection tools. **Journal of Software Engineering Research and Development**, Springer, v. 5, n. 1, p. 7, 2017.

PAIVA, T.; DAMASCENO, A.; PADILHA, J.; FIGUEIREDO, E.; SANT'ANNA, C. Experimental evaluation of code smell detection tools. In: ACM. **Workshop on Software Visualization, Evolution and Maintenance (VEM)**. [*S. l.*], 2015. p. 15.

PAIXÃO, M.; UCHÔA, A.; BIBIANO, A. C.; OLIVEIRA, D.; GARCIA, A.; KRINKE, J.; ARVONIO, E. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In: **17th MSR**. [*S. l.*: *s. n.*], 2020.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D. Do they really smell bad? a study on developers' perception of bad code smells. In: **2014 IEEE International Conference on Software Maintenance and Evolution**. [*S. l.*: *s. n.*], 2014. p. 101–110.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; FASANO, F.; OLIVETO, R.; LUCIA, A. D. A large-scale empirical study on the lifecycle of code smell co-occurrences. **Inf. Softw. Technol.**, Elsevier, v. 99, p. 1–10, 2018.

PALOMBA, F.; NUCCI, D. D.; PANICHELLA, A.; ZAIDMAN, A.; LUCIA, A. D. On the impact of code smells on the energy consumption of mobile applications. **Information and Software Technology**, Elsevier, v. 105, p. 43–55, 2019.

PALOMBA, F.; OLIVETO, R.; LUCIA, A. D. Investigating code smell co-occurrences using association rule learning: A replicated study. In: IEEE. **2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)**. [*S. l.*], 2017. p. 8–13.

PARNIN, C.; GÖRG, C.; NNADI, O. A catalogue of lightweight visualizations to support code smell inspection. In: **Proceedings of the 4th ACM symposium on Software visualization**. [*S. l.*: *s. n.*], 2008. p. 77–86.

PATE, J. R.; TAIRAS, R.; KRAFT, N. A. Clone evolution: a systematic review. **Journal of software:** Evolution and Process, Wiley Online Library, v. 25, n. 3, p. 261–283, 2013.

PAULO, E. V. de S.; LUCIA, A. D.; MAIA, M. de A. A systematic literature review on bad smells—5 w's: which, when, what, who, where. **IEEE Trans. Softw. Eng.**, IEEE, 2018.

PIETRZAK, B.; WALTER, B. Leveraging code smell detection with inter-smell relations. **Extreme Programming and Agile Processes in Software Engineering**, Springer, p. 75–84, 2006.

POLITOWSKI, C.; KHOMH, F.; ROMANO, S.; SCANNIELLO, G.; PETRILLO, F.; GUÉHÉNEUC, Y.-G.; MAIGA, A. A large scale empirical study of the impact of spaghetti code and blob anti-patterns on program comprehension. **Information and Software Technology**, Elsevier, v. 122, p. 106278, 2020.

REILLY, B. Social choice in the south seas: Electoral innovation and the borda count in the pacific island countries. **International Political Science Review**, Sage Publications London, v. 23, n. 4, p. 355–372, 2002.

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

SANTOS, J. A. M.; ROCHA-JUNIOR, J. B.; PRATES, L. C. L.; NASCIMENTO, R. S. do; FREITAS, M. F.; MENDONÇA, M. G. de. A systematic review on the code smell effect. **J. Syst. Softw.**, Elsevier, v. 144, p. 450–477, 2018.

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**. [*S. l.*: *s. n.*], 2016. p. 858–870.

SINGH, S.; KAUR, S. A systematic literature review: Refactoring for disclosing code smells in object oriented software. **Ain Shams Engineering Journal**, Elsevier, v. 9, n. 4, p. 2129–2151, 2018.

SINGJAI, A.; SIMHANDL, G.; ZDUN, U. On the practitioners' understanding of coupling smells—a grey literature based grounded-theory study. **Information and Software Technology**, Elsevier, v. 134, p. 106539, 2021.

SJØBERG, D. I.; YAMASHITA, A.; ANDA, B. C.; MOCKUS, A.; DYBÅ, T. Quantifying the effect of code smells on maintenance effort. **IEEE Trans. Softw. Eng.**, IEEE, v. 39, n. 8, p. 1144–1156, 2012.

SOARES, V.; OLIVEIRA, A.; PEREIRA, J. A.; BIBANO, A. C.; GARCIA, A.; FARAH, P. R.; VERGILIO, S. R.; SCHOTS, M.; SILVA, C.; COUTINHO, D. *et al.* On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. [*S. l.*: *s. n.*], 2020. p. 788–797.

