Inter-relação Entre *Smells* — Uma Análise de *Large Class*, *Complex Class* e Clone

Elder Vicente de Paulo Sobrinho

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2019

Elder Vicente de Paulo Sobrinho

Inter-relação Entre *Smells* — Uma Análise de *Large Class*, *Complex Class* e Clone

Tese de doutorado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Marcelo de Almeida Maia

Uberlândia
2019

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

P331i 2019

Paulo Sobrinho, Elder Vicente de, 1984

Inter-relação entre smells — Uma análise de large class, complex class e clone [recurso eletrônico] / Elder Vicente de Paulo Sobrinho. - 2019.

Orientador: Marcelo de Almeida Maia.

Tese (Doutorado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.

Modo de acesso: Internet.

Disponível em: http://dx.doi.org/10.14393/ufu.te.2019.1231

Inclui bibliografia.

Inclui ilustrações.

1. Computação. 2. Engenharia de software. 3. Refatoração de software. 4. Software - Manutenção. I. Maia, Marcelo de Almeida, 1969, (Orient.) II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

Angela Aparecida Vicentini Tzi Tziboy – CRB-6/947

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Ata da defesa de TESE DE DOUTORADO junto ao Programa de Pós-graduação em Ciência da Computação da Faculdade de Computação da Universidade Federal de Uberlândia.

Defesa de Tese de Doutorado: PPGCO-02/2019

Data: 25 de março de 2019 Hora de início: 8:00

Discente: Elder Vicente de Paulo Sobrinho Matrícula: 11513CCP003

Título do Trabalho: Inter-relação entre *smells* - Uma análise de *Large Class*, *Complex Class* e Clone

Área De Concentração: Ciência da Computação

Linha de pesquisa: Engenharia de Software

Reuniu-se na sala 1B132, Bloco 1B, Campus Santa Mônica da Universidade Federal de Uberlândia, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-Graduação em Ciência da Computação assim composta: Professores doutores: Fabiano Azevedo Dorça – FACOM/UFU, Edgard Afonso Lamounier Júnior – FEELT/UFU, Cláudio Nogueira Sant'Anna – DCC/UFBA, Eduardo Magno Lages Figueiredo - DCC/UFMG e Marcelo de Almeida Maia – FACOM/UFU, orientador do candidato.

Ressalta-se que o Prof. Dr. Cláudio Nogueira Sant'Anna participou da defesa por meio de vídeo conferência desde a cidade de Salvador - BA e o Prof. Dr. Eduardo Magno Lages Figueiredo da cidade de Belo Horizonte - MG. Os outros membros da banca e o aluno participaram *in loco*.

Iniciando os trabalhos o presidente da mesa Prof. Dr. Marcelo de Almeida Maia apresentou a Banca Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu os conceitos finais.

Em face do resultado obtido, a Banca Examinadora considerou o candidato **aprovado**.

Esta defesa de Tese de Doutorado é parte dos requisitos necessários à obtenção do título de Doutor. O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, legislação e regulamentação interna da Universidade Federal de Uberlândia.

Nada mais havendo a tratar foram encerrados os trabalhos às 11 horas e 20 minutos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Participou por meio de vídeo conferência

Prof. Dr. Cláudio Nogueira Sant'Anna
DCC/UFBA

Participou por meio de vídeo conferência

Prof. Dr. Eduardo Magno Lages Figueiredo
DCC/UFMG

Prof. Dr. Fabiano Azevedo Dorça
FACOM/UFU

Prof. Dr. Edgard Afonso Lamounier Júnior
FEELT/UFU

Prof. Dr. Marcelo de Almeida Maia
FACOM/UFU(Orientador)

Aos meus pais, Manoel e Leonídia e às minhas irmãs Erica e Taíse.
À minha amada Juliana Aparecida Vaz.

Agradecimentos

A Deus,
que sempre está a me olhar, me fortalecer e iluminar os meus passos.
Gratidão eterna, por tudo consquistado em minha vida.

Ao Prof. Marcelo,
meu orientador, pelas oportunidades e desafios que me foram proporcionados, por seu apoio, incentivo e inúmeros conselhos. A você, Marcelo, meu respeito e gratidão.

Aos meus pais, Manoel e Leonídia,
pelo amor, apoio e estímulo que incondicionalmente sempre é fornecido.
Por compreenderem minha ausência. A vocês dedico todo meu esforço.

À minhas irmãs Erica e Taíse,
pelo apoio que sempre me forneceram, e pelos momentos divertidos e alegres.

A minha amada Juliana Vaz,
pelo amor sublime dedicado a mim, por sempre estar ao meu lado me apoiando e por compreender os momentos de ausência.

Ao professor Andrea De Lucia,
pela colaboração e enriquecimento deste trabalho.

A CAPES e a Universidade Federal do Triângulo Mineiro (UFTM) pelo apoio.

Resumo

Bad Smells descrevem possíveis problemas no código, apontando potenciais oportunidades de refatoração. Diversos estudos empíricos evidenciaram que os *bad smells* têm um negativo impacto na compreensão e manutenção dos projetos de software. Consequentemente, a identificação de *bad smells* recebeu atenção de pesquisadores que propuseram várias abordagens de detecção e reestruturação. Contudo, a inter-relação da ocorrência no código dos diferentes tipos de *bad smells* ainda é carente de estudos, especialmente, aqueles focados na quantificação desta inter-relação. Assim, este trabalho descreve e apresenta um estudo empírico sobre a inter-relação dos *smells* Large Class, Complex Class e Duplicate Code. Como um dos resultados da inter-relação, destacamos que existem "padrões de ocorrência" entre estes *smells*, por exemplo: na co-ocorrência dos *smells* Large Class e Complex Class, os clones são predominantemente intra-classe. Também apresentamos um estudo da cronologia de *smells*, que é caracterizada pelo rastreamento da ancestralidade de uma dada entidade que apresenta algum *smell*. Os resultados indicam que, após um certo período inicial de amadurecimento dos sistemas, a proporção de classes sem *smell* é maior do que a quantidade com algum *smell*. Além disso, ao longo do ciclo de vida dos sistemas, diversas classes são excluídas e/ou migram de repositórios. Isso tem impacto nas ferramentas que auxiliam os desenvolvedores, pois elas não consideram esse fenômeno. Além disso, também demonstramos uma métrica capaz de separar as classes em grupos (*"Serão Removidas"* e *"Não serão removidas"*). Por fim, apresentamos algumas situações de *smells* com comportamento cíclico e que, raramente, a simples ocorrência de um dado *smell* se torna uma co-ocorrência. Os resultados apresentados revelaram achados que complementam o estado da arte da literatura. Portanto, destacamos que os achados têm impacto na área de *bad smells*, em particular no desenvolvimento de uma nova geração de ferramentas verificadoras da qualidade do código fonte.

Palavras-chave: Manutenção de Software, Re-engenharia de Software, Bad Smells.

Abstract

Bad smells have been defined to describe potential problems in code, possibly pointing out refactoring opportunities. Several empirical studies have highlighted that *bad smells* have a negative impact on the comprehension and maintainability of a software systems. Consequently, their identification has received recently attention from researchers who have proposed various approaches to detect and restructure them. However, studies on the inter-relationship of occurrences in source code of different types of *bad smells* are still lacking, especially those focused on the quantification of this inter-relationship. Thus, in this work, we describe and present an empirical study on the inter-relation of *smells* Large Class, Complex Class and Duplicate Code. As one of the main results of the inter-relation, we highlight that there are "occurrence patterns" among these *smells*, for example: in the co-occurrence of Large Class and Complex Class, the clones are predominantly intra-class. We also present a study of the chronology of smells, it is characterized by tracing the ancestry of a given entity that exhibits some smell. The results indicate that, after a certain initial period of the systems, the proportion of classes without smell is greater than the quantity with some smell. Furthermore, throughout the systems life cycle, several classes are deleted and/or they migrate from the repositories. This has an impact on the tools that help developers, because they do not consider this phenomenon. In addition, we also demonstrate a metric able to separating classes into groups (*"Removed"* and *"Not Removed"*). Finally, we present some situations that the smells have a cyclic behavior and rarely, the simple occurrence of smell becomes a co-occurrence. The results presented revealed findings that complement the state of the art in the literature. Therefore, the findings have an impact on the area of smells, in particular to the development of a new generation of tools.

Keywords: Software Maintenance, Software Reengineering, Bad Smells.

Lista de ilustrações

Lista de tabelas

Lista de siglas

ACM *Association for Computing Machinery*

AST *Abstract Syntax Tree*

AIC *Akaike Information Criterion*

CDSBP *Class Data Should Be Private*

CSS *Cascading Style Sheets*

McCabe *Cyclomatic Complexity*

CC *Complex Class*

DCG *Duplicate Code Group*

DSL *Domain Specific Language*

DECOR *DEtection & CORrection*

VIF *Variance Inflation Factor*

JDT *Eclipse Java Development Tools*

IC Intervalo de Confiança

LOC *Lines Of Code*

LC *Large Class*

MVC *Model View Controller*

NMD *Number of Methods Declared*

NAD *Number of Attributes Declared*

OBSG *Other Bad Smells Group*

OR *Odds Ratio*

PCA *Principal Component Analysis*

PADL *Pattern and Abstract-level Description Language*

RQ *Research Question*

UML *Unified Modeling Language*

Sumário

Capítulo 1

Introdução

Durante o ciclo de execução das atividades de desenvolvimento de software, os envolvidos (ex. desenvolvedores/arquitetos) deliberadamente ou acidentalmente podem empregar decisões e/ou técnicas subótimas (6). No contexto da Engenharia de Software, o uso de técnicas subótimas é conhecido como *Débito Técnico* (7, 8, 9). A principal característica das ações técnicas caracterizadas como *débito técnico* é a presença dualística e antagônica na forma de realizar algo, sendo que a abordagem adotada resulta na degradação de algum aspecto do software que no futuro deve ser restaurado/melhorado. Nesse contexto, Fowler[1] cita o exemplo: Um desenvolvedor precisa implementar uma funcionalidade ao sistema, ele vê duas maneiras de fazer: (i) na primeira, a implementação do código é rápida, mas tornará difícil a implementação de outras funcionalidades no futuro; (ii) por outro lado, na segunda opção, a implementação é demorada, mas resulta em um projeto que facilita a implementação de outras funcionalidades no futuro. Suryanarayana, Samarthyam e Sharma (7) complementam a definição de *débito técnico* dizendo que, a degradação de algum aspecto do software, resultante da prática subótima, continua aumentando ao longo do tempo em cada mudança no software. Isso compara-se a uma dívida financeira que cresce diariamente com a incidência de juros. Assim, mais tarde, o desenvolvedor compensa a "dívida", pagando um preço mais alto.

Nesse sentido, pesquisas aprimorando o conhecimento sobre *débito técnico* se tornam fundamentais, visto que, estudos anteriores relatam que até 80% do custo total do software é destinado a execução das atividades de manutenção e evolução (10). Além disso, aproximadamente 60% do tempo total gasto para realizar a manutenção do software é utilizado no entendimento do código fonte (10).

Nos componentes da Engenharia de Software, o conceito de *débito técnico* é muito amplo e por isso a literatura apresenta diversos tipos de aplicação, exemplo (11): (i) ***Débito Arquitetural***. Refere-se aos problemas encontrados na arquitetura do projeto, por exemplo, violação da modularidade, que pode afetar os requisitos de arquitetura (ex. desempenho); (ii) ***Débito de Código***. Refere-se aos problemas encontrados no

[1] <https://martinfowler.com/bliki/TechnicalDebt>

código-fonte que podem afetar negativamente a legibilidade do código, dificultando sua manutenção; (iii) ***Débito de Projeto***. Refere-se à dívida que pode ser descoberta analisando o código-fonte identificando o uso de práticas que violem princípios relacionados à programação orientada a objetos; (iv) ***Débito de Documentação***. Refere-se aos problemas encontrados na documentação do projeto de software e pode ser identificado buscando por documentação ausente, inadequada ou incompleta de qualquer tipo; (iv) ***Débito de Build***. Refere-se aos problemas relacionado à criação de *build*, que tornam essa tarefa mais difícil, levando mais tempo/processamento do que o necessário.

Dentre os tipos de débitos técnicos ***Débito de Código*** e ***Débito de Projeto*** (11), têm-se aqueles que são códigos subótimos referenciados na literatura como: *bad smell, code smell, anti-patterns, design smell e code-anomaly* (12, 4, 13, 14). *Bad smell* e *code smell* são os termos usados por Fowler e Beck (15) para descrever 22 estruturas de código subótimas que podem causar efeitos indesejáveis e até mesmo prejudiciais. De forma análoga, mas usando o termo *antipatterns*, Brown et al. (16) descrevem 40 estruturas de código subótimas. As estruturas apresentadas por Fowler e Beck (15) emergem a partir de problemas locais (*low-level*). Por outro lado, Brown et al. (16) descrevem estruturas a partir dos problemas de projeto (*high-level*) (4). *Bad smells* são estruturas candidatas a re-estruturação de sua estrutura interna sem necessariamente alterar seu comportamento externo que, consequentemente, melhora a qualidade do código (15, 16).

Estudos anteriores têm apontado o impacto negativo dos *bad smells* na manutenção de software. Nesse sentido, Aiko e Leon (17) relatam que 27% dos problemas de manutenção estão relacionados aos *bad smells*. Especificamente, Li e Shatnawi (18) relatam que os *bad smells:* Shotgun Surgery, God Class e God Method estão associados à propensão a falhas (*fault-proneness*) de software. De forma análoga, mas do ponto de vista arquitetural, alguns estudos (13, 19) confirmam a relação entre problemas arquiteturais e a ocorrência de *bad smells*. Essa relação fornece indícios da potencial ameaça que os *bad smells* representam, pois problemas arquiteturais podem contribuir para a descontinuidade do software e até mesmo a perda de sua hegemonia (20). Portanto, os *bad smells* podem estar associados a situações indesejadas ou até mesmo causar prejuízos financeiros às empresas. Assim, os *bad smells* podem ser usados como indicadores da qualidade do código, especialmente, direcionando quais ações os desenvolvedores devem adotar (ex.: Reestruturar).

Por outro lado, os *bad smells* nem sempre estão associados a situações indesejadas ou problemáticas (21). Nesse contexto, Rahman, Bird e Devanbu (22) relatam que *bugs* não estão significativamente associados ao *smell* Duplicate Code. Complementarmente, em certas situações, os desenvolvedores introduzem *bad smells* no código fonte porque eles são a melhor opção (23). Por fim, Li e Shatnawi (18) reportam que a ocorrência dos *bad smells* Refused Bequest, Feature Envy e Data Class não exercem impacto sobre a propensão a falhas (*fault-proneness*) de software.

Além do dualismo entre as situações prejudiciais/indesejadas e inofensivas, os *bad smells* também podem ocorrer em grandes quantidades nos projetos de software (24, 25). Em termos práticos, um projeto de software pode apresentar diversos tipos de *smells* (ex.: Duplicate Code, Shotgun Surgery, Data Class, Feature Envy, ...) e, a cada um desses tipos, podem possuir centenas de ocorrências no código. Essas peculiaridades dificultam o controle, a análise e/ou a erradicação dos *smells* que ocorrem nos projetos de software. Complementarmente, Macia et al. (26) mostraram que os problemas de projeto são caracterizados pela existência de vários tipos de *bad smells* que estão espalhados pelo código e, segundo Oizumi (27), alguns destes problemas de projeto de software são revelados/associados a vários *smells* que estão relacionados entre si através das estruturas do código (ex. chamadas de métodos). Nesse contexto, detectar e/ou analisar os *bad smells* de forma isolada (4, 28) limita a utilidade dos mesmos na identificação e/ou resolução de questões complexas (*design problem* (26), *design erosion* (20), etc.).

Diante das questões apontadas acima, há a necessidade de investigar a inter-relação existente entre os diferentes tipos de *bad smells*, pois na ausência de conhecimento das possíveis interações que ocorrem entre os *smells*, os desenvolvedores, poderiam adotar ações equivocadas e/ou desnecessárias ao reestruturar o software. Sugerimos que o estudo da inter-relação dos *smells* pode revelar padrões de codificação. Exemplo hipotético: classes com a co-ocorrência dos *smells* Large Class e Complex Class são esporadicamente refatoradas, além disso, grande parte dos fragmentos de clones dessas classes ocorrem entre entidades que apresentam a co-ocorrência destes mesmos *smells*. Esse possível padrão revela que os desenvolvedores tendem a criar/proliferar clones quando as classes apresentam determinadas características. Conhecendo essa tendência, os engenheiros de software poderiam adotar políticas que reduzam esse comportamento, exemplo: introduzir, durante o processo de revisão do código, uma "checagem" dos clones nas entidades grandes e complexas. Possivelmente isso viabiliza controlar a expansão dos clones, do tamanho e da complexidade.

