Investigando o Impacto das Coocorrências de *Code Smells* nos Atributos Internos de Qualidade

Júlio Serafim Martins[1], **Carla I. M. Bezerra**[1]

[1]Programa de Pós-Graduação em Computação (PCOMP)
Universidade Federal do Ceará (UFC) – Quixadá – CE – Brasil
Ingresso: 02/2019 - Defesa: 09/2021

juliomserafim@gmail.com,carlailane@ufc.br

Abstract. *This work aim 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.*

Resumo. *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: *Co-ocorrências de code smells. Refatoração. Atributos internos de qualidade.*

Eventos CBSoft: *SBES e SBCARS*

1. Introdução

Ao longo da sua evolução, o software sistematicamente sofre alterações que podem levar à deterioração de sua estrutura. Nesse contexto, *code smells* podem indicar problemas relacionados com aspectos da qualidade do código, tal como legilibidade e a modificabilidade. Tendo isso em vista, *code smells* podem causar problemas para os desenvolvedores em atividades na fase de manutenção do software [Fowler 2018].

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 and 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].

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* [de Sobrinho Paulo et al. 2018, Lacerda et al. 2020]. Outra grande lacuna apontada por revisões da literatura [Kaur and Singh 2019, Kaur and 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, de Sobrinho Paulo et al. 2018, Kaur and 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.

Dessa forma, esta dissertação teve 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.

2. Fundamentação teórica

2.1. Coocorrências de *code smells*

Code Smells podem indicar problemas relacionados com aspectos de qualidade do código. Durante seu desenvolvimento, pode ser introduzido em alguns sistemas vários ocorrências de *code smells*, dificultando a refatoração de todas essas anomalias. Uma coocorrência de *code smells* acontece quando existem mais de um *code smells* em uma mesma classe ou método [Pietrzak and Walter 2006]. Para detectar coocorrências de *code smells*, é necessário antes fazer a detecção de ocorrências individuais dessa anomalias. No entanto, é difícil escolher qual *code smell* refatorar em uma coocorrência [Lozano et al. 2015].

A Tabela 1 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 and Walter 2006], realizada no presente trabalho. Neste trabalho os *code smells* foram coletados usando duas ferramentas, JDeodorant [Kaur and Singh 2016] e JSpIRIT [Vidal et al. 2015]. Em seguida, foram identificadas as coocorrências dos *code smells* de forma manual.

Tabela 1. Exemplos de coocorrências de *code smells*

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

2.2. Atributos internos de qualidade

Atributos internos de qualidade, tais como tamanho, coesão e acoplamento podem ser medidos a partir dos artefatos de software. Quantificar atributos internos de qualidade é menos complexo do que fazer a medição de atributos externos, por exemplo, o tamanho de uma classe pode ser medida pela métrica LOC (Linhas de Código) [Morasca 2009]. As métricas propostas por [Chidamber and 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 and 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 e pela ferramenta *Understand*[1] para suportar a coleta das métricas.

3. Trabalhos relacionados

A Tabela 3 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

[1]https://www.scitools.com/

Tabela 2. Métricas dos atributos internos de qualidade [McCabe 1976, Chidamber and Kemerer 1994, Lorenz and Kidd 1994, Destefanis et al. 2014]

Atributos	Métrica	Descrição
Coesão	*Lack of Cohesion of Methods* (LCOM2) [Chidamber and 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 and 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 and 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 and 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 and 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 and 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 and 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 and 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 and 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 and 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 and Kidd 1994]	Número de linhas com comentário. Quanto maior o valor dessa métrica, maior será o tamanho do sistema.
	(CDL) [Lorenz and Kidd 1994]	Número de classes. Quanto maior o valor dessa métrica, maior será o tamanho do sistema.
	Instance Methods (NIM) [Lorenz and Kidd 1994]	Número de métodos de instância. Quanto maior for o valor desta métrica, maior será o tamanho do sistema.

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 na Tabela 3, 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*. Dessa forma, as 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.

Tabela 3. 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 and 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

4. 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.