STOL, K.-J.; RALPH, P.; FITZGERALD, B. Grounded theory in software engineering research: a critical review and guidelines. In: **Proceedings of the 38th International Conference on Software Engineering**. [*S. l.*: *s. n.*], 2016. p. 120–131.

STROGGYLOS, K.; SPINELLIS, D. Refactoring–does it improve software quality? In: IEEE. **5th WoSQ**. [*S. l.*], 2007. p. 10–10.

SUBRAMANYAM, R.; KRISHNAN, M. S. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. **IEEE Trans. Softw. Eng.**, IEEE, v. 29, n. 4, p. 297–310, 2003.

TAHIR, A.; DIETRICH, J.; COUNSELL, S.; LICORISH, S.; YAMASHITA, A. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. **Information and Software Technology**, Elsevier, v. 125, p. 106333, 2020.

TAHIR, A.; YAMASHITA, A.; LICORISH, S.; DIETRICH, J.; COUNSELL, S. Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow. In: **Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018**. [*S. l.*: *s. n.*], 2018. p. 68–78.

TAIBI, D.; JANES, A.; LENARDUZZI, V. How developers perceive smells in source code: A replicated study. **Information and Software Technology**, Elsevier, v. 92, p. 223–235, 2017.

TARWANI, S.; CHUG, A. Sequencing of refactoring techniques by greedy algorithm for maximizing maintainability. In: IEEE. **Proceedings of the International Conference on Advances in Computing, Communications and Informatics (ICACCI)**. [*S. l.*], 2016. p. 1397–1403.

TAUBE-SCHOCK, C.; WALKER, R. J.; WITTEN, I. H. Can we avoid high coupling? In: SPRINGER. **European Conference on Object-Oriented Programming**. [*S. l.*], 2011. p. 204–228.

TSANTALIS, N.; CHAIKALIS, T.; CHATZIGEORGIOU, A. Jdeodorant: Identification and removal of type-checking bad smells. In: IEEE. **12th CSMR**. [*S. l.*], 2008. p. 329–331.

TSANTALIS, N.; CHAIKALIS, T.; CHATZIGEORGIOU, A. Ten years of jdeodorant: Lessons learned from the hunt for smells. In: IEEE. **25th SANER**. [*S. l.*], 2018. p. 4–14.

UCHÔA, A.; FERNANDES, E.; BIBIANO, A. C.; GARCIA, A. Do coupling metrics help characterize critical components in component-based spl? an empirical study. In: **Proceedings of the 5th Workshop on Software Visualization, Evolution and Maintenance (VEM)**. [*S. l.*: *s. n.*], 2017. p. 36–43.

UCHôA, A.; BARBOSA, C.; OIZUMI, W.; BLENíLIO, P.; LIMA, R.; GARCIA, A.; BEZERRA, C. How does modern code review impact software design degradation? an in-depth empirical study. In: **36th ICSME**. [*S. l.*: *s. n.*], 2020. p. 1 – 12.

VIDAL, S.; OIZUMI, W.; GARCIA, A.; PACE, A. D.; MARCOS, C. Ranking architecturally critical agglomerations of code smells. **Science of Computer Programming**, Elsevier, v. 182, p. 64–85, 2019.

VIDAL, S.; VAZQUEZ, H.; DIAZ-PACE, J. A.; MARCOS, C.; GARCIA, A.; OIZUMI, W. Jspirit: a flexible tool for the analysis of code smells. In: IEEE. **34th SCCC**. [*S. l.*], 2015. p. 1–6.

VIDAL, S. A.; MARCOS, C.; DÍAZ-PACE, J. A. An approach to prioritize code smells for refactoring. **Automated Software Engineering**, Springer, v. 23, n. 3, p. 501–532, 2016.

WALTER, B.; FONTANA, F. A.; FERME, V. Code smells and their collocations: A large-scale experiment on open-source systems. **J. Syst. Softw.**, Elsevier, v. 144, p. 1–21, 2018.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [*S. l.*]: Springer Science & Business Media, 2012.

YAMASHITA, A.; MOONEN, L. Do developers care about code smells? an exploratory survey. In: IEEE. **20th WCRE**. [*S. l.*], 2013. p. 242–251.

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

YAMASHITA, A.; ZANONI, M.; FONTANA, F. A.; WALTER, B. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In: IEEE. **31st ICSME**. [*S. l.*], 2015. p. 121–130.