Dada a importância do conceito de débito técnico e devido a sua amplitude, este trabalho é dedicado ao estudo da inter-relação específica de três tipos de *bad smells*, a saber: Duplicate Code (DC) — *"code fragments similar to one another in syntax and semantics"* (29), Large Class (LC) — *"occurs when a class is trying to do too much, it often shows up as too many instance variables"* (15) e Complex Class (CC) — *"a class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOC"* (14). Esses *smells* foram escolhidos porque: (i) Duplicate Code é um *smell* amplamente estudado, entretanto, poucos artigos investigaram empiricamente a interação dele com outros *smells*. Fowler e Beck (15) e outros trabalhos, já descreveram e conjecturaram uma possível relação desse *smell* com *smells* relacionados às estruturas grandes e/ou complexas. Entretanto não existem estudos empirícos que investiguem esta questão; (ii) semanticamente parece existir uma possível relação entre os *smells* supracitados, por

exemplo, entidades complexas com muitas estruturas de desvio de fluxo, teriam maior chance de apresentar códigos clonados, pois os idiomas existentes em algoritmos poderiam ser reaproveitados por meio de copiar-colar, quando não fossem modularizados em funções; (iii) esses *smells* são bem conhecidos pelos desenvolvedores/pesquisadores e estão bem consolidados na literatura. Consequentemente, isso sugere que esses *smells* podem ser relevantes para manter a qualidade e/ou manutenabilidade do código; (iv) os *smells* em questão são comumente encontrados no código fonte de diversos projetos e a quantidade de instâncias deles é suficientemente grande para permitir uma análise estatística que permita ter evidências mais substanciadas; (v) a literatura apresenta alguns estudos que usam possíveis inter-relações de *smells* para priorizar a refatoração (ex. Liu et al. (30)), contudo, não encontramos estudos empíricos quantificando essas inter-relações de *smells*.

1.1 Objetivos

O objetivo geral deste trabalho consiste em efetuar uma análise empírica quantitativa e qualitativa da inter-relação dos *smells* Duplicate Code, Large Class e Complex Class para estabelecer possíveis implicações de uma eventual inter-relação na melhoria da qualidade do código. Esta análise restringe-se a projetos de software desenvolvidos na linguagem Java, em especial nas classes que são acometidas pela ocorrência isolada e/ou conjunta dos *smells* em questão.

Para atingir nossa meta, três objetivos específicos foram estabelecidos. Esses forneceram diretrizes para o planejamento e execução do estudo. A seguir, os mesmos são apresentados:

- Obter um mapeamento dos estudos científicos que investigaram a inter-relação de *smells* para encontrar lacunas que carecem de posterior investigação.

- Identificar diferentes combinações dos *smells* citados, bem como verificar formas de associações entre eles.

  Um dado *smell* pode se apresentar em diferentes intensidades. Por exemplo, uma Large Class normalmente é identificada por meio de um limiar de tamanho. Assim, as classes podem ser grandes, por ultrapassarem o limiar com uma margem muito pequena ou por uma margem muito ampla. Logo, pretende-se avaliar a ocorrência de diferentes combinações de intensidade dos referidos *smells*. Complementarmente, uma classe pode apresentar um ou mais tipos de *smells*. Portanto, para estudar a inter-relação de *smells* é necessário explorar essa diversidade através de um estudo que considere diferentes combinações, como por exemplo: comparando se há diferença entre as classes em que o *smell* Large Class ocorre isoladamente daquelas entidades em que esse *smell* co-ocorre com o *smell* Complex Class.

- ❑ Indicar implicações na melhoria da qualidade do código, extraídas de combinações da ocorrência de *smells*. As implicações devem ser a principal entrega do estudo em termos de resultado prático, pois idealmente busca-se alternativas efetivas para a prática de melhoria de qualidade de código.

1.2 Contribuições

Esta seção apresenta as principais contribuições do presente trabalho de doutorado:

1. Uma ampla revisão sistemática da literatura sobre o tema *bad smell*. Essa revisão demonstrou novos horizontes de pesquisa, exemplo: O estudo empírico da inter-relação de *smells* é deficiente e necessita de aprofundamento.

2. Um estudo empírico com a quantificação e análise da inter-relação entre os *smells* Large Class, Complex Class e Duplicate Code, que demonstrou a existência de "padrões de codificação" entre esses *smells*, exemplo: na co-ocorrência dos *smells* Large Class e Complex Class, os clones são predominantemente intra-classe.

3. Um estudo da cronologia de *smells* que revelou estratégias para a criação de uma nova geração de ferramentas verificadoras da qualidade do código. Estratégias essas baseadas em padrões temporais de codificação dos desenvolvedores. A exemplo, usando a métrica de *"estabilidade média das transições"*, é possível construir um classificador que indique quais classes devem passar pelo processo de reestruturação que resulte na sua própria remoção do sistema. Reduzindo e/ou evitando múltiplas atividades de desenvolvimento, ou seja, efetuar uma alteração na classe que no futuro sofrerá reestruturação que causará sua remoção.

Enunciado da Tese

A inter-relação dos *smells* Duplicate Code, Large Class e Complex Class pode ser usada para identificar padrões de codificação adotado pelos desenvolvedores. Adicionalmente, esses padrões contém informação relevante que poderia ser usada pelos engenheiros de software para controlar a qualidade do código dos projetos de software, monitorando e gerenciando a expansão dos clones, o tamanho e a complexidade das classes.

1.3 Organização do Texto

O restante desta tese está organizada da seguinte forma: Capítulo 2 apresenta a fundamentação teórica sobre o tema *bad smell*; Capítulo 3 relata uma revisão sistemática da literatura sobre o tema *bad smell*; Capítulo 4 descreve e apresenta o estudo empírico

sobre a inter-relação dos *smells* Large Class (LC), Complex Class (CC) e Duplicate Code (DC); Capítulo 5 relata um estudo empírico da cronologia de *smells* e suas co-ocorrências; Por fim, o Capítulo 6 expressa a conclusão deste trabalho.

Capítulo 2

Fundamentação Teórica

Durante o ciclo de vida, um projeto de software passa por mudanças contínuas com o objetivo de ser aprimorado e para incorporar determinadas funcionalidades (31). Infelizmente, essas mudanças são executadas por desenvolvedores que devem atender aos apertados prazos determinado pelo mercado e/ou clientes. Consequentemente a qualidade do código fonte é muitas vezes negligenciada, aumentando o risco de introduzir *bad smells* (ou simplesmente *code smells* ou *smells*), ou seja, sintomas de possíveis problemas de projeto que se manifestam no código fonte (15). Este fenômeno também é conhecido como "*technical debt*" (8), sendo um débito que o desenvolvedor tem com a organização do sistema. A curto prazo, os débitos técnicos podem trazer benefícios, tais como, maior produtividade e/ou menor tempo entre *releases*, contudo, a longo prazo esses podem resultar em esforço extra na manutenção do código. A exemplo desse tipo de situação, os desenvolvedores podem adicionar várias responsabilidades a uma classe sem observar a necessidade de incluí-las em classes distintas. Como resultado, a classe cresce rapidamente e devido às responsabilidades adicionadas a classe se torna muito complexa e a qualidade do código se deteriora. Essas classes são conhecidas como Large Class (15) e podem causar um problema de projeto no código fonte (ex. *anti-pattern*[1]). Fowler e Beck (15), Brown et al. (16) definiram um catálogo com mais de 30 *bad smells*, sendo que para cada um deles, eles reportam a definição bem como as operações de refatoração destinadas a removê-los.

Hipóteses de que os problemas de projeto (ex. *bad smells*) apresentam impacto negativo na compreensão e manutenção do software, na última década, foram alvo de estudos com o objetivo de analisar empiricamente seu impacto na manutenção de software. Atualmente, há evidências empíricas de que o código que contém *bad smell* é significativamente mais propenso a alterações do que os outros (33). Além disso, o código que participa de *anti-patterns*/*bad smells* tem maior propensão à falha (33, 18).

[1] Muitas vezes os termos *bad smells* e *anti-pattern* são usados como sinônimos. No entanto, o cenário relatado demonstra que *bad smells* denotam algo que "provavelmente" está errado no código. Por outro lado, *anti-pattern* é certamente um problema de projeto existente no código fonte (32).

Estudos destinados a analisar o impacto dos *bad smells* na compreensão do código (34) demonstram que a presença de *smells* não diminui o desempenho dos desenvolvedores, apesar de que a combinação deles resulta em uma diminuição significativa do desempenho (34, 35). Embora os resultados indiquem que a ocorrência isolada de *bad smells* não sejam prejudiciais, eles também revelam que os *smells* são bem difusos nos softwares e, muitas vezes, seus componentes são afetados por mais de um *smell*. Além disso, há evidência empírica de que a quantidade de *smells* nos projetos de software aumenta ao longo do tempo, mas há poucos casos em que eles são removidos através de operações de refatoração (36, 37).

Estes achados sugerem que os *bad smells* e/ou *anti-patterns* devem ser cuidadosamente detectados e monitorados e, sempre que necessário, as operações de refatoração devem ser planejadas e meticulosamente executadas. Infelizmente, a identificação e a correção deste tipo de problema não é trivial, especialmente em grandes projetos de software. Para melhorar o entendimento deste trabalho e para facilitar a compreensão, apresentamos neste capítulo os principais conceitos relacionados ao tema *bad smell*, bem como uma visão detalhada das ferramentas usadas para detectá-los.

2.1 Principais *Bad Smells* da Literatura

Segundo Fowler e Beck (15), *smells* são estruturas no código que sugerem a possibilidade de refatoração. Por outro lado, Palomba et al. (38), sugerem que *smells* são *"symptoms of poor design and implementation choices"*. Em geral, a literatura apresenta cerca de 104 tipos de *bad smells* (39), sendo que alguns não possuem técnicas ou ferramentas que possam ser usadas na identificação de suas instâncias, como exemplo temos o *smell* Poltergeists proposto por Brown et al. (16). Essa subseção apresenta a definição dos *smell* mais recorrente na literatura, ver Tabela 3 do estudo realizado por Sobrinho, Lucia e Maia (39).

2.1.1 Duplicate Code (DC)

Segundo Cai e Kim (29) *"*Clones *are code fragments similar to one another in syntax and semantics"*. Fowler e Beck (15), denotam esse conceito como "Duplicate Code". Neste contexto, Bian et al. (40) propõem que os Clones são classificados em quatro tipos: (i) Fragmentos de código idênticos, exceto por variações no espaçamento e comentários; (ii) Trechos de código similares, onde identificadores e/ou variáveis podem ter sido renomeados(as); (iii) Fragmentos de código com semelhança sintática que apresentam uma ou mais declarações adicionadas/modificadas/deletadas; (iv) Por fim, temos aqueles trechos que executam a mesma operação usando sintaxes diferentes. Algumas vezes, Clones do tipo II (41) e/ou tipo III (42) são referenciados como *"near-miss clone"*.

Algumas ações são consideradas precursoras de Clones, uma delas é o copiar-colar e, por isso, alguns estudos referenciam este *smell* como *copy/cut-paste* (24). Além dos Clones entre entidades do mesmo projeto, segundo German et al. (43), Clones podem ocorrer entre projetos de software distintos. Nesse caso, o termo *"sibling code"* refere-se a um Clone que se desenvolve em um projeto diferente do qual o código se originou. Algumas vezes, esse tipo de Clone tem implicações legais (ex. direitos autorais — *Copyright*) e, por isso, há estudos de Clones voltados para a detecção de plágios (44).

2.1.2 Large Class (LC)

Segundo Fowler e Beck (15), o *smell* Large Class *"occurs when a class is trying to do too much, it often shows up as too many instance variables"*. Por outro lado, Brown et al. (16) definem que Blob Class *"is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data"*. Adicionalmente, Lanza e Marinescu (3) descreve o *smell* God Class como *"classes that tend to centralize the intelligence of the system. Performs too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes"*.

Certos estudos (45, 46), generalizam o conceito e consideram que Large Class também é conhecido como Blob, Winnebago, e/ou God Class. Por outro lado, Mantyla, Vanhanen e Lassenius (46), reportam que Large Class pode ser detectado e analisado sobre dois pontos de vista: 1) relacionado à medida de tamanho da classe, usando métricas tradicionais (ex. *Lines Of Code* — LOC)[2]; 2) o outro ponto está relacionado à falta de coesão, exemplo: classes que apresentam responsabilidades com pouca ou nenhuma relação entre si.

Alguns artigos (14, 48, 49) estudam Large Class relacionando o tamanho (ex. LOC) e a complexidade (ex. *McCabe Cyclomatic* (50)). Nesse caso, a noção de complexidade é definida como: *"a class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOC"* (14).

A literatura também identifica um tipo de Large Class denominado Brain Class, descrito como: *"classes tend to be complex and centralize the functionality of the system, but, differently from* God Classes*, they do not use much data from foreign classes and are slightly more cohesive"* (51). Alguns artigos (52, 53) investigam tanto o conceito de Large Class quanto o de Brain Class.

Em geral, o problema do *smell* Large Class é resolvido dividindo-se a classe original e um conjunto de classes mais coesas (15), neste caso, aplicando a operação de refatoração *Extract Class*.

[2] No artigo (14) este conceito é referenciado como "Large Class", enquanto que o artigo (47) usa o termo "Large Class Only".

2.1.3 Feature Envy (FE)

O *smell* FE ocorre quando o método de uma classe está mais interessado por funcionalidades de outra classe (ex. o método X da classe A invoca muitos métodos da classe B) (38). Por isso, alguns a denominam como entidade invejosa (32). Esse *smell* é caracterizado pelo grande número de dependências e, normalmente, isso influência negativamente na coesão e no acoplamento da classe onde o método caracterizado por FE está implementado. De fato, métodos FE reduzem a coesão da classe porque provavelmente implementa diferentes responsabilidades com relação àqueles implementados por outros métodos da classe e aumenta o acoplamento devido às muitas dependências com métodos da classe invejada.

Segundo Carneiro et al. (54), esse *smell* pode ser visto como um interesse erroneamente associado à classe, ou seja, o código do método não implementa o principal interesse da classe. Por isso, o código desse método deveria estar localizado em uma classe diferente. Portanto, esse *smell* é um sinal da violação do princípio de agrupamento de comportamentos (55). Recomenda-se que um método FE seja movido para a classe que ele inveja e, quando for o caso, move-se também o(s) atributo(s) associado(s) a esse método (55).

2.1.4 Long Method (LM)

Este *smell* também é conhecido como God Method (56, 57, 58) ou Brain Method (3). Segundo Chatzigeorgiou e Manakos (55), geralmente LM, são pedaços de código de tamanho grande, complexos e com baixa coesão que, consequentemente, necessitam de mais tempo e esforço para compreender, depurar, testar e manter. Nesse contexto, coesão refere-se ao tipo intra-método (59). Em complemento, Fowler e Beck (15) também associam a existência de muitos parâmetros e variáveis temporárias à ocorrência desse *smell*. Por outro lado, alguns estudos apresentam uma visão mais simplista desse *smell*, exemplo: Bavota e Russo (9) e Hecht et al. (56) consideram como LM todo método com muitas linhas de código.

Em geral, a técnica usada para resolver esse *smell* é simplificar o código através da quebra dele em métodos menores, criando métodos que podem ser reutilizados (55).

2.1.5 Complex Class (CC)

Este *smell* viola o princípio em que as classes precisam ser abertas para extensão e fechadas para modificação (60). Segundo Brown et al. (16), as classes com esse *smell* apresentam alta complexidade ciclomática. Isso quer dizer que possuem diversas estruturas condicionais e caminhos de execução distintos. Contudo, segundo Bavota et al. (61), uma classe que apresenta no mínimo um método com alta complexidade ciclomática pode ser considerada CC. Além da complexidade, Khomh et al. (33) também consideram o tamanho do(s) método(s) em termos de LOC. Estudos empíricos (48) revelam que esse

smell sempre é identificado pelos desenvolvedores que por sua vez sempre o classificam no mais alto nível de severidade, indicando a necessidade de refatorar.

2.1.6 Outros *Smells*

Outros *smells* recorrentes na literatura estão sempre associados a estes que descrevemos: (i) Data Class são classes que têm apenas atributos e métodos assessores (34). Então é natural que esse *smell* seja amplamente associado a outros (ex. Large Class); (ii) Long Parameter List caracterizado por métodos que têm uma longa lista de parâmetros (48), são frequentemente associados a Long Methods (15).

A literatura também apresenta alguns *smells* que consideram a forma e/ou frequência com que as entidades são alteradas durante a evolução do projeto de software. Nessa categoria temos, por exemplo: (i) Shotgun Surgery que ocorre quando uma alteração em determinada classe desencadeia várias alterações pontuais em diversas outras classes (38); (ii) Divergent Change que ocorre quando uma classe é alterada por diferentes formas por diversas razões (38). Assim, nos projetos de software em que as alterações realizadas pelos desenvolvedores são gerenciadas pelos sistemas de controle de versão (ex. Git (62), Subversion (63)), esses *smells* podem ser detectados analisando o histórico de registro dessas alterações (*commits* (64)).

2.2 Principais Estratégias de Identificação de *Smells*

Essa subseção apresenta resumidamente e de forma abrangente como os principais *bad smells* são detectados por ferramentas e/ou técnicas documentadas na literatura científica.

2.2.1 Duplicate Code (DC)

Como não se sabe de antemão quais fragmentos de código estão duplicados e onde estão, a ferramenta de detecção de clones inevitavelmente precisa comparar todos os fragmentos de código com todos os demais fragmentos de código. Portanto, existem diversas técnicas para detectar instâncias deste *smell* e as principais estão resumidamente descritas nos próximos parágrafos.

`Técnica de Análise Textual`. Nessa abordagem, dois fragmentos de clones são comparados uns aos outros na forma de texto/*string* e os códigos similares são relatados como clones (65). Geralmente, os códigos comparados têm pouca ou nenhuma normalização (ex. remover espaços e comentários). Assim, as abordagens baseadas em texto, geralmente não dependem da linguagem, uma vez que o código é considerado uma sequência de caracteres. *DuDe* (66) e *Simian*[3] são ferramentas baseadas nessa técnica. Enquanto a ferramenta

[3] <http://www.harukizaemon.com/simian>

Nicad (67), se baseia em uma abordagem híbrida, que combina uma análise sensível à linguagem com uma análise de similaridade que independe da linguagem.