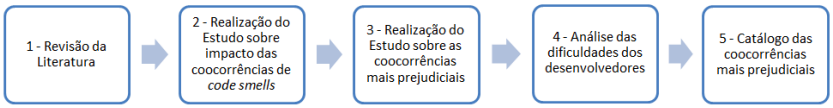

Figura 1. Metodologia de pesquisa

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 na Seção 5, 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 na Seção 6. 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 é 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 que devem ser removidas pelos desenvolvedores.

5. Estudo sobre o impacto das coocorrências de *code Smells* nos atributos internos de qualidade

Esse estudo investigou o impacto das coocorrências de *code smells* nos atributos internos de qualidade. Nós consideramos 7 tipos de *code smells* (*Feature Envy*, *God Class*, *Dispersed Coupling*, *Intensive Coupling*, *Refused Parent Bequest*, *Shotgun Surgery* e *Long Method*) e suas coocorrências em 3 sistemas Java OO de código fechado, e 5 atributos internos de qualidade **(coesão, herança, tamanho, acoplamento e complexidade)** utilizando as métricas da Tabela 2. 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.

5.1. As coocorrências mais frequentes de *code smells*

Existem pelo menos cinco tipos de *code smells* que tendem a formar uma coocorrência: *Feature 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.

5.2. 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 sistemas estudados neste trabalho.

5.3. 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 and Krishnan 2003,

Darcy et al. 2005, Neamtiu et al. 2013, Alenezi and Almustafa 2015]. No sistema S1, a complexidade foi reduzida em 22,94% e no sistema S3 para 21,97%.

5.4. A perspectiva do desenvolvedor sobre a coocorrência de *code smells*

Observa-se que, sob a perspectiva do desenvolvedor, que 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 (4) *Intensive Coupling–Long Method* com uma diferença significativa em relação as coocorrências que estão no topo.

6. Estudo sobre as coocorrências mais prejudiciais para atributos internos de qualidade e percepções dos desenvolvedores

Esta seção 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*. Nesse estudo foram analisados 6 tipos de *code smells* (*Feature Envy*, *God Class*, *Dispersed Coupling*, *Intensive Coupling*. *Shotgun Surgery* e *Long Method*) e suas coocorrências em 5 sistemas OO de código fonte fechado e 4 atributos internos de qualidade **(coesão, herança, acoplamento e complexidade)** utilizando as métricas da Tabela 2. 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 removeram cerca de 82 coocorrências.

6.1. Coocorrências de *code smells* que são mais prejudiciais para atributos internos de qualidade

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 and 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].

6.2. Coocorrências de *code smells* mais prejudiciais no ponto de vista dos desenvolvedores

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*.

6.3. As principais dificuldades enfrentadas pelos desenvolvedores na remoção de coocorrências

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 and Moonen 2013a, Palomba et al. 2014, Tahir et al. 2018].

7. Catálogo das coocorrências de *code smells* mais prejudiciais

Com os resultados encontrados neste trabalho e em estudos anteriores da literatura, foi possível gerar e elaborar um catálogo sobre as coocorrências de *code smells*, direcionado para desenvolvedores de software.

7.1. 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 na Seção 6, 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].

7.2. 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 da Seção 6 que a remoção de *God Class–Shotgun Surgery* nos sistemas trouxe benefício para a coesão, complexidade e acoplamento. Além disso, no trabalho da Seção 5, *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.

No trabalho da Seção 6, 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.

7.3. Coocorrências de *code smells* para se remover com o objetivo de melhorar a coesão

Lista das coocorrências que ao serem removidas, melhora (aumenta) a coesão:

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 *God Class* está presente em todas as coocorrências que melhoraram a coesão após serem removidas. Esse resultado reforça os achados de [AlOmar et al. 2019], que indica que 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].

7.4. Coocorrências de *code smells* para se remover com o objetivo de melhorar a complexidade

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 and 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].

7.5. Coocorrências de *code smells* para se remover com o objetivo de melhorar o acoplamento

Lista de 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 and 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 and 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 and Marinescu 2007, Yamashita et al. 2015].