APÊNDICE A – APÊNDICE A – FORMULÁRIO SOBRE AS ATIVIDADES DE REFATORAÇÃO

Formulário sobre as Atividades de Refatoração preenchido pelos desenvolvedores.

Questão 1. Qual *code smells* você mais refatorou?

( ) *Feature Envy*

( ) *God Class*

( ) *Dispersed Coupling*

( ) *Intensive Coupling*

( ) *Refused parent bequest*

( ) *Shotgun Surgery*

( ) *Long Method*

Questão 2. Baseado na questão anterior, por qual motivo você escolheu os *smells* da questão passada para refatorar?

_____

Questão 3. Em uma escla de 1 a 5, onde 1 é o mais fácil e o 5 o mais difícil, qual o nível de dificuldade que você teve para refatorar cada *code smell*:

( ) *Feature Envy*

( ) *God Class*

( ) *Dispersed Coupling*

( ) *Intensive Coupling*

( ) *Refused parent bequest*

( ) *Shotgun Surgery*

( ) *Long Method*

Questão 4. Baseado na questão anterior, por qual motivo você achou esses *smells* mais difíceis ou fáceis de refatorar?

_____

Questão 5. Quais coocorrências de *code smells* você refatorou:

( ) *Feature Envy–Long Method*

( ) *Dispersed Coupling–Long Method*

( ) *Dispersed Coupling–Feature Envy*

( ) *God Class–Long Method*

( ) *Intensive Coupling–Long Method*

( ) *God Class–Shotgun Surgery*

( ) *Feature Envy–God Class*

( ) *Feature Envy–Refused Parent Bequest*

( ) *Dispersed Coupling–God Class*

( ) *Feature Envy–Shotgun Surgery*

Questão 6. Baseado na questão anterior, por qual motivo você achou essas coocorrências para refatorar?

Questão 7. Em uma escala de 1 a 5, onde 1 é o mais fácil e 5 é o mais difícil, qual o nível de dificuldade que você teve para refatorar cada coocorrência de *code smells*: (Deixe em branco as que você não refatorou)

( ) *Feature Envy–Long Method*

( ) *Dispersed Coupling–Long Method*

( ) *Dispersed Coupling–Feature Envy*

( ) *God Class–Long Method*

( ) *Intensive Coupling–Long Method*

( ) *God Class–Shotgun Surgery*

( ) *Feature Envy–God Class*

( ) *Feature Envy–Refused Parent Bequest*

( ) *Dispersed Coupling–God Class*

( ) *Feature Envy–Shotgun Surgery*

Questão 8. Baseado na questão anterior, por qual motivo você escolheu essas coocorrências como mais difícil ou mais fácil de refatorar?

Questão 9. Indique a técnica de refatoração que você utilizou para refatorar a coocorrência. (por exemplo Move Field, Move Method, Extract Method, Inline Method, Extract Interface, Extract Method, Extract Superclass, Pull Up Field, Pull Up Method, Rename Method, Move Field, Push Down Field, Push Down Method)?

Sobre o Experimento

Questão 10. Quais foram as principais dificuldades durante o experimento?

APÊNDICE B – *APÊNDICE B – FORMULÁRIO DE CARACTERIZAÇÃO*

Formulário de caracterização entregue aos desenvolvedores

Questão 1. Seu nome:

_____

Questão 2. Escreva seu tempo de experiência com desenvolvimento (em anos):

_____

Questão 3. Selecione sua formação:

( ) Graduação

( ) Mestrado

( ) Doutorado

Questão 4. Antes do experimento, você já conhecia o conceito de *code smells*?

( ) Sim

( ) Não

Nível de Conhecimento Classifique seu nível de conhecimento em relação a cada dos tópicos abaixo. Marque uma opção.

Questão 5. Métricas de Qualidade:

( ) Eu nunca tinha ouvido falar

( ) Eu tinha ouvido falar mais nunca apliquei

( ) Eu tenho um conhecimento geral mas nunca apliquei

( ) Eu tenho conhecimento mediano e aplico algumas vezes

( ) Eu tenho um conhecimento profundo e geralmente eu aplico

( ) Eu sou um *expert* no assunto e aplico isso quase todos os dias

Questão 6. *Code Smells*:

( ) Eu nunca tinha ouvido falar

( ) Eu tinha ouvido falar mais nunca apliquei

( ) Eu tenho um conhecimento geral mas nunca apliquei

( ) Eu tenho conhecimento mediano e aplico algumas vezes

( ) Eu tenho um conhecimento profundo e geralmente eu aplico

( ) Eu sou um *expert* no assunto e aplico isso quase todos os dias

Questão 7. Coocorrências de *Code smells*:

( ) Eu nunca tinha ouvido falar

( ) Eu tinha ouvido falar mais nunca apliquei

( ) Eu tenho um conhecimento geral mas nunca apliquei

( ) Eu tenho conhecimento mediano e aplico algumas vezes

( ) Eu tenho um conhecimento profundo e geralmente eu aplico

( ) Eu sou um *expert* no assunto e aplico isso quase todos os dias

Questão 8. Refatoração de código:

( ) Eu nunca tinha ouvido falar

( ) Eu tinha ouvido falar mais nunca apliquei

( ) Eu tenho um conhecimento geral mas nunca apliquei

( ) Eu tenho conhecimento mediano e aplico algumas vezes

( ) Eu tenho um conhecimento profundo e geralmente eu aplico

( ) Eu sou um *expert* no assunto e aplico isso quase todos os dias

Questão 9. Linguagem Java:

( ) Eu nunca tinha ouvido falar

( ) Eu tinha ouvido falar mais nunca apliquei

( ) Eu tenho um conhecimento geral mas nunca apliquei

( ) Eu tenho conhecimento mediano e aplico algumas vezes

( ) Eu tenho um conhecimento profundo e geralmente eu aplico

( ) Eu sou um *expert* no assunto e aplico isso quase todos os dias

Questão 10. Spring Framework:

( ) Eu nunca tinha ouvido falar

( ) Eu tinha ouvido falar mais nunca apliquei

( ) Eu tenho um conhecimento geral mas nunca apliquei

( ) Eu tenho conhecimento mediano e aplico algumas vezes

( ) Eu tenho um conhecimento profundo e geralmente eu aplico

( ) Eu sou um *expert* no assunto e aplico isso quase todos os dias

APÊNDICE C – *APÊNDICE C – ANÁLISE QUALITATIVA QP2*

Resposta dos desenvolvedores para a QP2 do estudo do Capítulo 5.

Participante	**Trecho**	**Coocorrência**
P11	Feature Envy e God Class são o tipo de coocorrência mais prejudiciais e a God Class é a mais prejudicial devido sua dificuldade refatoração	Feature Envy–God Class
P1	Feature Envy e God Class	Feature Envy–God Class
P5	Feature Envy e God Class	Feature Envy–God Class
P13	Na issue que estou tabalhando que é Feature Envy e God Class	Feature Envy–God Class
P6	São Feature Envy e God Class	Feature Envy–God Class
P12	Feature Envy e God Class	Feature Envy–God Class
P3	Dispersed Coupling e God Class	Dispersed Coupling–God Class
P8	Dispersed Coupling e God Class	Dispersed Coupling–God Class
P14	Dispersed Coupling e God Class	Dispersed Coupling–God Class
P9	Dispersed Coupling e God Class pois as duas são prejudiciais	Dispersed Coupling–God Class
P7	Dispersed Coupling e God Class na minha opinião	Disperse Coupling–God Class
P2	God Class e Long Method porque God Class dá muito trabalho refatorar	God Class–Long Method
P10	God Class e Long Method	God Class–Long Method
P4	God Class e Long Method	God Class–Long Method

APÊNDICE D – *APÊNDICE D – TEMPLATE DO DIÁRIO EM PORTUGUÊS*

Template do diário utilizado pelos desenvolvedores durante o experimento. Nesse template os desenvolvedores escreviam suas percepções sobre as atividades de refatoração.

Questão 1. Escreva seu Nome:

_____

Questão 2. Escreva o nome do projeto alocado:

_____

Questão 3. Eu estou atualmente trabalhando na refatoração das seguintes coocorrências:

_____

Questão 4. Minhas principais dificuldades na remoção dessas anomalias são:

_____

Questão 5. As coocorrências mais prejudiciais para os sistemas são:

_____

Questão 6. Eu escolhi as coocorrências mais prejudiciais acima porque:

_____

Questão 7. Eu estou usando os seguintes métodos de refatoração para remover as coocorrências:

_____

Métodos de refatoração mostrados no treinamento:

- Extract Class
- Extract Subclass
- Move Method
- Extract Method
- Extract Super Class
- Replace Inheritance with Delegation

APÊNDICE E – *APÊNDICE E – ANÁLISE QUALITATIVA QP3*

Algumas respostas dos desenvolvedores para a QP3 do estudo do Capítulo 5.