`Técnica de Análise Léxica`. Essa técnica também é referenciada como *token-based* (65). Para detectar os clones, *tokens* são extraídos do código fonte por meio da análise léxica. Então, um conjunto de *tokens*, de uma granularidade específica, são organizados em uma sequência através de determinadas estruturas (*suffix tree* ou *suffix array*). Por fim, as entradas da estrutura são comparadas umas com as outras. *CCFinder* (68), *CP-Miner* (69) e PMD[4] são ferramentas que utilizam essa técnica. Segundo Rattan, Bhatia e Singh (65), essa técnica detecta grande parte dos clones e possui altos valores de *recall* com *precision* razoável.

`Técnica de Análise Sintática`. Em contraste com as técnicas anteriores, essa abordagem necessita que o código fonte seja transcrito em uma árvore sintática abstrata (*abstract syntax tree* — AST) e por isso, geralmente, ela é mais lenta do que as técnicas anteriores (65). A árvore é formada conforme a linguagem de programação, no caso do Java, pode-se utilizar o *Eclipse Java Development Tools* (JDT[5]) para montar a AST. Além dos clones do tipo I e II, a detecção de clones baseada em árvore também possibilita identificar clones do tipo III (65). A ferramenta *Deckard* (70) utiliza o princípio baseado em AST.

`Técnica de Análise Semântica`. Essa técnica utiliza análise estática para gerar um grafo de dependências do programa (*Program Dependence Graph* — PDG) (65), de forma que é possível considerar a sintaxe e a semântica do código. Assim, os vértices do grafo representam as declarações do código e os fluxos de dados e controles são descritos pelas arestas. As dependências entre os vértices do grafo podem revelar as semelhanças semânticas no código. Em comparação com as outras técnicas, essa possibilita encontrar mais tipos de clones. A ferramenta *Duplix* (71) utiliza este tipo de abordagem.

2.2.2 Large Class (LC)

Conforme apresentado, este *smell* pode ser descrito de diversas formas. Consequentemente, várias interpretações emergem na literatura e estas são concretizadas pela implementação e/ou proposição de técnicas de detecção. Nos próximos parágrafos, apresentamos algumas delas.

`Técnica Baseada em Dados Contidos no Sistema de Controle de Versão`. Palomba et al. (38) descrevem uma ferramenta denominada *HIST* (*Historical Information for Smell deTection*) que utiliza o histórico de alterações do projeto de software para determinar quais entidades são LC. A ideia é que se uma entidade é LC, então é provável que ela deverá ser alterada com mais frequência. Então, este *smell* é identificado pelo percentual de alterações de uma dada classe. Empiricamente os autores estabele-

[4] <http://pmd.sourceforge.net/pmd-4.3.0/cpd>
[5] <https://www.eclipse.org/jdt/>

ceram que o limiar de 8% fornece boa precisão de detecção. Entretanto, os autores não disponibilizaram uma versão da ferramenta.

A literatura apresenta diversas formas de detectar esse *smell* usando métricas calculadas diretamente no código fonte, sem a necessidade de usar os dados do sistema de controle de versão. Segundo Marinescu (1), o limiar destas métricas pode ser definido: (i) estatisticamente, através dos percentis da distribuição de frequência do conjunto de dados; (ii) em termos absolutos, um valor numérico é explicitamente informado.

`Técnica Baseada no Limiar Absoluto`. Segundo Brown et al. (16), entidades com esse *smell* são caracterizadas pela existência de mais de 60 atributos, métodos ou ambos. Por outro lado, Lorenz e Kidd (72) sugerem que as classes de interface com o usuário não devem apresentar mais de 40 métodos e as outras classes não devem apresentar mais de 20 métodos. Os autores do artigo (45) analisaram a percepção humana em relação a diversas regras estabelecidas por meio de valores absolutos (ex. número de variáveis ≥ 10, número de métodos ≥ 30, número de linhas de código ≥ 500, complexidade ≥ 100 e coesão ≥ 10). O resultado indica que a percepção humana do *smell* Large Class não se correlaciona com as métricas e seus respectivos limiares absolutos. As ferramentas PMD e Checkstyle usam este tipo de estratégia para detectar instâncias LC, em especial, usam respectivamente os limiares 1000 e 2000 para a métrica de número de atributos e métodos (73). Por outro lado, Liu et al. (30) consideram a seguinte regra: $LOC\ of\ Method > 100 \vee McCabe\ of\ Method > 20$.

`Técnica Baseada no Limiar Estatístico`. Marinescu (1) sugere que este *smell* pode ser detectado usando três métricas diferentes: (i) *Weighted Method Count* (WMC) que é a soma da complexidade ciclomática *McCabe* de todos os métodos da classe; (ii) *Tight Class Cohesion* (TCC) que é o número de métodos públicos da classe que diretamente acessam atributos dela; (iii) *Access to Foreign Data* (ATFD), representa o número de classes externas a partir das quais uma determinada classe acessa atributos, diretamente ou via métodos *getters*. Essas métricas são combinadas por meio de operadores lógicos e relacionais, como apresentado na Figura 1. Nesse caso, os limiares são: (i) WMC estabelecido pelo valor do 75° percentil; (ii) TCC se encontra no 25° percentil e (iii) ATFD é configurada com o valor absoluto igual a um.

O princípio do limiar baseado no percentil é empregado nas ferramentas DECOR (4) e HULK (74). Em ambos os casos, a métrica usada para identificar o *smell* LC é definida pela somatória entre a quantidade de atributos e métodos de cada classe, sendo que o limiar é estabelecido em função do 95° percentil.

2.2.3 Feature Envy (FE)

Este *smell* é um dos mais recorrentes na literatura (39), talvez, pela influência negativa na coesão e no acoplamento da classe onde o método FE está implementado ou porque, por

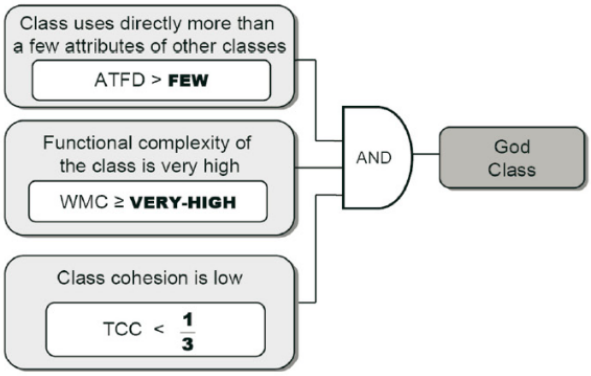


Figura 1 – Estratégia para detectar Large Class segundo Marinescu (1, 2).

meio da percepção humana, os desenvolvedores demonstram habilidade para identificar ocorrências desse *smell* (48).

`Técnica Baseada em Dados Contidos no Sistema de Controle de Versão`. A ferramenta HIST (38) também possibilita a detecção desse *smell*. Nesse caso, o raciocínio é que métodos afetados por esse *smell* são proporcionalmente mais alterados com as entidades que ele inveja do que com a entidade que ele se encontra. Portanto, usando a análise dos registros contidos no sistema de controle de versão é possível identificar instâncias dele. O estudo empírico demonstra que o limiar ideal para identificar esse *smell* é de 70%, ou seja, quando no mínimo 70% dos *commits* de um dado método são caracterizados por alterações que envolvem entidades de outras classes.

`Técnica Baseada no Limiar Estatístico`. Análogo ao apresentado para o *smell* anterior, este *smell* também pode ser detectado usando o limiar calculado com base no percentil dos dados. Nesse caso, Lanza e Marinescu (3) descrevem a regra apresentada na Figura 2, onde: (i) *Access to Foreign Data* (ATFD), representa o número de classes externas a partir das quais uma determinada classe acessa atributos, diretamente ou via métodos *getters*; (ii) *Locality of Attribute Accesses* (LAA), descrito como o número de atributos da classe dividido pelo número total de variáveis acessadas pelo método analisado; (iii) *Foreign Data Providers* (FDP) número de classes das quais o método acessa seus atributos.

`Técnica Baseada na Similaridade`. Nesta categoria temos as ferramentas e/ou técnicas que quantificam a distância ou similaridade que determinada entidade apresenta em relação ao restante do sistema, ou mesmo em relação a um conjunto específico de entidades. Para esse *smell*, a similaridade entre um método e uma classe deve ser alto quando o número de entidades em comuns é grande. Portanto, um método FE apresenta baixo coeficiente de similaridade com a classe que ele se encontra e alto coeficiente com outra entidade. Tsantalis e Chatzigeorgiou (75) detalham a ferramenta *JDeodorant* que utiliza este tipo de técnica para detectar o *smell* FE. Essa ferramenta baseia-se no cálculo da distância de *Jaccard*.

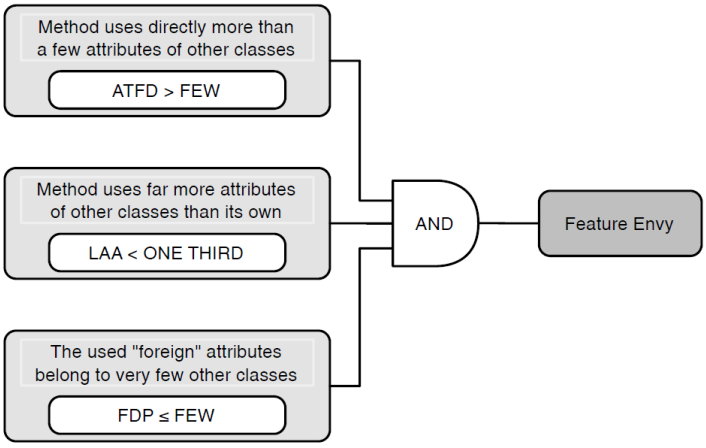


Figura 2 – Estratégia para detectar Feature Envy segundo Lanza e Marinescu (3).

2.2.4 Long Method (LM)

Estudos demonstram que métodos com este *smell* se relacionam direta e/ou indiretamente com vários outros tipos de *smells*, exemplo: (i) Long Method e Large Class são indicadores da presença do problema de projeto denominado Spaghetti Code (4). Naturalmente isso pode explicar o grande número de pesquisas que considera esse *smell* (39).

`Técnica Baseada no Limiar Absoluto`. Palomba (76), considera que os métodos com mais de 100 linhas de código sofrem do *smell* LM. As ferramentas PMD e Checkstyle detectam instâncias LM, usando respectivamente os limiares 100 e 150 para a métrica número de atributos e métodos (73). Fontana et al. (58), usam diversas técnicas de *machine learning* (ex. J48) para determinar os limiares absolutos de determinadas métricas. Os resultados empíricos apontam que os LM são regidos pela seguinte regra: $LOC\ of\ Method \geq 80 \wedge McCabe\ of\ Method \geq 10$. Por outro lado, Liu et al. (30) consideram a seguinte regra: $LOC\ of\ Method > 50 \vee McCabe\ of\ Method > 10$.

`Técnica Baseada no Limiar Estatístico`. A ferramenta DECOR (77), utiliza o valor da métrica LOC do método para identificar a ocorrência do *smell* LM, sendo que o limiar é estabelecido em função do 95° percentil da métrica em questão. De forma semelhante, Bavota et al. (61) define a regra desse *smell* como: *All methods having LOCs higher than the average of the system.* Lanza e Marinescu (3), também apresentam uma regra com múltiplas métricas e o limiar delas também é estabelecido por suas respectivas distribuições.

2.2.5 Complex Class (CC)

Em geral, o cálculo da complexidade de uma dada classe é obtido somando-se a métrica *McCabe* de complexidade ciclomática de cada método da classe (61). Segundo Tufano et al. (78), essa abordagem é definida pela descrição da métrica *Weighted Methods per Class*

(WMC).

`Técnica Baseada no Limiar Absoluto`. Segundo Bavota et al. (61) uma classe que apresenta no mínimo um método com alta complexidade ciclomática (>10) pode ser considerada CC. Além da métrica *McCabe*, Khomh et al. (33) também consideram o tamanho do(s) método(s) em termos de LOC.

`Técnica Baseada no Limiar Estatístico`. A ferramenta DECOR (77) calcula a complexidade da classe somando o valor da métrica *McCabe* de cada método da classe. Então, esse valor é usado para identificar a ocorrência do *smell* CC, sendo que o limiar é estabelecido em função do 95° percentil da métrica em questão (47).

2.2.6 Outras Estratégias/Smell(s)

A literatura apresenta diversas outras abordagens para detectar os mais diversos tipos de *smells*, exemplo: (i) `NosePrints` (79) é uma ferramenta destinada à visualização/-representação gráfica dos *smells*, ou seja, auxilia na inspeção visual deles possibilitando filtrar falsos positivos; (ii) `P-EA` (80) é uma abordagem baseada em algoritmo genético que foi concebida para gerar regras de detecção de *smells* baseadas em métricas estruturais do código fonte; (iii) `TACO` (81) é uma técnica baseada em métodos de recuperação de informação (*Information Retrieval* — IR) que executa uma análise textual nos identificadores (ex. nomes de variáveis) e comentários de cada componente do código.

Em nossa artigo (39), realizamos um levantamento com diversas ferramentas/técnicas usadas para manipular mais de 104 tipos de *smells*. A Tabela 9, deste artigo apresenta quais *smell* cada ferramenta está ou estava habilitada a lidar. No geral, catalogamos 77 ferramentas e estas foram colocadas em três grupos distintos: (i) *Public*, o artigo que usa/propõe a ferramenta compartilha um *link*, que no ano de 2017 ainda estava ativo, possibilitando baixar o código fonte e/ou uma versão compilada; (ii) *Deprecated*, igual ao *Public* mas o *link* não está ativo; (iii) *Commercial*, que é necessário comprar licença de uso; (iv) *Ad-Hoc*, os pesquisadores não compartilham qualquer versão da implementação. Os dados mostram que 75% dos *smells* podem ser detectados por ferramentas públicas e 47% por ferramentas *Ad-Hoc*.

2.3 Ferramentas do Estudo Empírico em Detalhes

Nos experimentos dos Capítulos 4 e 5, usaremos a ferramenta PMD[6] e o utilitário DECOR[7] (4). Os detalhes das estratégias usadas por essas ferramentas estão descritos nas próximas duas subseções. Elas serão detalhadas tendo como foco apenas os *smell* que serão estudados empiricamente nos Capítulos 4 e 5, ou seja, os *smells* Duplicate Code, Large e Complex Class. Essas ferramentas foram escolhidas porque são de código

[6] <https://pmd.github.io/>
[7] *DEtection & CORrection* — <https://bitbucket.org/ptidejteam/>

fonte aberto, o que permite verificar a implementação e ainda possibilita a confecção de customizações.

2.3.1 Detectando Clones (Ferramenta PMD)

A ferramenta PMD utiliza um conjunto de procedimentos para identificar fragmentos de códigos duplicados, em essência, o código fonte é *tokenized* e agrupado conforme regras internas que possibilitam encontrar os clones. *Tokenization*, ou análise léxica, é o processo de converter uma sequência de caracteres em uma sequência de *tokens*, que são *strings* com um significado atribuído conforme a linguagem de programação.

Assim, considere o exemplo, descrito nos próximos parágrafos, baseado no fragmento de código Java abaixo:

```
1 System.out.println("Hello");
2 System.out.println("World");
```

O primeiro passo na detecção de clones é ler cada arquivo de código fonte e realizar a *tokenization* deles, neste caso, a ferramenta PMD utiliza o JavaCC[8] para gerar os *tokens*. Como exemplo, para o fragmento *"System.out.println"*, a ferramenta JavaCC produz cinco *tokens*: *"System, ., out, ., println"*. Com o intuito de reduzir o espaço de busca e os fragmentos de clones desnecessários, na *tokenization*, o PMD descarta espaços em branco e alguns outros *tokens*, como: declarações de importação, declarações de pacotes e símbolos de ponto e vírgula. Para o fragmento de código anterior, a *tokenization* produz a seguinte saída:

System	.	out	.	println	(	"	Hello	"	)	System	.	out	.	println	(	"	World	"	)

Na sequência, o PMD cria uma tabela de ocorrências que é basicamente uma lista dos *tokens* e suas respectivas localizações. Abaixo, temos o exemplo de como fica essa lista para a *tokenization* produzida acima.

System	1, 11
.	2, 4, 12, 14
out	3, 13
println	5, 15
(	6, 16
"	7, 9, 17, 19
)	10, 20

Essa tabela é usada para expandir os *tokens* em conjuntos de *tokens*, para isso ele monta o encadeamento seguindo a localização sequenciada dos *tokens*, exemplo: o *token "System"* ocorre nas posições 1 e 11, na sequência (2 e 12), sempre temos o *token "."*,

[8] <https://javacc.org/>

portanto, temos o conjunto de *token "System."*. Para a lista de *tokens* acima, na primeira expansão, temos os seguintes conjuntos de *tokens*:

System.	1, 11
.println	4, 14
("	6, 16
")	10, 20

Observe que o *token "out"* não aparece nessa lista. Isso porque ele não é um *token* candidato a compor um registro exclusivo na tabela de conjuntos de *tokens*. Quando esse *token* é expandido (posições 4 e 14), surge o *token "."*, contudo, este já está sendo usado por outro registro (*"System."*) na tabela de conjuntos de *tokens*. Expandindo novamente, temos o seguinte conjunto de *tokens*:

System.out	1, 11
.println("	4, 14

Após outras expansões, encontra-se que o fragmento *"System.out.println("")"* é um código clonado, pois este se repete em duas posições (1 e 11). A ferramenta PMD utiliza o conceito de *tokens* e com o intuito de recuperar apenas clones relevantes, essa ferramenta solicita que o usuário informe um *threshold* mínimo para o número de *tokens* dos clones.