7.6. Coocorrências de *code smells* para se remover com o objetivo de melhorar a herança

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].

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.

8. Publicações

Como resultados desse trabalho de mestrado foram publicados três artigos em conferências, apresentados na Tabela 4.

Tabela 4. 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 (Seção 5).
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 (Seção 6).

9. Conclusões e Trabalhos Futuros

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. 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 impacto de cada um dos tipos de coocorrências encontradas em um sistema. 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.

Como trabalhos futuros para continuação deste trabalho, pode-se destacar os seguintes: (i) ampliar e reproduzir essa pesquisa para um maior número de sistemas; (ii) avaliar o impacto da remoção das coocorrências de *code smells* para outros aspectos de qualidade de software, como: manutenibilidade, arquitetura de software, modularidade e compreensão; (iii) avaliar o impacto da remoção das coocorrências de *code smells* para outras plataformas, como a plataforma de desenvolvimento *mobile*; e (iv) desenvolver uma ferramenta que faça a detecção e refatoração das coocorrências de *code smells* de forma automática.

Referências

Abbes, M., Khomh, F., Gueheneuc, Y.-G., and Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *15th CSMR*, pages 181–190. IEEE.

Abid, C., Alizadeh, V., Kessentini, M., Ferreira, T. d. N., and Dig, D. (2020). 30 years of software refactoring research: A systematic literature review. *arXiv preprint arXiv:2007.02194*.

Alenezi, M. and Almustafa, K. (2015). Empirical analysis of the complexity evolution in open-source software systems. *International Journal of Hybrid Information Technology*, 8(2):257–266.

AlOmar, E. A., Mkaouer, M. W., Ouni, A., and Kessentini, M. (2019). On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE.

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., and Garcia, A. (2017). How does refactoring affect internal quality attributes?: A multi-project study. In *31st SBES*, pages 74–83. ACM.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493.

Darcy, D. P., Kemerer, C. F., Slaughter, S. A., and Tomayko, J. E. (2005). The structural complexity of software an experimental test. *IEEE Trans. Softw. Eng.*, 31(11):982–995.

de Sobrinho Paulo, E. V., De Lucia, A., and de Almeida Maia, M. (2018). A systematic literature review on bad smells—5 w's: which, when, what, who, where. *IEEE Trans. Softw. Eng.*

Destefanis, G., Counsell, S., Concas, G., and Tonelli, R. (2014). Software metrics in agile software: An empirical study. In *International Conference on Agile Software Development*, pages 157–170. Springer.

Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L., and Oizumi, W. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? *Inf. Softw. Technol.*, page 106347.

Fernandes, E., Vale, G., Sousa, L., Figueiredo, E., Garcia, A., and Lee, J. (2017). No code anomaly is an island. In *16th ICSR*, pages 48–64. Springer.

Fontana, F. A., Ferme, V., and Zanoni, M. (2015). Towards assessing software architecture quality by exploiting code smell relations. In *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*, pages 1–7. IEEE.

Fowler, M. (2018). *Refactoring:* improving the design of existing code. Addison-Wesley Professional.

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

Kaur, A. and Dhiman, G. (2019). 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*, pages 909–921. Springer.

Kaur, S. and Singh, P. (2019). How does object-oriented code refactoring influence software quality? research landscape and challenges. *Journal of Systems and Software*, 157:110394.

Kaur, S. and Singh, S. (2016). Spotting & eliminating type checking code smells using eclipse plug-in: Jdeodorant. *International Journal of Computer Science and Communication Engineering*, 5(1).

Kokol, P., Kokol, M., and Zagoranski, S. (2021). Code smells: A synthetic narrative review. *arXiv preprint arXiv:2103.01088*.

Lacerda, G., Petrillo, F., Pimenta, M., and Guéhéneuc, Y. G. (2020). Code smells and refactoring: a tertiary systematic review of challenges and observations. *J. Syst. Softw.*, page 110610.

Lanza, M. and Marinescu, R. (2007). *Object-oriented metrics in practice:* using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science & Business Media.

Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics:* a practical guide. Prentice-Hall, Inc.

Lozano, A., Mens, K., and Portugal, J. (2015). Analyzing code evolution to uncover relations. In *2nd PPAP*, pages 1–4. IEEE.

Martin, R. C. (2000). Design principles and design patterns. *Object Mentor*, 1(34):597.

Martins, J., Bezerra, C., Uchôa, A., and Garcia, A. (2020). Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, pages 52–61.

Martins, J., Bezerra, C. I. M., and Uchôa, A. (2019). 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*, pages 1–8.

McCabe, T. J. (1976). A complexity measure. *IEEE Trans. Softw. Eng.*, (4):308–320.

Morasca, S. (2009). A probability-based approach for measuring external attributes of software artifacts. In *3rd ESEM*, pages 44–55. IEEE Computer Society.

Neamtiu, I., Xie, G., and Chen, J. (2013). Towards a better understanding of software evolution: an empirical study on open-source software. *J. Softw.: Evol. Process*, 25(3):193–218.

Oizumi, W., Garcia, A., da Silva Sousa, L., Cafeo, B., and Zhao, Y. (2016). Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *38th ICSE*, pages 440–451. IEEE.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., and De Lucia, A. (2018). A large-scale empirical study on the lifecycle of code smell co-occurrences. *Inf. Softw. Technol.*, 99:1–10.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., and De Lucia, A. (2014). Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 101–110.

Pate, J. R., Tairas, R., and Kraft, N. A. (2013). Clone evolution: a systematic review. *Journal of software:* Evolution and Process, 25(3):261–283.

Pietrzak, B. and Walter, B. (2006). Leveraging code smell detection with inter-smell relations. *Extreme Programming and Agile Processes in Software Engineering*, pages 75–84.

Politowski, C., Khomh, F., Romano, S., Scanniello, G., Petrillo, F., Guéhéneuc, Y.-G., and Maiga, A. (2020). A large scale empirical study of the impact of spaghetti code and blob anti-patterns on program comprehension. *Information and Software Technology*, 122:106278.

Santana, A., Cruz, D., and Figueiredo, E. (2021). An exploratory study on the identification and evaluation of bad smell agglomerations. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1289–1297.

Santos, J. A. M., Rocha-Junior, J. B., Prates, L. C. L., do Nascimento, R. S., Freitas, M. F., and de Mendonça, M. G. (2018). A systematic review on the code smell effect. *J. Syst. Softw.*, 144:450–477.

Singjai, A., Simhandl, G., and Zdun, U. (2021). On the practitioners' understanding of coupling smells—a grey literature based grounded-theory study. *Information and Software Technology*, 134:106539.

Subramanyam, R. and Krishnan, M. S. (2003). Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310.

Tahir, A., Yamashita, A., Licorish, S., Dietrich, J., and Counsell, S. (2018). 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*, pages 68–78.

Taibi, D., Janes, A., and Lenarduzzi, V. (2017). How developers perceive smells in source code: A replicated study. *Information and Software Technology*, 92:223–235.

Taube-Schock, C., Walker, R. J., and Witten, I. H. (2011). Can we avoid high coupling? In *European Conference on Object-Oriented Programming*, pages 204–228. Springer.

Vidal, S., Vazquez, H., Diaz-Pace, J. A., Marcos, C., Garcia, A., and Oizumi, W. (2015). Jspirit: a flexible tool for the analysis of code smells. In *34th SCCC*, pages 1–6. IEEE.

Walter, B., Fontana, F. A., and Ferme, V. (2018). Code smells and their collocations: A large-scale experiment on open-source systems. *J. Syst. Softw.*, 144:1–21.

Yamashita, A. and Moonen, L. (2013a). Do developers care about code smells? an exploratory survey. In *20th WCRE*, pages 242–251. IEEE.

Yamashita, A. and Moonen, L. (2013b). Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *35th ICSE*, pages 682–691. IEEE.

Yamashita, A., Zanoni, M., Fontana, F. A., and Walter, B. (2015). Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *31st ICSME*, pages 121–130. IEEE.