Participante	Trecho	Codificação	Categoria
P11	O código contendo as coocorrências que fazem muitas chamadas de funções de outras classes	Chamadas excessivas de funções	*Complexity of Methods and Functions*
P9	Códigos contendo *Disperse Coupling* por chamar vários métodos e por isso demanda muita cautela na sua remoção	Cuidado da refatoração	Refactoring Effort
P11	pois uma manutenção nesse sistema seria bastante trabalhosa	Manutenção Trabalhosa	Refactoring Effort
P11	identificar as partes sejam retiradas do método para melhorar sua identificação sem mexer em sua funcionalidade	Identificação de trechos do código	Difficulty in Understanding the Source Code
P4	é mais difícil de entender códigos contendo coocorrências	Entendimento do código	Difficulty in Understanding the Source Code
P2	as coocorrências contribuem bastante para o acoplamento tornado trabalhosa a refatoração	Acoplamento e refatoração mais difícil	Refactoring Effort
P4	um método realizando o trabalho de outro	Método mal planejado	Complexity of Methods and Functions
P5	as coocorrências deixam mais complicado entender o que está acontecendo no código a classe que tem o God class dá muito trabalho e tem que refatorar pra diminuir o tamanho	Coocorrências prejudicando a legibilidade	Difficulty in Understanding the Source Code
P3	excesso de métodos chamados na camada de controller	Código execessivo	Large Amount of Source Code
P6	as coocorrências são difíceis pela quantidade de vezes que elas aparecem	Quantidade de coocorrências	Large Amount of Source Code
P11	Na issue #32 as coocorrências fazem utilização de vários métodos e funções de outras classes, já na issue #5 elas deixam o método complexo	Coocorrências e complexidade de métodos	Complexity of Methods and Functions
P8	as coocorrências afetam a legibilidade tanto da classe quanto do método afetado	Falta de legibilidade	Difficulty in Understanding the Source Code
P8	Afeta o entendimento do código e portanto a legibilidade do código da classe	Compreensão do código de uma classe	Difficulty in Understanding the Source Code
P1	é muito difícil de remover essas anomalias	DIficuldade de refatoração	Refactoring Effort
P5	o código fica muito grande com dois smells coocorrendo	Coocorrência de smells	Large Amount of Source Code
P13	dificuldade da remoção dessas coocorrências em um projeto em andamento ou em produção	Remoção em projeto em produção	Refactoring Effort
P14	impacto negativo em manutenções do código	Previsão de dificuldade na manutenção	Refactoring Effort
P12	impacto no mapeamento das classes para realizar as refatorações	Mapeamento das classes	Refactoring Effort
P12	rastreabilidade do código	Entendimento do código	Difficulty in Understanding the Source Code
P9	as coocorrências realmente deixam o código muito grande e desorganizado	Código mal organizado	Large Amount of Source Code
P5	coocorrências deixam os métodos e classes bem complexas deixando difícil de entender	Complexidade de classes e métodos	Difficulty in Understanding the Source Code
P10	coocorrências realmente dificultam a manutenção do código	Dificuldade de Manutenção	Refactoring Effort
P11	Essas coocorrências são mais prejudiciais por conta que muitas pessoas não são acostumadas a refatorá-las e removê-las principalmente as que tem Disperse Coupling	Remoção difícil por falta de conhecimento	Refactoring Effort
P8	porque afeta a legibilidade pois algum code smell junto com Long Method é complicado de analisar	Smell junto com Long Method	Difficulty in Understanding the Source Code
P7	as coocorrências estavam deixando a classe sem sentido e desorganizada pois tinha métodos que não pertenciam a determinada classe	Entendimento da classe	Difficulty in Understanding the Source Code
P9	o acoplamento provocado pelo Disperse Coupling deixa difícil fazer mudança ou manutenção	Acoplamento e refatoração	Refactoring Effort
P1	eu tenho dificuldade na remoção de coocorrências e verificar se a solução que eu dei foi adequada e de fato solucionou o smell	Dificuldade de remoção	Difficulty in Understanding the Source Code
P6	algumas vezes foi difícil entender onde a anomalia estava afetando	Identificação do smell	Difficulty in Understanding the Source Code
P8	eu tive dificuldade no que fazer para refatorar a anomalia e e qual operação de refatoração utilizar em cada situação	Insegurança na refatoração	Difficulty in Understanding the Source Code