Essa ferramenta também permite realizar um filtro baseado no tipo de clone. Nesse caso, é possível considerar clones exatos (tipo I — Fragmentos de código idênticos, exceto por variações no espaçamento, layout e comentários (82)) e/ou clones estruturais (tipo II — Os fragmentos de código são estruturalmente e sintaticamente idênticos, exceto por variações nos identificadores, literais, tipos, layout e comentários (82)). Além disso, no processo de análise, também é possível desconsiderar os comentários de código realizado pelos desenvolvedores. No fim, a saída produzida pela ferramenta PMD conta com a quantidade de linhas clonadas, as próprias linhas tidas como clones, o número de *tokens* dessas linhas e o nome do(s) arquivo(s) onde o fragmento clonado se encontra.

2.3.2 Detectando *Large* e *Complex Class* (Ferramenta DECOR)

Para detectar os *smells* Large Class e Complex Class usamos a ferramenta DECOR. Internamente, ela transcreve o código fonte em uma árvore de nós organizados hierarquicamente conforme a estrutura léxica e/ou sintática da linguagem de programação em questão. A técnica usada para criar essa árvore de nós é referenciada como *"abstract syntax tree"* (AST), que segundo Vinh et al. (83) cada nó da árvore descreve uma construção ocorrida no código fonte e, por isso, Cooper e Torczon (84) consideram essa representação estruturada como sendo a mais próxima do código fonte original.

A ferramenta DECOR não implementa a construção da AST, em linhas gerais, ela provê mecanismos para extrair e/ou relacionar informações contidas na AST e implementa

regras, baseadas nessas informações, que possibilita identificar instâncias de *smells*. No DECOR, a construção da AST dos códigos desenvolvidos em Java ocorre utilizando-se o *Eclipse Java Development Tools* (JDT[9]).

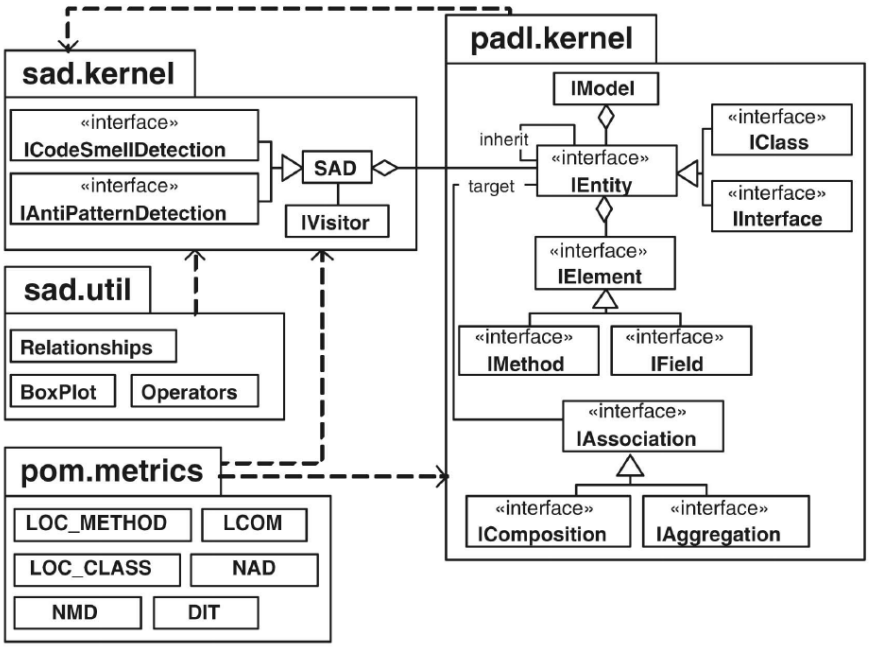


Figura 3 – Arquitetura de detecção de *smells* do DECOR (4).

Segundo Moha et al. (4), PADL (*Pattern and Abstract-level Description Language*) da Figura 3, é um metamodelo que independe da linguagem de programação usado pelo DECOR para representar sistemas orientados a objetos, incluindo relacionamentos de classes binárias e acessores. O PADL cria as ASTs usando o JDT e provê um conjunto de constituintes (classes, interfaces, métodos, campos, relacionamentos) usados para criar modelos dos sistemas, fornecendo também mecanismos para manipular estes modelos e gerar outros modelos, usando o padrão de projeto *Visitor*.

Segundo Munro (85), a partir da AST vários atributos de um sistema podem ser medidos e estes podem resultar em métricas do sistema. Isso pode ser verificado claramente na Figura 3 que representa a arquitetura geral de detecção de *smells* do DECOR. Nessa figura, o pacote *POM.metrics* calcula métricas do sistema conforme o modelo gerado no pacote *PADL.kernel*. Segundo Moha et al. (4), *POM.metrics* é capaz de calcular 44 métricas distintas, como: linhas de código de uma classe (LOC_CLASS), número de métodos declarados na classe (NMD), falta de coesão nos métodos (LCOM). Observe ainda que o pacote *SAD.kernel* é quem efetivamente implementa as regras de detecção dos *smells* e esses *smells* estão organizados convenientemente de duas formas: *AntiPattern* e *CodeSmell*.

Ainda segundo Moha et al. (4), DECOR consegue identificar 8 *AntiPatterns* e 21 *CodeSmells*, totalizando 29 *smells*. Em termos de implementação, os *CodeSmells* (ex. Long

[9] <https://www.eclipse.org/jdt/>

Method) são detectados usando relações diretas com alguma(s) métrica(s); por outro lado, os *AntiPatterns* são entidades que apresentam relações com múltiplos *CodeSmells*. Essa observação pode ser vista na regra de detecção do *AntiPattern* (*smell*) Spaghetti Code, representada na Figura 4, observa-se que esse *smell* é fruto de uma relação com seis *CodeSmells* (28). O *smell* Spaghetti Code ocorre[10] em classes que: (i) apresentam métodos com muitas linhas (linha 3 da Figura 4); (ii) apresentam métodos sem parâmetros (linha 4 da Figura 4); (iii) não usam herança (linha 5 da Figura 4); (iv) não usam polimorfismo (linha 6 da Figura 4); (v) apresentam entidades com nomes procedurais como *Make, Create, Compute* (linha 7 da Figura 4); (vi) declaram e/ou usam variáveis globais (linha 8 da Figura 4). Nessa subseção, vamos concentrar apenas nos *smells* Large e Complex Class, portanto, para detalhes de outras regras de detecção consulte a Figura 4 do artigo (4).

```
1  RULE_CARD:SpaghettiCode {
2    RULE:SpaghettiCode
           { INTER LongMethod NoParamete NoInheritance
           NoPolymorphism ProceduralName UseGlobalVariable };
3    RULE:LongMethod       { METRIC LOC_METHOD VERY_HIGH 10.0 };
4    RULE:NoParameter      { METRIC NMNOPARAM VERY_HIGH 5.0 };
5    RULE:NoInheritance    { METRIC DIT 1 0.0 };
6    RULE:NoPolymorphism   { STRUCT NO_POLYMORPHISM };
7    RULE:ProceduralName   { LEXIC CLASS_NAME
                                 (Make, Create, Exec...) };
8    RULE:UseGlobalVariable { STRUCT USE_GLOBAL_VARIABLE };
9  };
```

Figura 4 – Regra usada pelo DECOR para detectar o *smell* Saghetti Code (4).

Large Class. No DECOR, uma classe é considerada grande quando a mesma apresenta muitas declarações de métodos e atributos. Por essa definição observa-se dois componentes: um subjetivo relativo ao limiar (muitas) e o outro numérico relativo à(s) métrica(s) usada na detecção de *smells*. Estes componentes são definidos como:

- Numérico: Considerando o *smell* Large Class, o pacote *POM.metrics* define duas métricas numéricas: (i) *Number of Methods Declared* (NMD) e (ii) *Number of Attributes Declared* (NAD), que para cada classe do sistema ($_{Ci}$), são combinadas por somatória ($NMD_{Ci} + NAD_{Ci}$) na detecção desse *bad smell*.

- Subjetivo: Para resolver essa questão do limiar, as métricas e/ou suas combinações são discretizadas em níveis, como: Baixo, Médio e Alto (ver Figura 5). A ideia é que as classes, onde o(s) valor(es) da(s) métrica(s) se encontra(m) em um nível específico, são classes candidatas a conter instâncias de determinado *smell*. A discretização é

[10] Neste caso, classes que têm uma interseção com as características enumeradas.

realizada através da técnica estatística *boxplot* (86), que permite destacar particularidades estatísticas de uma distribuição e também possibilita a identificação de valores anormalmente altos ou baixos (*outliers*) (5). Assim, conforme a distribuição da métrica $NMD_{Ci}+NAD_{Ci}$, medida para cada classe ($_{Ci}$) do sistema, DECOR define que todas as classes que apresentam valores anormalmente altos para essa métrica ($NMD_{Ci}+NAD_{Ci} \geq Máximo$) são classes que contêm o *smell* Large Class. Contudo, na prática, essa equação dicotômica é implementada considerando que algumas instâncias desse *smell* podem apresentar limiares pouco abaixo do limite superior (Máximo) obtidos pelo *boxplot*. Portanto, para esse *smell*, a implementação do DECOR introduz o termo de incerteza (*Fuzziness*) que representa uma faixa de valores entre o máximo e o mínimo do *boxplot* (ver Figura 5). Então a equação do DECOR para esse *smell* pode ser reescrita como: $NMD_{Ci}+NAD_{Ci} \geq Máximo-Fuzziness$, onde $Fuzziness = (Máximo - Mínimo) * 10\%$.

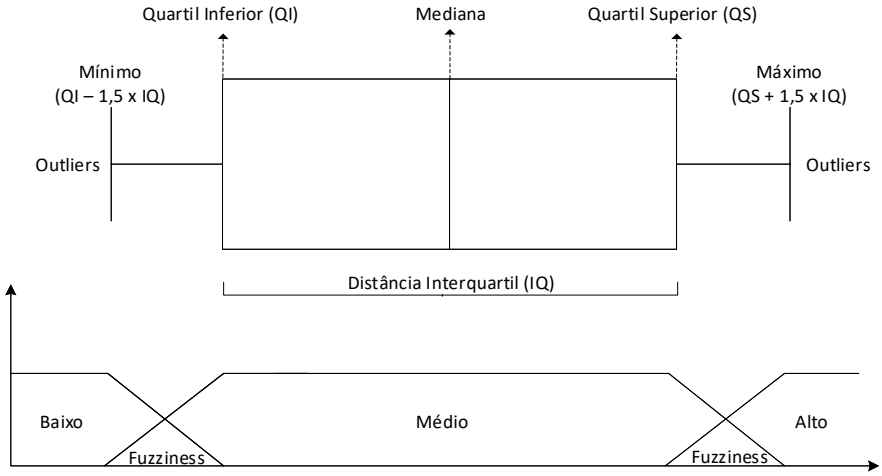


Figura 5 – Boxplot — Estratégia de detecção dos *smells* Large e Complex Class (Baseado na Figura 2 do artigo (5)).

Complex Class. No DECOR, uma classe é considerada complexa quando a mesma apresenta muitas estruturas de decisão, como *if statement*. Observe que em termos de componentes, essa definição é semelhante àquela apresentada no *smell* Large Class, ou seja, temos o componente subjetivo e o numérico:

- Numérico: O pacote de métricas do DECOR implementa o cálculo da complexidade de uma entidade usando o conceito *Cyclomatic Complexity* (McCabe) descrito em (87). Em essência, essa métrica quantifica o número de caminhos linearmente independente dentro de uma entidade (ex. classe, método). No contexto de detecção do *smell* Complex Class, essa métrica é calculada para cada classe do sistema ($McCabe_{Ci}$).

- Subjetivo: Este item, segue os mesmos princípios descritos no *smell* LARGE CLASS. Contudo, para esse *smell*, a métrica usada na equação de detecção é diferente, em especial, se usa a métrica $McCabe$ detalhada no item anterior. Portanto, para esse *smell*, a equação de detecção do DECOR pode ser escrita como: $McCabe_{Ci} \geq Máximo - Fuzziness$, onde $Fuzziness = (Máximo - Mínimo) * 10\%$ (ver Figura 5).

Capítulo 3

Co-estudos em *Bad Smells*: Mapeamento Sistemático da Literatura

Conforme apresentado no primeiro capítulo, nosso objetivo é realizar um estudo que investiga a possível inter-relação entre *smells*. Para tanto, é necessário observar na literatura como o fenômeno vem sendo estudado. No presente capítulo, investigamos quais *bad smells* e a frequência que eles ocorrem nos artigos científicos. Também apresentamos uma visão que considera o porquê alguns *smells* surgem nos artigos concomitantemente com outros *smells*. Neste caso, o motivo pode ser por mera conveniência experimental, coincidência por estarem presentes nos sistemas estudados ou devido à ocorrência/suposição de alguma semântica que inter-relaciona os *smells* do estudo. Devido à larga extensão do tema *bad smell*, evidenciado pelo alto número de estudos realizados em décadas, decidimos, primeiramente, verificar se há revisões/*surveys* que consideram estes aspectos. Assim, nos próximos parágrafos, apresentamos uma visão geral dos *surveys* mais relevantes relacionados a *bad smell*.

Rattan, Bhatia e Singh (65), apresentam uma revisão sistemática do *smell* Duplicate Code. Os autores visavam identificar técnicas e ferramentas de detecção. Eles também apresentam uma classificação dos artigos e uma comparação das ferramentas usadas nos artigos da revisão sistemática. O banco de dados de artigos usados na revisão foi construído pesquisando o termo *"clone"* nos seguintes repositórios: IEEExplore, ACM DL, ScienceDirect, Springer, and Wiley. Entretanto, não consideram a questão da inter-relação entre *smells*.

Bandi, Williams e Allen (88), identificaram quais técnicas e métricas têm sido avaliadas empiricamente no processo gradual que afeta negativamente a qualidade do software (*code decay*). Eles reportaram que as métricas de acoplamento são amplamente usadas na detecção de *code decay*. A base de dados deste artigo foi criada com base em uma pesquisa textual composta de várias palavras e operadores lógicos nos sites: IEEExplore, ACM, Scopus e Google Scholar. Este artigo apresenta um escopo abrangente, pois inclui o conceito de *bad smells*, bem como os conceitos de violação da arquitetura e regras de

projeto. Contudo também não consideram a inter-relação entre *smells*.

Pate, Tairas e Kraft (89), conduziram uma revisão sistemática focada nos métodos usados, nos padrões encontrados e na evolução dos códigos duplicados. Eles estudaram 30 artigos, obtidos de forma análoga ao trabalho Bandi, Williams e Allen (88), e observaram que os pesquisadores têm tirado conclusões sobre o comportamento ou intenção do desenvolvedor com base apenas em dados analíticos resultantes da análise do código fonte. Os autores também indicam a necessidade de estudos empíricos envolvendo não apenas o código fonte, mas também os desenvolvedores. O estudo também reporta: *"there are contradictions among the reported findings, particularly regarding the lifetimes of clone lineages"*. Essa revisão por sua vez também não considera a inter-relação entre *smells*.

Zhang, Hall e Baddoo (90), fizeram uma revisão sistemática da literatura publicada entre os anos de 2000 e 2009. Os autores buscaram responder a quatro questões: (i) Quais *bad smells* têm recebido maior atenção? (ii) Quais são os objetivos dos artigos que estudam *bad smells*? (iii) Quais métodos/técnicas são usadas no estudo de *bad smells*? (iv) Há evidências de que os *bad smells* indicam problemas no código fonte? Para responder eles selecionaram artigos em alguns periódicos/revistas (JSS, EMSE, IST, JSME, TOSEM e SP&E) que estudaram um ou mais *bad smells* apresentados por Fowler e Beck (15). A etapa inicial de seleção dos artigos é conduzida por meio de um filtro baseado em pesquisa textual formada por termos e operadores lógicos. Ao final da seleção, 39 artigos foram considerados relevantes e foram investigados. Quanto a primeira pergunta (i), descobriram que os principais *smells* que mais atraíram atenção foram: Duplicade Code (54%), Feature Envy (31%), Refused Bequest (28%), Data Class (26%), Long Method (21%) e Large Class (21%). Reportam ainda que Duplicate Code tende a ser estudado isoladamente e é mais estudado por ser de fácil entendimento. Para a segunda questão (ii) eles encontraram que 49% dos artigos têm como objetivo desenvolver ferramentas e métodos para detectar *bad smells*, 33% são destinados a melhorar o entendimento dos *bad smells* e 15% estão focados no desenvolvimento de ferramentas/-métodos de refatoração dos *bad smells*. Na terceira indagação (iii), 52% dos estudos são empíricos, 33% são focados na execução de experimentos, 12% são questionários/*surveys*. Por fim, para a quarta questão (iv), apenas 5 artigos dos 39 selecionados investigaram o impacto dos *bad smells*. Os autores sugerem que a falta de estudos no impacto dos *bad smells* pode ser explicada porque há um senso comum do impacto negativo dos *smells* e os pesquisadores não acreditam que haja valor em buscar evidências sobre isso, consequentemente, os pesquisadores se concentraram em investigar como identificar os *bad smells*. No entanto, curiosamente, quatro dos cinco artigos que investigaram o impacto dos *bad smells*, mostram que nem todos os *bad smells* têm impacto negativo no código, exemplo: Duplicate Code pode aumentar confiança do software; Data Class, Refused Bequest e Feature Envy não estão significativamente associados com falhas do software. Análogo aos demais estudos, este também não considera inter-relação de *smells*.

Podemos observar algumas limitações nas revisões sistemáticas anteriores que ajudam a justificar a execução de uma nova revisão sistemática. Alguns estudos estão limitados apenas em um determinado *bad smell*, mais especificamente em DUPLICATE CODE (65, 82). Por outro lado, temos *surveys* muito abrangentes (88) que investigam os artigos relacionados a muitos dos fatores que afetam gradativamente e negativamente a qualidade do software (ex. *bad smells, violations of architecture, design rules*). Nesse caso, o alto número de fatores que afetam a qualidade do software torna impraticável um estudo mais abrangente. Há também revisões sistemáticas que investigam a percepção humana dos profissionais (desenvolvedores e/ou pesquisadores) quanto ao tema *bad smell* (91). Reforçamos que não encontramos revisões anteriores que consideram a inter-relação entre *smells*, portanto, para complementar os estudos anteriores, uma revisão detalhada dos *smells* seria benéfica. Dos *surveys* apresentados, Zhang, Hall e Baddoo (90) estudam um conjunto de artigos com a maior diversidade de *smells*. Entretanto, este artigo está limitado apenas aos estudos publicados entre os anos de 2000 e 2009 deixando uma lacuna de artigos publicados entre 2010 e 2017 que não foram explorados, além de desconsiderar os artigos que não estudam os *smells* descritos por Fowler e Beck (15), ou seja, este trabalho desconsidera os artigos que estudam *bad smells* de outros autores (ex. Brown et al. (16)). Além disso, nas revisões anteriores, não conseguimos encontrar uma revisão que investigasse a inter-relação de *smells*.

Assim, este capítulo apresenta e descreve os detalhes de nossa revisão sistemática da literatura no tema *bad smell* entre os anos de 1990 e 2017 que leva em consideração diferentes tipos de *smells*. Adicionalmente, no processo de seleção dos artigos, não iremos considerar apenas uma definição específica de *smells* (ex. Fowler e Beck (15)). Nós consideramos um conceito amplo da definição de *bad smell*, ou seja, um sintoma de uma má decisão observada na estrutura de baixo nível de um programa. Além disso, consideramos e analisamos os artigos que realizaram estudos da inter-relação de *smells*.

3.1 Perguntas de Pesquisa

Como desejamos analisar os artigos que estudaram a inter-relação de *smells*, devemos diferenciar os artigos que consideram os *smells* para intencionalmente investigar alguma possível relação entre eles (**co-estudo**), daqueles artigos em que os *smells* surgem no mesmo estudo apenas por conveniência e/ou coincidência do planejamento experimental (**co-ocorrência**). Assim, usando esses conceitos, nessa subseção definimos as perguntas de pesquisa (*Research Questions* — RQs) usadas em nossa revisão sistemática:

RQ1 Há *bad smells* significativamente mais estudados do que outros? Se sim, há alguma razão específica?

Objetivo: Identificar possíveis lacunas nas pesquisas de *bad smells*, fornecendo informações para guiar futuras pesquisas.

RQ2 Quais são as co-ocorrências de *smells* mais comuns?

Objetivo: Identificar o quão provável os *bad smells* poderiam estar inter-relacionados, possivelmente direcionando novas pesquisas onde o estudo conjunto desses *smells* ainda não foram estabelecidos.

RQ3 Considerando os *smells* mais recorrentes, quais são as principais descobertas no co-estudo de *smells*?

Objetivo: Compreender as limitações e os desafios das pesquisas realizadas sobre o tema de *bad smell*, considerando um possível relacionamento derivado da co-ocorrência de diferentes *smells* no código fonte. Em outras palavras, investigamos os artigos que relatam resultados em co-estudos de *smells*.

3.2 Método

Segundo Kitchenham, Budgen e Brereton (92), uma revisão sistemática deve seguir um método formal e reproduzível, possibilitando a identificação, avaliação e interpretação de estudos científicos (artigos) que estão relacionados ao tema desejado (*bad smells*).

A execução da nossa revisão sistemática consiste em três etapas principais: a) Definição do protocolo para selecionar e analisar artigos relevantes; b) Execução do protocolo e c) Descrever os resultados. Nosso protocolo consiste dos seguintes elementos: i) *Extração de Dados a Partir dos Veículos de Publicação*: essa parte mostra como selecionar artigos relevantes diretamente dos veículos de publicação predeterminados; ii) *Extração de Dados das Referências*: para minimizar a possibilidade de algum artigo relevante não ser incluído na revisão sistemática, as referências dos artigos selecionados na etapa anterior são examinadas de forma não recursiva; iii) *Análise do Banco de Dados Final*: Representa o conjunto de todos os artigos relevantes. As próximas subseções descrevem cada um dos elementos que compõem este protocolo.

3.2.1 Extração de Dados a Partir dos Veículos de Publicação

Nossa estratégia para selecionar os artigos que farão parte desta revisão sistemática será baseada na **inspeção manual** do metadados de todos os artigos publicados nos principais *veículos* de engenharia de software.

Os *veículos* selecionados para análise dos artigos são aqueles usados em revisões anteriores. Ao comparar a lista de *veículos* das revisões: (65, 88, 89, 90, 93), extraímos aquelas que aparecem em no mínimo três trabalhos e têm *topic area* relacionado ao tema *bad smell*.

O resultado foi uma lista preliminar com 14 *veículos* (ex. ICSE, TSE, WCRE, ASE). De forma complementar, e por considerá-las importantes, também adicionamos alguns *veículos* que aparecem em até dois destes artigos (ESEC/FSE, FASE, ICSME, OOPSLA, ECOOP e SANER). Ao final, nossa lista de *veículos* é formada por 20 *veículos*, sendo:

ASE - International Conference on Automated Software Engineering; **CSMR** - Conference on Software Maintenance and Reengineering; **ECOOP** - European Conference on Object-Oriented Programming; **EMSE** - Empirical Software Engineering; **ESEC/FSE**[1] - European Software Engineering Conference and International Symposium on Foundations of Software Engineering; **ESEM** - Symposium on Empirical Software Engineering and Measurement; **FASE** - Fundamental Approaches to Software Engineering; **ICSE** - International Conference on Software Engineering; **ICSME** - International Conference on Software Maintenance and Evolution; **IWPC/ICPC** - International Workshop/Conference on Program Comprehension; **JSME** - Journal of Software Maintenance and Evolution: Research and Practice; **JSS** - Journal of Systems and Software; **MSR** - Working Conference on Mining Software Repositories; **OOPSLA** - Conference on Object-Oriented Programming, Systems, Languages and Applications; **SCAM** - Working Conference on Source Code Analysis and Manipulation; **SP&E** - Software: Practice and Experience; **TOSEM** - ACM Transactions on Software Engineering and Methodology; **TSE** - IEEE Transactions on Software Engineering; **WCRE** - Working Conference on Reverse Engineering. Também consideramos **SANER** - International Conference on Software Analysis, Evolution and Reengineering, que é o resultado da fusão entre WCRE e CSMR. Então os artigos publicados no WCRE-CSMR (2014) foram agrupados na conferência SANER.

Os *veículos* acima estão representados na Figura 6a por meio de retângulos. *Conferences* e *Workshops* são os retângulos parcialmente preenchidos e os *Journals* são os retângulos sem preenchimento. A figura também apresenta a quantidade de artigos encontrada em cada *veículo*. Observa-se que do total de 29.077 artigos, o *Journal* JSS é o mais representativo com 3.731 (12,8%) artigos.

Alguns *veículos* têm seus artigos publicados em vários sites de indexação. Isso pode ser visto na conferência ASE, a 28th edição está publicada no site do IEEExplore e a edição 29th se encontra no site da ACM. Para esses *veículos*, consideramos os artigos indexados em ambos os sites. Com o intuito de facilitar a visualização dessa informação, a Figura 6a apresenta uma legenda (*Publisher*) que permite verificar onde os artigos de cada *veículo* estão publicados.

A partir dos artigos publicados nos *veículos*, anteriormente selecionados, construímos um repositório local intermediário contendo os metadados de todos os artigos desses *veículos*. Esse repositório permite a identificação de tantos artigos quanto possível, desde que relacionado com qualquer tema em engenharia de software. Assim, nos próximos parágrafos, descrevemos como esse repositório foi usado para obter apenas os artigos de

[1] Ambos ESEC/FSE e apenas FSE

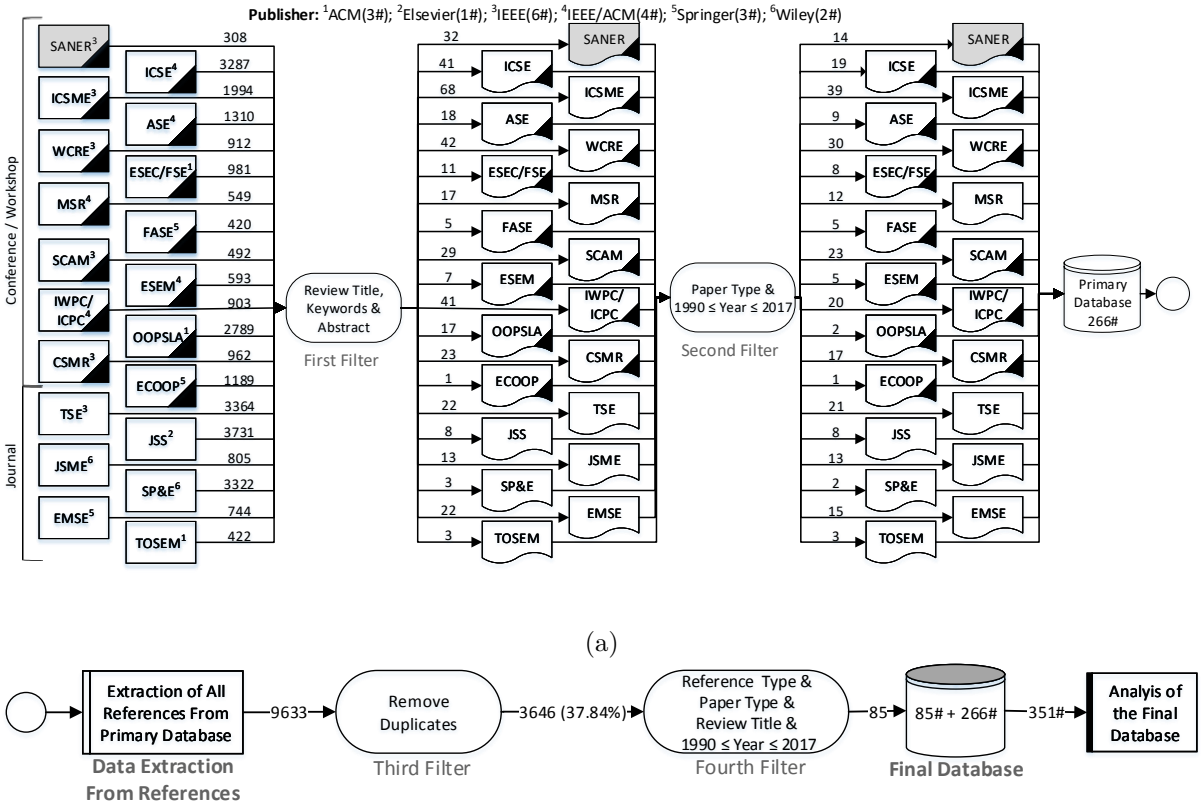


Figura 6 – Representação do protocolo. (a) *Primary Database*. (b) *Final Database*.

interesse e montar o *Primary Database* (ver Figura 6a).

As etapas envolvidas na confecção deste repositório local intermediário são:

1. Encontrar os sites de indexação (ACM, IEEExplore, etc.) dos *veículos*;
2. Obter os arquivos de metadados (BibTeX[2]), de cada artigo, em cada *veículo*;
   a) Complementar o metadado com o *Abstract* do artigo;
3. Importar os metadados para uma ferramenta de gerenciamento (Mendeley[3]).

Usando esse repositório local intermediário, aplicamos dois filtros iniciais (ver Figura 6a), que visam selecionar apenas artigos que estudam o tema *bad smell*. No restante desta subseção, detalhamos estes filtros.

First Filter. Para cada artigo do repositório, lemos o conteúdo dos campos *Title, Abstract* e *Keywords*, ou seja, realizamos a inspeção manual desses campos. Caso o artigo trate explicitamente do conceito de *bad smell*, então o mesmo é encaminhado para a próxima etapa. Se esses campos não forem suficientes para saber se este artigo é sobre o tema em questão, então, algumas seções de artigo (ex., Introdução, Conclusão) são

[2] <http://www.bibtex.org>
[3] <https://www.mendeley.com>

examinadas. No entanto, a aplicação desses critérios ao conjunto preliminar de artigos teve um desafio: alguns artigos apresentam uma classificação incerta, ou seja, a inclusão ou exclusão do *Primary Database* é discutível. Refatoração é um assunto recorrente nos artigos de classificação incerta, isso porque a refatoração pode ser usada para diferentes fins (ex. melhorar o projeto, a legibilidade, a estrutura, o desempenho, a manutenção e/ou compreensão (15)). Em geral, os artigos de classificação incerta não discutem ou investigam códigos com *bad smells*, eles apenas apresentam técnicas/métodos que podem ser implicitamente usadas no contexto dos *smells*. Então, os artigos foram incluídos no *Primary Database* apenas se eles se enquadram em pelo menos um dos seguintes critérios: (i) artigo que reporta resultados empíricos/qualitativos no tema *bad smell* (ex. detecção, análise, refatoração de *smells*); (ii) artigo que reporta ferramenta/método usado para tratar *bad smells*; (iii) artigo que reporta o uso de *bad smells* em domínios estreitamente relacionados (ex. projeto/documentação). Por outro lado, se o artigo apenas menciona os *smells*, mas não discute ou investiga o problema dos *smells*, então o mesmo é excluído do *Primary Database*.

Second Filter. Neste ponto, o filtro de seleção é dividido em duas condições que devem ser atendidas simultaneamente para que o artigo passe para a próxima etapa: (i) ***Paper Type***: o artigo analisado não pode ser um *short paper*, ou equivalente, pois estes geralmente são estudos/resultados preliminares. Além disso, *short papers* podem ser estendidos e publicados como *full paper*. Assim, selecionamos apenas os *full papers*, especificamente, aqueles publicados na trilha principal da conferência (ex., *Research Track* do SANER). (ii) ***Threshold Year***: O ano de publicação do artigo deve ser maior ou igual a 1990 e menor do que Abril de 2017. O limite inferior do período é baseado no ano em que os primeiros estudos de refatoração foram publicados (94, 95). Já o limite superior é porque a coleta e análise dos dados ocorreu em Abril de 2017.

Após o segundo filtro, o conjunto de artigos selecionados, é referenciado como *Primary Database* (ver Figura 6a). O *Primary Database* é composto por um conjunto de 266 artigos que estudam o tema *bad smell*.

Afim de maximizar a seleção de artigos relevantes, as referências dos artigos que compõem o *Primary Database* serão examinadas na próxima etapa do protocolo. Esse procedimento é detalhado na próxima subseção.

3.2.2 Extração de Dados das Referências

A extração das referências do *Primary Database*, foi realizada de forma não recursiva e automatizada. As referências recuperadas foram submetidas a outros dois filtros, como mostrado na Figura 6b:

Third Filter. Remoção das referências com títulos duplicados;

Fourth Filter. Inclusão dos artigos com as seguintes características: (i) ***Reference Type***: A nova sugestão deve ser um artigo. Livros, relatórios técnicos e similares são

descartados. (ii) ***Paper Type***: Segue os mesmos critérios adotados no *Second Filter* (ver *Paper Type* na Subseção 3.2.1); (iii) ***Review Title***: análogo ao *First Filter* da Subseção 3.2.1, o artigo deve tratar do tema *bad smell*. Entretanto, nesse filtro, a verificação da relevância com o tema desejado ocorre apenas pela análise do título. Se o título não é suficientemente claro sobre a relação com tema *bad smell*, o artigo é descartado; (iv) ***Threshold Year***: Segue os mesmos critérios adotados na subseção anterior (ver *Threshold Year* na subseção 3.2.1).

Podemos observar que 9.633 referências foram extraídas e 62,15% (5.987#) são referências duplicadas. Nesta etapa, identificamos mais 85 artigos relevantes ao tema *bad smell*. Esses 85 artigos estão distribuídos em 60 novos *veículos* (ex., *Information and Software Technology — IST*, *Dagstuhl Seminar Proceedings — DSP*). Considerando os novos *veículos*, o que mais contribuiu com novos artigos, contribuiu com apenas 5 artigos. A maior parte dos novos *veículos* (76,6%) contribuiu com apenas um artigo.

Assim, construímos o *Final Database* com 266 artigos do *Primary Database* e 85 artigos originados na revisão das referências.

3.2.3 Execução do Protocolo e Qualidade dos Dados

O protocolo foi executado pelo autor deste trabalho, o qual têm cinco anos de experiência no desenvolvimento de software, e também têm experiência em refatoração.

O critério para, manualmente, filtrar os artigos relacionados ao tema *bad smell* foi alvo de uma avaliação do grau de conformidade. Então, selecionamos randomicamente uma amostra de 130 artigos do conjunto inicial de 29.077 artigos (observe que consideramos todos os artigos antes de aplicar o primeiro filtro, ver Figura 6). Neste caso, não consideramos os artigos extraídos das referências do *Primary Database* porque no quarto filtro, se o título não é suficientemente claro sobre a relação com o tema *bad smell*, o mesmo é descartado. Essa amostra tem, em igual proporção, artigos que foram incluídos e excluídos do *Primary Database*, também incluímos artigos que apresentam uma classificação incerta (*borderline*) que foram incluídos e excluídos do *Primary Database*. Na sequência, o discente e seu orientador independentemente interpretaram e aplicaram os critérios de inclusão e exclusão nos artigos da amostra de 130 trabalhos. No final, o coeficiente Kappa foi calculado (96). Observamos que o valor foi de 0,92 que, segundo Landis e Koch (97), é caracterizado como substancial. Assim, podemos confiar na análise do discente para o resto dos artigos.

Além disso, os envolvidos discutiram as divergências de classificação sem considerar os critérios de tipo do artigo (*short/full paper*) e ano de publicação. Em geral, tivemos apenas cinco conflitos (3,8% de 130 artigos). A partir da análise qualitativa, observamos que não é possível controlar 100% dos fatores. No entanto, as discrepâncias ocorrem a uma taxa muito pequena e não afetarão os resultados.

3.2.4 Análise do Banco de Dados Final

Analisando os dados do *Final Database* observamos que 227 (64,7%) artigos estudam exclusivamente o *bad smell* Duplicate Code. Os outros 124 (35,3%) artigos, estudam um conjunto de *bad smells* (ex., Duplicate Code com Large Class) ou estudam *bad smell(s)* diferente de Duplicate Code (ex., Feature Envy). Decidimos distinguir estes dois conjuntos de artigos e vamos referenciá-los como: *Duplicate Code Group* (DCG) e *Other Bad Smells Group* (OBSG), respectivamente.

Relembrando, também definimos uma terminologia específica para tratar o fato de que diferentes *smells* podem surgir no mesmo artigo. Consideramos que quando diferentes *smells* são investigados no mesmo artigo, dizemos que eles co-ocorrem neste artigo. **Co-ocorrência** não significa necessariamente que eles sejam estudados juntos investigando alguma relação entre eles. Nesse caso, sempre que os *smells* ocorrem no mesmo artigo para intencionalmente investigar alguma possível relação entre eles, dizemos que esses *smells* são **co-estudados** no artigo.

Para cada artigo, extraímos (i) a lista de todos os tipos de *smells* ocorridos no artigo; (ii) o ano de publicação que foi extraído dos arquivos de citação; (iii) as ferramentas e (iv) projetos usados nos experimentos; (v) o objetivo do artigo; a lista de (vi) autores, (vii) instituições e (viii) países. Seguindo a definição dos *smells* apresentado em cada artigo, agrupamos a terminologia dos *smells* em termos exclusivos. Portanto, a partir da inspeção manual, definimos a lista de todos os *smells* considerados neste trabalho. No próximo parágrafo, apresentamos um exemplo deste agrupamento.

Segundo Fowler e Beck (15), *"*Large Class *occurs when a class is trying to do too much, it often shows up as too many instance variables"*. Brown et al. (16), definem *"*Blob Class *is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data"*. Segundo Lanza e Marinescu (3), *"*God Class *refers to classes that tend to centralize the intelligence of the system. Performs too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes"*. Certos estudos (45, 46) generalizam os conceitos e consideram que Large Class é também conhecido como Blob, Winnebago, e/ou God Class. Por outro lado, Mantyla, Vanhanen e Lassenius (46) reportam que Large Class pode ser detectado e analisado a partir de dois pontos de vista: 1) relacionado à medição do tamanho da classe, geralmente usando métricas tradicionais (ex., *Lines Of Code* — LOC)[4]; 2) o outro ponto está relacionado à falta de coesão, por exemplo, classes que têm responsabilidades com pouca ou nenhuma relação entre si. Alguns artigos (14, 48, 49) estudam "Large Class" considerando o tamanho (ex. LOC) e a complexidade (ex. *McCabe Cyclomatic* — McCabe (50)). Neste caso, a noção de complexidade — referenciada como "Complex Class" — é definida como: *"a class that has (at least) one large and complex method, in*

[4] Em (14) este conceito é referenciado como "Large Class" enquanto que neste trabalho usamos o termo "Large Class Only", extraído de Khomh, Di Penta e Gueheneuc (47).

terms of cyclomatic complexity and LOC" (14). A literatura também identifica um tipo de Large Class chamado Brain Class, descrito como: *"classes tend to be complex and centralize the functionality of the system, but, differently from* God Classes*, they do not use much data from foreign classes and are slightly more cohesive"* (51). Alguns artigos (52, 53) investigam simultaneamente o conceito de Large e Brain Class. Nesta revisão sistemática, o termo Large Class é usado para agrupar os artigos que estudam os conceitos: Blob, Winnebago, e/ou God Class, mesmo porque não encontramos artigos que, simultaneamente, estudem uma combinação desses *smells*. Por outro lado, encontramos artigos que estudam as seguintes combinações: Complex Class *vs.* Large Class; Brain Class *vs.* Large Class; Large Class Only *vs.* Large Class. Portanto, nossa classificação considera Complex Class, Brain Class, e Large Class Only separadamente do *smell* Large Class.

3.3 Resultados

As subseções a seguir detalham as observações de prevalência, co-ocorrência, co-estudo, evolução e diversidade relacionando os diferentes tipos de *smells*.

3.3.1 RQ1: Há *bad smells* significativamente mais estudados do que outros? Se sim, há alguma razão específica?

A Tabela 1, mostra todo o conjunto de *smells* estudados pelos artigos que estão no *Final Database*. A coluna *Together* mostra o número de artigos onde esses *smells* co-ocorrem com outros *smells*, a coluna *Alone* denota o número de artigos onde o *bad smell* foi estudado isoladamente no artigo e a coluna *Total* é a somatória das duas anteriores.

Analisando os dados da Tabela 1, observamos que os *smells* mais estudados são: (i) Duplicate Code; (ii) Large Class; (iii) Feature Envy e (iv) Long Method.

Tabela 1 – Bad smells ordenado pelo número de artigos no *Final Database*.

Bad Smells	[1]Together		[2]Alone		Total	
Duplicated Code	18	5,1%	227	64,7%	245	69,8%
Large Class (Blob Class, God Class)	79	22,5%	8	2,3%	87	24,8%
Feature Envy	46	13,1%	3	0,9%	49	14,0%
Long Method (God Method)	47	13,4%	1	0,3%	48	13,7%
Data Class	37	10,5%	0	0,0%	37	10,5%
Shotgun Surgery	32	9,1%	0	0,0%	32	9,1%
Refused Bequest	30	8,5%	0	0,0%	30	8,5%
Long Parameter List	26	7,4%	0	0,0%	26	7,4%
Spaghetti Code	23	6,6%	0	0,0%	23	6,6%
Message Chains Class	18	5,1%	0	0,0%	18	5,1%

continua na próxima página

continuação da página anterior

Bad Smells	[1]Together		[2]Alone		Total	
Few Methods (Lazy Class, Small Class)	17	4,8%	0	0,0%	17	4,8%
Abstract Class (Speculative Generality)	16	4,6%	0	0,0%	16	4,6%
Function Class (Func. Decomposition)	16	4,6%	0	0,0%	16	4,6%
Data Clumps	15	4,3%	0	0,0%	15	4,3%
Complex Class Only	14	4,0%	0	0,0%	14	4,0%
Swiss Army Knife	14	4,0%	0	0,0%	14	4,0%
Field Public (CDSBP)	13	3,7%	0	0,0%	13	3,7%
Divergent Change	12	3,4%	0	0,0%	12	3,4%
Misplaced Class	12	3,4%	0	0,0%	12	3,4%
Brain Method	9	2,6%	0	0,0%	9	2,6%
Temporary variable, several purposes	9	2,6%	0	0,0%	9	2,6%
Dispersed (Extensive) Coupling	8	2,3%	0	0,0%	8	2,3%
Intensive Coupling	8	2,3%	0	0,0%	8	2,3%
AntiSingleton	7	2,0%	0	0,0%	7	2,0%
Interface Segregation Principle Violation	7	2,0%	0	0,0%	7	2,0%
Switch Statements	7	2,0%	0	0,0%	7	2,0%
Tradition Breaker	7	2,0%	0	0,0%	7	2,0%
Unit Test Smells	1	0,3%	6	1,7%	7	2,0%
Duplicated code in conditional branches	6	1,7%	0	0,0%	6	1,7%
Large Class Only	6	1,7%	0	0,0%	6	1,7%
Schizophrenic class	6	1,7%	0	0,0%	6	1,7%
Use interface instead of implementation	6	1,7%	0	0,0%	6	1,7%
Brain Class	5	1,4%	0	0,0%	5	1,4%
Middle Man	4	1,1%	1	0,3%	5	1,4%
Ambiguous Interface	4	1,1%	0	0,0%	4	1,1%
Inappropriate Intimacy	4	1,1%	0	0,0%	4	1,1%
Parallel Inheritance Hierarchies	4	1,1%	0	0,0%	4	1,1%
Component Concern Overload	3	0,9%	0	0,0%	3	0,9%
Connector Envy	3	0,9%	0	0,0%	3	0,9%
Duplicate Pointcut	3	0,9%	0	0,0%	3	0,9%
God Pointcut	3	0,9%	0	0,0%	3	0,9%
Lexicon Bad Smells	1	0,3%	2	0,6%	3	0,9%
Primitive Obsession	3	0,9%	0	0,0%	3	0,9%
Redundant Pointcut	3	0,9%	0	0,0%	3	0,9%
Scattered Parasitic Functionality	3	0,9%	0	0,0%	3	0,9%
Smells in Android (Specific)	1	0,3%	2	0,6%	3	0,9%
Type Check (State Check)	3	0,9%	0	0,0%	3	0,9%
Anonymous Pointcut	2	0,6%	0	0,0%	2	0,6%
Classes with Different Interfaces	2	0,6%	0	0,0%	2	0,6%
Composition Bloat	2	0,6%	0	0,0%	2	0,6%
Controller Class	2	0,6%	0	0,0%	2	0,6%
Cyclic Dependency	2	0,6%	0	0,0%	2	0,6%
Extraneous Connector	2	0,6%	0	0,0%	2	0,6%
Forced Join Point	2	0,6%	0	0,0%	2	0,6%
God Aspect	2	0,6%	0	0,0%	2	0,6%
Idle Pointcut	2	0,6%	0	0,0%	2	0,6%
Instanceof	2	0,6%	0	0,0%	2	0,6%
Lava Flow (Dead Code)	2	0,6%	0	0,0%	2	0,6%
Lazy Aspect	2	0,6%	0	0,0%	2	0,6%

continua na próxima página

continuação da página anterior

Bad Smells	[1]Together		[2]Alone		Total	
Linguistic Antipatterns	0	0,0%	2	0,6%	2	0,6%
Low Cohesion Only	2	0,6%	0	0,0%	2	0,6%
Typecasts	2	0,6%	0	0,0%	2	0,6%
Wide Subsystem Interface	2	0,6%	0	0,0%	2	0,6%
Abstract Method Introduction	1	0,3%	0	0,0%	1	0,3%
Annotation Bundle	0	0,0%	1	0,3%	1	0,3%
BaseClassKnowsDerivedClass	1	0,3%	0	0,0%	1	0,3%
BaseClassShouldBeAbstract	1	0,3%	0	0,0%	1	0,3%
Borrowed Pointcut	1	0,3%	0	0,0%	1	0,3%
Child Class	1	0,3%	0	0,0%	1	0,3%
Class Global Variable	1	0,3%	0	0,0%	1	0,3%
Class One Method	1	0,3%	0	0,0%	1	0,3%
Comments	1	0,3%	0	0,0%	1	0,3%
Distorted Hierarchy	1	0,3%	0	0,0%	1	0,3%
Empty catch blocks	1	0,3%	0	0,0%	1	0,3%
Extraneous Adjacent Connector	1	0,3%	0	0,0%	1	0,3%
Field Private	1	0,3%	0	0,0%	1	0,3%
God Package	1	0,3%	0	0,0%	1	0,3%
Has Children	1	0,3%	0	0,0%	1	0,3%
Incomplete Library Class	1	0,3%	0	0,0%	1	0,3%
Junk Material	1	0,3%	0	0,0%	1	0,3%
Many Attributes	1	0,3%	0	0,0%	1	0,3%
ManyFieldAttributesButNotComplex	1	0,3%	0	0,0%	1	0,3%
Method No Parameter	1	0,3%	0	0,0%	1	0,3%
Multiple Interface	1	0,3%	0	0,0%	1	0,3%
No Inheritance	1	0,3%	0	0,0%	1	0,3%
No Polymorphism	1	0,3%	0	0,0%	1	0,3%
Not Abstract	1	0,3%	0	0,0%	1	0,3%
Not Complex	1	0,3%	0	0,0%	1	0,3%
Obsolete Parameter	1	0,3%	0	0,0%	1	0,3%
One Child Class	1	0,3%	0	0,0%	1	0,3%
Parent Class Provides Protected	1	0,3%	0	0,0%	1	0,3%
Promiscuous Package	1	0,3%	0	0,0%	1	0,3%
Rare Overriding	1	0,3%	0	0,0%	1	0,3%
Simulation of multiple inheritance	1	0,3%	0	0,0%	1	0,3%
Smells in CSS (Specific - DSL)	0	0,0%	1	0,3%	1	0,3%
Smells in JavaScript (Specific - DSL)	0	0,0%	1	0,3%	1	0,3%
Smells in MVC Arq, (Specific)	0	0,0%	1	0,3%	1	0,3%
Smells in Puppet (Specific - DSL)	0	0,0%	1	0,3%	1	0,3%
Two Inheritance	1	0,3%	0	0,0%	1	0,3%
Unused Interface	1	0,3%	0	0,0%	1	0,3%
Useless Class	1	0,3%	0	0,0%	1	0,3%
Useless Field	1	0,3%	0	0,0%	1	0,3%
Useless Method	1	0,3%	0	0,0%	1	0,3%
Various Concerns	1	0,3%	0	0,0%	1	0,3%

[1] Co-ocorrência de *bad smells* no mesmo artigo (ex.: Large Class e Feature Envy).

[2] Ocorrência única do *bad smell* (ex.: Duplicate Code ocorre isoladamente no artigo).

DSL: Domain Specific Language.

Duplicate Code. Analisando a Tabela 1, observamos que Duplicate Code surge no topo da lista, presente em 69,8% dos artigos. Note também que em 92,6% (227 dos 245 artigos) dos casos, Duplicate Code é estudado isoladamente (conforme descrito anteriormente, esses artigos foram categorizados como DCG).

Large Class. Este *smell* ocorre em 87 (24,8%) artigos. Entretanto, apenas uma pequena fração (8 artigos) estuda este *smell* isoladamente (ver Tabela 1).

Feature Envy. Este *smell* ocorre em 39,5% (49#) dos artigos classificados como OBSG. Comparando com o *smell* anterior (Large Class), observamos uma redução expressiva (43,6%) no número de artigos. Análogo ao Large Class, esse *smell* também foi esporadicamente estudado isoladamente (ver Tabela 1).

Long Method. Segundo a Tabela 1, este *smell* ocorre em 48 (13,7%) artigos. Analisando a coluna *Alone*, também observamos que esse *smell* foi esporadicamente estudado isoladamente. Numericamente, esse *smell* é similar ao Feature Envy.

Também observamos que os *smells* mais estudados estão relacionados às métricas de complexidade, tamanho e volume. Possivelmente, isso pode estar relacionado à percepção de ameaça, Palomba et al. (48) relatam: *"smells related to complex/long source code are generally perceived as an important threat by developers"*.

Analisando os artigos, extraímos os principais fatores usados pelos autores para justificar a escolha dos *smells*:

- A capacidade das ferramentas disponíveis (ex. `InCode`, `DECOR`, `iPlasma`) em tratar os *bad smells* (17, 52, 98);
- Os *bad smells* devem estar presentes no código fonte dos sistemas analisados com uma certa frequência (14, 17, 35, 33, 99);
- Popularidade e difusão entre os profissionais. Alguns artigos consideram apenas aqueles *bad smells* considerados como uma ameaça pelos desenvolvedores de software (17, 48, 91);
- Representatividade nos problemas de projeto (33, 47, 48, 99);
- A tradição e/ou consolidação na literatura científica (ex. (16, 15)) também é mencionado como um dos fatores (14, 35, 33, 99);
- A inter-relação de alguns *bad smells* também podem influenciar o conjunto de *smells* analisados pelos artigos, por exemplo, estudos de Large Class, na maior parte dos casos (90,8%), também envolvem outros *bad smells* como Long Method e/ou Data Class (ver Tabela 1). Entretanto, esse fator não é explicitamente mencionado na literatura;
- A simplicidade conceitual dos *smells* é outro ponto implícito que influencia no interesse, exemplificando, Duplicate Code requer apenas o entendimento do conceito de similaridade. Por outro lado, entender outros *smells* pode exigir a assimilação de vários conceitos, como o, Spaghetti Code que está vinculado a múltiplas características estruturais (herança, polimorfismo) e semânticas (nomes de classes e métodos que sugerem programação procedimental) (4).

Também investigamos os fatores que podem explicar porque Duplicate Code é significativamente mais estudado (ocorrendo em 69,8% dos artigos) do que os outros tipos de *smells*. Adicionalmente aos fatores anteriores, outro motivo do interesse nesse *smell* é que ele é extremamente versátil com várias aplicações. Os seguintes itens suportam a hipótese de versatilidade:

- ❏ este tipo de *smell* pode ser encontrado intra (100, 101, 102) e inter (43, 103, 104) projetos de software;
- ❏ este tipo de *smell* pode ser considerado independentemente da linguagem de programação (44, 105, 106, 107) e/ou paradigma (108, 109, 110, 111).

Dada a peculiaridade deste *smell* (Duplicate Code), que em relação aos outros *smells*, apresenta um número maior de artigos e ao fato de que esse *smell* quase sempre é estudado isoladamente, ao contrário do que ocorre com os outros tipos de *smells*, conjecturamos que Duplicate Code é um caso especial de *bad smell* que foi amplamente estudado de maneira diferente dos outros tipos de *smells*.

O segundo mais recorrente é o *smell* Large Class. Segundo Kim, Zimmermann e Nagappan (112) o processo de refatoração pode aumentar algumas métricas (ex. LOC) que estão associadas com o *bad smell* Large Class, a saber: *"preferentially refactored modules experience a higher rate of reduction in certain complexity measures, but increase LOC and crosscutting changes more than the rest of modules"*. Isso sugere que o *smell* Large Class pode ser encontrado mesmo em códigos refatorados, indicando que a remoção desse *smell* não é trivial, ajudando a explicar o grande interesse nele.

3.3.2 RQ2: Quais são as co-ocorrências de *smells* mais comuns?

A coluna *Together* na Tabela 1, considera o número de artigos onde cada *smell* co-ocorre com outros *smells*. Entretanto, essa coluna não detalha quais são essas co-ocorrências. Então, para os cinco *smells* mais estudados, a Tabela 2 detalha com quais *bad smells* eles co-ocorrem nos artigos (ex., Large Class co-ocorre com Long Method em 41 artigos).

Tabela 2 – Co-ocorrência dos principais *bad smells*

	Duplicated Code (18#\245#)		**Large Class (79#\87#)**		**Feature Envy (46#\49#)**		**Long Method (47#\48#)**		**Data Class (37#\37#)**	
# Papers	Large Class	13	Long Method	41	Large Class	40	Large Class	41	Large Class	37
	Feature Envy	11	Feature Envy	40	Long Method	31	Feature Envy	31	Feature Envy	25
	Long Method	11	Data Class	37	Shotgun Surgery	27	Refused Bequest	23	Shotgun Surgery	22
	Data Class	7	Refused Bequest	30	Data Class	25	Long Param. List	21	Long Method	18
	Long Param. List	7	Shotgun Surgery	30	Refused Bequest	20	Data Class	18	Refused Bequest	17
	Refused Bequest	7	Long Param. List	26	Data Clump	13	Shotgun Surgery	18	Data Clump	12
	Shotgun Surgery	7	Spaghetti Code	23	Long Param. List	13	Speculative Generality	14	Long Param. List	9
	Data Clump	6	Message Chains	17	Misplaced Class	12	Message Chains	13	Brain Method	8
	Schizophrenic Class	5	Lazy Class	16	Divergent Change	11	Complex Class Only	12	Misplaced Class	8
	Divergent Change	4	Func. Decomposition	16	Duplicated Code	11	Lazy Class	12	Duplicated Code	7
	...		...		...		...		...	

Nosso conjunto de dados mostra que apenas 18 dos 245 artigos, Duplicated Code (ver Tabela 1) co-ocorre com outros *bad smells* e este co-ocorre principalmente com aqueles que podem ser detectados com métricas de tamanho (ex. Large Class, Long Method), ver Tabela 2.

Diferentemente de Duplicated Code, em muitos artigos (90,8%), Large Class co-ocorre com outros *bad smells*, ver a coluna *Together* na Tabela 1. Este fato pode ser explicado pela sua característica intrínseca, a saber: estar relacionado à diversas responsabilidades da entidade. Neste caso, essa característica se sobrepõe à característica de outros *smells*. Então, a segunda coluna da Tabela 2 mostra os principais *bad smells* que co-ocorrem com Large Class no mesmo artigo. Observamos que os artigos que estudam Large Class: 1) 47,1% também considera Long Method; 2) 45,9% considera Feature Envy; 3) 42,5% considera Data Class, e 4) 34,4% considera Shotgun Surgery.

O *smell* Feature Envy co-ocorre com outros *smells*, especialmente com Large Class (81,6%) e/ou Long Method (63,2%). Contudo, a co-ocorrência de Feature Envy e Shotgun Surgery também é significante (55,1%).

Em 97,9% dos casos, Long Method co-ocorre com outros *bad smells*. Como mencionado previamente, este fato pode ser explicado pela alta correlação de sua ocorrência com métricas de tamanho/volume (ex. LOC). Os principais *bad smells* que co-ocorrem com Long Method são: 1) Large Class (85,4%), 2) Feature Envy (64,5%), 3) Refused Bequest (47,9%) e 4) Long Parameter List (43,7%).

Por fim, e segundo a Tabela 2, observamos que Data Class sempre co-ocorre com Large Class e este *smell* co-ocorre com os quatro principais *smells* (Duplicate Code, Large Class, Feature Envy, Long Method). Segundo Fowler e Beck (15), Data Class são classes que contêm apenas campos e métodos *getters* e *setters* para os campos da classe. Assim, é natural que esse *smell* co-ocorra com outros.

3.3.3 RQ3: Considerando os *smells* mais recorrentes, quais são as principais descobertas no co-estudo de *smells*?

Na subseção anterior, apresentamos as principais co-ocorrências de *bad smells* nos artigos. A co-ocorrência pode ser apenas uma coincidência. Assim, nesta subseção, investigamos não apenas a frequência dos *smells* estudados no mesmo artigo, mas aprofundamos nos objetivos e nos resultados relacionados à inter-relação encontrada entre os *smells* estudados no mesmo artigo.

Selecionamos os cinco *smells* mais recorrentes (ver Tabela 1), e aqueles artigos que apresentam co-ocorrência de *smells* (ver Tabela 2) foram examinados para encontrar os artigos de co-estudo de *smells*. Lembre-se que consideramos co-estudos aqueles artigos onde a co-ocorrência de *smells* no artigo é intencionalmente projetada para investigar

alguma inter-relação ou interação entre eles.

Nosso conjunto de dados apresenta 93 artigos que estudam um conjunto de *bad smells* (co-ocorrência) e eles foram classificados como *"apenas co-ocorrência"* ou "co-estudo". A Tabela 26 do apêndice detalha essa classificação para cada artigo OBSG, bem como apresenta uma lista dos *smells* estudados em cada artigo OBSG. A classificação de apenas co-ocorrência é o tipo mais comum (79,5%) de pesquisa. Em geral, esses artigos estão interessados em melhorar a *precision* e/ou *recall* das técnicas/ferramentas usadas para manipular os *bad smells*. Nesta categoria, temos artigos com diferentes tipos de estratégias (ex. baseado em métricas (2), algoritmos de aprendizado de máquina (58), algoritmos evolucionários paralelos (80)) e alguns deles (ex. estudos empíricos) comparam suas abordagens com abordagens de pesquisas anteriores (ex., (5) compara a eficácia de sua abordagem Bayesiana com a ferramenta DECOR (4)). Por outro lado, os artigos de co-estudos são focados em identificar as instâncias de *smells* que são relevantes (ex., instâncias[5] cuja refatoração melhora a manutenibilidade) segundo algum critério. Por exemplo, Oizumi et al. (13) co-estudaram *smells* para identificar problemas arquiteturais: eles sugerem que aglomerações de *smells* são significantemente melhores indicadores de problemas do código do que instâncias individuais. Similarmente, Yamashita e Moonen (35) sugerem que as interações entre *bad smells* afetam a manutenção. Também observamos que as interações entre *smells* são obtidas aplicando alguma técnica que correlaciona a co-ocorrência dos *smells* que estão no mesmo artefato (ex., *Principal Component Analysis* — PCA (35)) e então, os itens correlacionados, que são estatisticamente significantes, são analisados qualitativamente para fornecer um sentido semântico.

A Tabela 3 é similar à Tabela 2; entretanto, essa tabela detalha quais *bad smells* foram co-estudados (ex. Large Class foi co-estudado com Data Class em 8 artigos). Observamos que apenas 2 dos 18 artigos, onde Duplicate Code co-ocorre com outros *bad smells*, são artigos de co-estudo. Note também que esta observação (baixo número de co-estudos) ocorre para outros *smells*. Nos próximos parágrafos, reportamos os principais resultados do co-estudo de *smells*.

Tabela 3 – Co-estudo dos principais *bad smells*.

	Duplicated Code (2#\18#)		**Large Class (15#\79#)**		**Feature Envy (9#\46#)**		**Long Method (12#\47#)**		**Data Class (11#\37#)**	
# Papers	Feature Envy	2	Data Class	8	Large Class	8	Large Class	7	Large Class	8
	Large Class	2	Feature Envy	8	Long Method	6	Feature Envy	6	Feature Envy	5
	Long Method	2	Long Method	7	Data Class	5	Data Class	4	Long Method	4
	Data Clump	1	Shotgun Surgery	5	Shotgun Surgery	5	Divergent Change	4	Shotgun Surgery	4
	External Duplication	1	Controller Class	3	Divergent Change	3	Shotgun Surgery	4	Controller Class	3
	Internal Duplication	1	Divergent Change	3	ISP Violation	3	Class Global Variable	3	Data Clump	3
	Long Param. List	1	ISP Violation	3	Duplicated Code	2	Method No Parameter	3	Divergent Change	3
	Message Chains	1	Complex Class	2	Long Param. List	2	No Inheritance	3	ISP Violation	2
	Primitive Obsession	1	Duplicated Code	2	Temporary variable	2	No Polymorphism	3	Inappropriate Intimacy	1
	Schizophrenic Class	1	Long Param. List	2	Data Clump	1	Spaghetti Code	3	Long Param. List	1
	...		...		...		...			

[5] Algumas instâncias de *smells* não afetam a manutenibilidade, ex., Data Clump indica poucos problemas de manutenção (113).

Duplicate Code. A análise da subseção anterior revelou que este *smell* é amplamente estudado isoladamente, contudo, há alguns estudos com a co-ocorrência do *smell* Duplicate Code com outros *smells* e alguns deles são co-estudos. Liu et al. (30) usam a possível relação entre Long Method e Duplicate Code para apoiar a priorização de refatoração, isso porque, ao remover Duplicate Code, o Long Method também irá desaparecer. Segundo Fowler e Beck (15), essa relação existe: *"When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables,* Duplicate Code *cannot be far behind"*. Similarmente, Parnin, Görg e Nnadi (79) também reportam essa relação, a saber: *"the (long) method is often difficult to understand and may contain* Duplicate Code*"*. Entretanto, essa suposição não é verdade para todas as instâncias de Long Method e Duplicate Code (ex., refatorar os *clones* em um Long Method com poucas linhas clonadas, não necessariamente, remove o Long Method) e nenhum desses artigos detalha as situações onde a relação entre Long Method, Large Class e Duplicate Code é relevante. Então, consideramos que ainda é necessário identificar as situações em que a relação entre Duplicate Code e Long Method é relevante porque nem todos os Long Methods são causados por Duplicate Code e vice-versa. Similarmente, consideramos importante estudar a relação de Duplicate Code e Large Class, bem como suas implicações, porque o primeiro pode causar o último, mas o último não é provável que seja sempre uma consequência do primeiro.

Large Class. A literatura relaciona o *bad smell* Large Class aos *smells* Long Method e Long Parameter List por meio das métricas de volume/tamanho, a saber: (a) *"*Long Method *is a method with a high number of lines of code and a lot of variables and parameters are used"* (47); (b) *"Consider a parameter list long when the number of parameters exceeds 5"* (114). Por outro lado, Large Classes são altamente acopladas às Data Classes (115), que apresentam principalmente dependências com métodos Feature Envy (35). Alguns estudos relacionam Large Class ao *smell* Feature Envy (116, 35), e Long Methods ao *smell* Large Class (117, 35).

Palomba et al. (38) propuseram uma nova técnica de detecção de *bad smells* baseado na mineração do histórico de alterações (*commits* dos repositórios). Essa técnica é essencialmente útil para encontrar *smells* manifestados nas mudanças do código. Na verdade, eles hipotetizaram certas relações de *smells* para propor uma estratégia de detecção baseada em mudanças do código, como por exemplo que alterar Large Class com Long Methods, que muitas vezes são afetados por Feature Envy, sugere disparar alterações em várias partes do sistema, que sinaliza a ocorrência de Shotgun Surgery. Outra descoberta relacionada a essa relação é: *"the* God Class *and classes with* Shotgun Surgery *were changed more frequently than the other classes"* (117).

Feature Envy. Como mencionado anteriormente, este *smell* é amplamente estudado com outros *smells*, especialmente com Large Class e/ou Long Method. Isso é espe-

rado, porque esses *smells* estão relacionados à *"quantidade de responsabilidades que estão implementadas em uma entidade (método/classe)"*. Assim, se uma entidade implementa muitas responsabilidades, a probabilidade de que essa entidade esteja mais interessada em outras entidades também cresce e este comportamento está relacionado à definição de Feature Envy. Este tipo de relação foi reportado em (118, 116). O *smell* Shotgun Surgery pode ocorrer quando uma única responsabilidade é dividida entre um grande número de classes (sintoma de dispersão de código (119)). Assim, a dispersão das responsabilidades poderia potencialmente introduzir Feature Envy. No entanto, essa relação é obscura na literatura.

Long Method. Liu et al. (30) criaram um cenário da possível relação entre Feature Envy e Long Method, a premissa é que Long Methods resulta da combinação de *bad smells* (ex. Feature Envy e Duplicate Code). Em (35), eles mostram que a maioria das Large Classes também manifestam Long Methods, e muitas destas Large Classes acessam dados/métodos de outras áreas do sistema.

Data Class. Segundo Yamashita e Moonen (35), muitos dos artefatos com Data Class apresentam dependências com métodos Feature Envy. Pietrzak e Walter (116) reportam que em 92% dos casos de existência de Data Class também indicam a presença de Feature Envy. Estratégias para detectar Large Class, geralmente, consideram o tamanho (ex. LOC) e a ocorrência de Data Class (120). Segundo Moha et al. (28), uma Large Class está associada a diversas Data Classes. Em (116), Data Class sugere a existência de Large Class porque Data Class está relacionada com Feature Envy. Isso ajuda a explicar porque Data Class sempre co-ocorre com Large Class (ver Tabela 2).

Observamos que algumas relações de *smells*, mesmo com sentido semântico, ainda não foram explicitamente co-estudadas na literatura, ex., Fontana et al. (58) aponta que *"*Long Method *refers to methods that tend to centralize the functionality of a class"*, sugerimos que os clientes de uma classe que tem Long Method(s) provavelmente não precisam invocar métodos adicionais dessa classe. Em outras palavras, é improvável que os clientes invoquem métodos herdados da super-classe (121). Assim, conjecturamos uma possível relação entre Long Method e Refused Bequest. No entanto, apenas hipóteses podem ser encontradas. Um ponto interessante a observar é que os *smells* provavelmente estão inter-relacionados, especialmente, se eles ocorrerem em Large Classes. Em outras palavras, se o tamanho dos elementos de código crescer, provavelmente diferentes tipos de *smells* relacionados à essas grandes estruturas também aparecerão. Neste sentido, poderia pensar em *smells* para melhor caracterizar anomalias de grandes estruturas, bem como para usar essa caracterização na avaliação do impacto (negativo) nos produtos e processos e, possivelmente, orientar o processo de refatorar essas anomalias.

Karasneh et al. (122), investigaram a relação entre a qualidade dos modelos de projeto (UML) e o código fonte. Eles relatam que, em média, a proporção de classes nos modelos

de projeto e no código fonte que têm os mesmos *smells* é de 37%. Isso é uma evidência de que os *smells* podem surgir muito cedo, ou seja, na fase de projeto do sistema. O estudo em questão foi realizado com sete tipos de *smells* e eles não consideraram a possível interação entre os *smells*. Assim, conforme nossa classificação, este estudo é um exemplo de co-ocorrência de *smells*. Sugerimos que estudos empíricos que considerem as interações entre *smells* são necessárias.

3.4 Considerações Finais

Os dados coletados na revisão sistemática se apresentaram extremamente extensos, permitindo abordar diversos aspectos do tema *bad smell*. Contudo, muitos destes não foram detalhados neste capítulo porque não foram considerados relevantes para justificar o estudo empírico apresentado no capítulo seguinte, bem como os *smells* usados nele. Aspectos de como os pesquisadores dos diversos tipos de *smells* interagem entre si; onde os artigos sobre o tema são publicados; como o interesse pelo tema *bad smell* evoluiu ao longo do tempo e como esse interesse está distribuído entre os pesquisadores da área, são alguns dos pontos que não detalhamos neste capítulo. Contudo, a íntegra dessa revisão pode ser consultada e estudada pelo artigo *"A systematic literature review on bad smells — 5 W's: which, when, what, who, where"*, de nossa autoria, publicado no periódico *"IEEE Transactions on Software Engineering"* (39).

Assim, restringindo-se aos dados apresentados neste capítulo, os resultados da nossa revisão sistemática sugerem que alguns *smells* são mais recorrentes na literatura do que outros. Duplicate Code é o *bad smell* mais estudado, é interessante notar que o mesmo é largamente estudado/investigado isoladamente. Observamos que, em menor proporção, outros *smells* são recorrentes na literatura (Large Class, Feature Envy, Long Method), entretanto, estes co-ocorrem nos artigos com maior frequência do que o *smell* Duplicate Code. Além disso, em um certo grau, esses *smells* estão em sua maioria relacionados à métricas de volume/tamanho (ex. Large Class, Long Method). Complementarmente, observamos que a quantidade de artigos de co-estudo é relativamente pequena.

Com base nos dados, acreditamos que estudar a interação de *smells* seja importante para entender como a interação entre eles pode afetar a manutenibilidade do software, bem como pode auxiliar na confecção de técnicas/ferramentas que podem ser usadas pelos desenvolvedores na remoção de *smells*. Reforçamos que não encontramos estudos que avaliem como usar essa inter-relação de *smells*, especialmente na condição de priorização de *smells* à refatoração.

Neste sentido, no próximo capítulo, vamos apresentar um estudo empírico exploratório da interação dos *smells*: Duplicate Code, Large Class e Complex Class. O motivo na escolha destes *smells* são: (i) Duplicate Code é um *smell* largamente estu-

dado, entretanto, poucos artigos investigaram empiricamente a interação dele com outros *smells*. Mesmo que analiticamente, Fowler e Beck (15) e outros trabalhos já tenham descrito e conjecturado uma possível relação desse *smell* com *smells* relacionados à estruturas grandes e/ou complexas; (ii) semanticamente vemos uma possível relação entre os *smells* em questão (ver detalhes no próximo capítulo); (iii) esses *smells* são bem conhecidos pelos desenvolvedores/pesquisadores e estão bem consolidados na literatura; (iv) os *smells* em questão são comumente encontrados no código fonte de diversos projetos e a quantidade de instâncias deles é suficientemente grande para permitir uma análise estatística; (v) a existência de estudos (ex. Liu et al. (30)) que usam essas possíveis relações na refatoração, mesmo que não tenhamos estudos empíricos quantificando e analisando as mesmas.

Capítulo 4

Estudo Empírico Exploratório: Co-estudo de *Smells*

Conforme apresentado no capítulo anterior, a literatura é carente de estudos que investigam a possível inter-relação entre *smells*, em especial, como isso pode ser usado pelos desenvolvedores para melhorar e/ou monitorar a qualidade dos projetos de software. Nesse contexto, Macia et al. (26) mostraram que problemas de projeto são caracterizados pela existência de vários *bad smells* espalhados pelo código. De forma análoga, segundo Oizumi (27), vários *bad smells* interagem entre si, através das estruturas do código (ex. chamadas de métodos), a fim de refletir um problema de projeto no software. Assim, observamos indícios da inter-relação dos *smells*. Deste modo, detectar e/ou analisar os *bad smells* de forma isolada (ex. (4, 28)), limita a utilidade dos mesmos na identificação e resolução de questões complexas (ex. *design problem* (26), *design erosion* (20)).

Portanto, no presente capítulo, investigamos a inter-relação entre *bad smells*. Essa investigação é importante, pois sem o completo conhecimento das possíveis interações que ocorrem entre os *smells*, os desenvolvedores poderiam adotar ações equivocadas e/ou desnecessárias ao identificar a existência de *bad smells* no código. Informações sobre a interação entre os *bad smells* é especialmente útil quando o volume de *bad smells* nos projetos de software é elevado (24, 25), pois isso dificulta a tomada e execução de ações (ex. refatorar). A literatura apresenta indícios de que algumas interações de *smells* são prejudiciais à qualidade do software (ex. modularidade, compreensibilidade, reusabilidade). Portanto, cogitamos que essas devem ser refatoradas com maior prioridade. Por outro lado, outras interações de *smells* são inofensivas e podem permanecer no código do software. Nesse cenário, Liu et al. (30) consideram relevante toda ocorrência simultânea de Duplicate Code e Long Method. Liu et al. (30) priorizam a refatoração baseada na possível inter-relação de *bad smells*, a saber: Duplicate Code pode causar Long Method, consequentemente, ao remover Duplicate Code, o Long Method irá desaparecer. Entretanto, refatorar Duplicate Code não necessariamente remove Long Method, pois isso depende de quanto do Long Method é causado pelo Duplicate

Code. Portanto, a abordagem apresentada por Liu et al. (30) carece de refinamento nas condições de contorno, exemplo: adicionando condições que permitam separar os *smells* "inofensivos" daqueles que são "nocivos". Este fato revela a necessidade de caracterizar e quantificar as interações entre *bad smells*, especialmente aquelas interações contendo *smells* que os desenvolvedores apontam como ameaça à qualidade do software, exemplo: segundo Palomba et al. (48), God Class, Long Method, Spaghetti Code são nocivos.

Em nossa revisão sistemática encontramos na literatura um elevado número de tipos de *smells*, a saber: 104. Alguns destes são específicos, existindo apenas em determinados contextos: i) Aniche et al. (123), fornecem um catálogo de seis *smells* que são específicos de sistemas *web* que usam o padrão MVC (*Model View Controller*); ii) Hecht, Moha e Rouvoy (124), propõem o *smell* Internal Getter/Setter (IGS) que é específico para sistemas que executam em ambientes Android; iii) outros *smells* ocorrem apenas em determinadas linguagens de programação (ex. CSS (125), JavaScript (126), Puppet (127)) que são *Domain Specific Language* (DSL). Isso demonstra a diversidade e heterogeneidade de *smells*. Portanto, investigar a inter-relação entre todos os tipos de *bad smells* resulta em um grande número de combinações. Assim, por questões de viabilidade, é necessário focar em determinadas combinações. Deste modo, nosso estudo empírico exploratório está focado nos *smells* que estão consolidados na literatura; que são bastante difundidos na comunidade dos desenvolvedores e que apresentam intuitivamente alguma relação semântica. Estudamos especificamente a inter-relação dos *smells* Duplicate Code (DC), Large Class (LC) e Complex Class (CC). Uma relação semântica fora conjecturada por Fowler e Beck (15) e outros trabalhos, pois de alguma forma o *smell* Duplicate Code se relaciona com entidades grandes e/ou complexas. Intuitivamente, entidades complexas com muitas estruturas de desvio de fluxo, têm maior chance de apresentar códigos clonados. Isso porque na presença de muitas expressões condicionais que se diferem apenas nas condições[1], certamente executam o mesmo código e, portanto, estes fragmentos são códigos clonados. Por outro lado, estruturas grandes possivelmente são mais complexas. Contudo, os projetos de software podem apresentar classes grandes que não são complexas (ex. classes com muitas funcionalidades distintas e, proporcionalmente, poucas estruturas de desvio de fluxo), ou mesmo, classes complexas que não são grandes (ex. classes com poucas funcionalidades distintas mas, cada uma, apresenta diversas estruturas de desvio de fluxo). Na prática, intuí-se que o número de funcionalidades e/ou estruturas de desvio de fluxo têm influência na prevalência de clones. Por isso, neste estudo empírico, a análise dos clones é feita agrupando-se as entidades conforme os *smells* que ocorrem em cada uma (*Large, Complex* e *Large+Complex*). Portanto, o estudo da inter-relação de *smells* ocorre considerando a existência isolada deles, bem como suas co-ocorrências. A próxima subseção detalha as etapas e pressupostos que

[1] <https://refactoring.guru/smells/duplicate-code>

guiaram a realização deste estudo empírico.

4.1 Planejamento Experimental

O objetivo do estudo experimental descrito neste capítulo é quantificar a associação entre os *smells* Large Class, Complex Class e Duplicate Code. Possibilitando a identificação de padrões de codificação usado pelos desenvolvedores, exemplo hipotético: Quando uma classe apresenta simultaneamente os *smells* Large e Complex Class, geralmente, seus clones são internos, ou seja, entre os métodos que constituem essa classe. Este tipo de informação é útil para, por exemplo, propor técnicas/métodos de refatoração, como aquela descrita por Liu et al. (30) que estabelece a ordem de refatoração de certos *smells*. Contudo, a análise de associação quantifica a relação mas não estabelece causa-efeito. Portanto, para atingir nosso objetivo, também realizamos uma análise qualitativa dos dados. Adicionalmente, no Capítulo 5, iremos investigar o histórico de versão (*commits*) dos projetos de software para determinar como os clones surgem nas classes e se existe alguma relação temporal dele com outros *smells* (Large e/ou Complex Class).

Assim, essa seção estabelece o delineamento conceitual dos detalhes envolvidos no plano experimental. Para isso, nas próximas subseções contextualizamos a terminologia usada nas questões de pesquisa e as definimos formalmente. Na sequência, descrevemos os itens envolvidos na metodologia empregada.

4.1.1 Aspectos Introdutórios

O objetivo desta subseção consiste em abordar os conceitos que dizem respeito ao planejamento experimental, evidenciando os aspectos estratégicos adotados na presente investigação empírica.

4.1.1.1 Caracterização da Terminologia de Inter-Relação de *Smells*

No primeiro momento, devemos definir e detalhar quais as possíveis formas de inter-relação de *smells*, exemplo: **co-ocorrência**, **dependência/associação**, **distância dos *commits***. Assim, considere os exemplos descritos nos próximos parágrafos.

A classe *C1* (Código 1) apresenta três tipos diferentes de *smell* ($S1_C$, $S2_C$, $S3_C$). Por outro lado, a classe *C2* (Código 2) demonstra, isoladamente, outro tipo de *smell* ($S4_C$). Observe que as classes apresentam quatro tipos de *smells* diferentes e estes ocorrem na granularidade de classes/arquivo, indicado pelo sufixo "$_C$". De forma análoga, mas na granularidade de método, as duas classes apresentam outros quatro tipos diferentes de *smells* ($S5_M$, $S6_M$, $S7_M$, $S8_M$). Note que os métodos *M4* e *M2* apresentam o mesmo *smell* ($S6_M$).

Código 1 – Classe C1.

```
1  public class C1 { S1_C, S2_C, S3_C
2   void M1() { S5_M
3    M5();
4    M4();
5   }
6   void M2() { S5_M,S6_M
7    M5();
8    M6();
9   }
10  void M3() { S8_M
11   M2();
12   M5();
13  }
14 }
```

Código 2 – Classe C2.

```
1  public class C2 { S4_C
2   void M4() { S6_M
3    ...
4   }
5
6   void M5() { S7_M
7    ...
8   }
9
10  void M6() {
11   ...
12  }
13
14 }
```

No exemplo acima, usando a **estrutura do código**, os *smells* podem ser inter-relacionados por meio da: i) **co-ocorrência**: no caso da classe *C1*, os *smells* $S1_C$, $S2_C$, $S3_C$, $S5_M$, $S6_M$, $S8_M$ estão no mesmo arquivo e podem ser agrupados conforme sua granularidade (classe/método); ii) **Dependência/Associação**: considerando as invocações realizadas pelo método *M2*, os *smells* $S1_C$, $S2_C$, $S3_C$, $S4_C$, $S5_M$, $S6_M$, $S7_M$ estão inter-relacionados e também podem ser agrupados conforme sua granularidade (classe/-método).

Outra forma de inter-relacionar os *smells* pode ser por meio do histórico de alterações do código (*commits*). Observe que os *smells* dos métodos *M3* e *M4* não apresentam inter-relacionamento pela estrutura do código. Contudo, ao analisar os históricos de *commits* observamos que estes métodos foram alterados simultaneamente para corrigir determinado *bug*. Assim, considerando a correção do *bug*, podemos inferir algum tipo de relação semântica/sistêmica entre esses métodos, visto que a **distância dos *commits*** é zero pelo fato de que estes métodos foram alterados no mesmo *commit*.

Pelos casos apresentados, podemos observar que há diversas formas de inter-relacionar os *smells*. Em geral, as formas surgem a partir da inferência de algum tipo de relação semântica e/ou sistêmica entre as entidades que apresentam instância(s) de tipos de *smell*. Por considerar a **co-ocorrência** na mesma entidade a forma mais direta e simples de coletar os dados, bem como de fornecer sentido semântico à inter-relação dos *smells*, usaremos ela neste estudo empírico. Observe que devido à escolha dos *smells* usados nesse estudo empírico (Large/Complex Class), o termo entidade se refere à granularidade de "classes" dos sistemas. Portanto, estudamos a co-ocorrência de *smells* no nível de classe.

4.1.1.2 Caracterização da Terminologia de Intensidade dos *Smells*

Outro aspecto abordado neste estudo empírico refere-se à intensidade dos *smells*. Em geral, a intensidade é uma forma de quantificar o quão crítico é cada instância de *smell*, em comparação com outras instâncias de *smell* (128). Essa quantificação pode ocorrer de diversas formas: Steidl e Eder (129) propuseram uma abordagem baseada em duas técnicas de refatoração (*pull-up method, extract method*) que considera o tamanho (LOC) e quantidade de parâmetros necessários para completar a refatoração. Por outro lado, Fontana et al. (128) consideram o *threshold* das métricas usadas na detecção dos *smells* para agrupá-los por nível de intensidade, em especial, sua abordagem é baseada no valor das métricas e à medida que seu valor aumenta, a classificação de quão crítico é o *smell* também aumenta (*Very-Low, Low, Mean, High, Very-High*).

No contexto de nosso estudo empírico, considerar a intensidade dos *smells* é importante para identificar e verificar, se à medida que os *smells* se tornam mais críticos, se eles acabam associando-se a outros tipos de *smells* (ex. à medida que a classe se torna mais complexa, a prevalência de outros tipos de *smells* também aumenta). Assim, neste capítulo, usaremos a distribuição das métricas de detecção de *smells* para classificar a intensidade dos mesmos. Para isso, separamos apenas as classes do sistema que foram caracterizadas com um dado *smell* (ex. Complex Class) e coletamos o valor da métrica usada para detectar o *smell* em questão. Assim, obtemos um conjunto de dados com o valor da métrica e o usamos para discretizar em dois níveis de intensidade. Outros detalhes estão estabelecidos na subseção 4.1.3.

A próxima subseção detalha as questões de pesquisa que conduziram à elaboração metodológica e execução experimental.

4.1.2 Questões de Pesquisa

Essa subseção apresenta as perguntas de pesquisa (*Research Questions* — RQs) consideradas nesse estudo empírico sobre a co-ocorrência de *smells* e seus níveis de intensidade. As questões de pesquisa estão agrupadas em duas categorias: (i) na primeira estão as questões relativas à intensidade dos *smells*; (ii) na sequência, temos aquelas relacionadas à quantidade de *smells*.

I) A intensidade dos *smells*.

A intensidade do *bad smell*, ocorrendo isoladamente e/ou em co-ocorrência com outros, é usada para verificar a associação de prevalência com outro(s) *smell(s)*.

Neste conjunto de questões, desejamos avaliar o que acontece com a prevalência de *clones* quando a intensidade de um dado *smell* ou conjunto de *smells* se torna mais crítico, exemplo hipotético: a prevalência de *clones* apresenta relação

direta com o incremento da complexidade de uma entidade (classe) considerada Large Class.

Questões de Pesquisa:

RQ1.1 A intensidade do *smell* Complex Class está associada à prevalência de *clones*? Se sim, quanto ela está associada?

A Figura 7 representa graficamente a RQ1.1. Neste caso, ela descreve a associação de clones entre as entidades menos complexas e aquelas mais complexas.

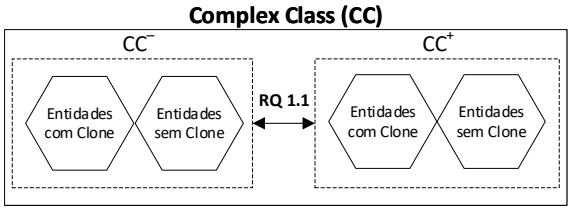


Figura 7 – Diagrama de representação da RQ1.1

RQ1.2 A intensidade do *smell* Large Class está associada à prevalência de *clones*? Se sim, quanto ela está associada?

A Figura 8 representa graficamente a RQ1.2. Neste caso, ela descreve a associação de clones entre as entidades menores e aquelas que são maiores.

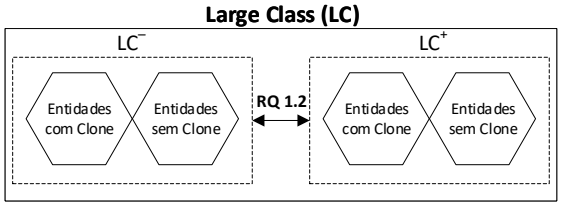


Figura 8 – Diagrama de representação da RQ1.2

RQ1.3 A intensidade da co-ocorrência, na mesma entidade (classe), dos *smells* Complex Class e Large Class está associada à prevalência de *clones*? Se sim, quanto ela está associada?

A Figura 9 representa graficamente a RQ1.3. Neste caso, ela descreve a associação de clones entre as entidades acometidas pela co-ocorrência dos *smells* Large e Complex Class, conforme a intensidade de cada *smell*. Como estes *smells* co-existem, a análise dos dados deve ocorrer: (i) considerando a intensidade do *smell* Complex Class (CC^{-}&$LC^{\text{qualquer nível (x)}}$ *vs* CC^{+}&$LC^{\text{qualquer nível (x)}}$) e (ii) tomando como base a intensidade do *smell* Large Class (LC^{-}&$CC^{\text{qualquer nível (x)}}$ *vs* LC^{+}&$CC^{\text{qualquer nível (x)}}$). Portanto, temos dois modelos que devem ser comparados a fim de identificar aquele que melhor se adéqua aos dados.

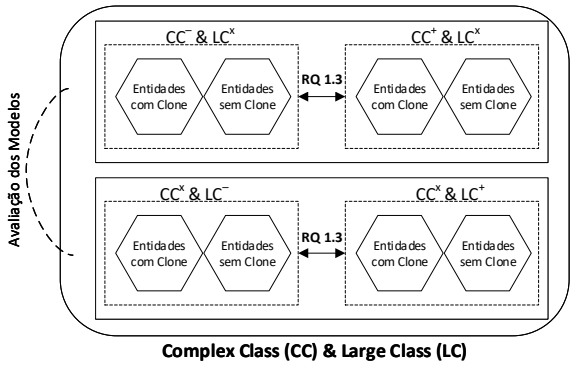


Figura 9 – Diagrama de representação da RQ1.3

II) A quantidade de tipos de *smells*.

Desconsiderando a intensidade dos *smells*, o incremento na quantidade de tipos de *smells* que co-ocorrem na mesma entidade é usado para verificar a associação de prevalência com outro(s) *smell(s)*.

Neste conjunto de questões, desejamos avaliar o que acontece com a prevalência de *clones* quando a quantidade de tipos de *smell* que co-ocorrem na mesma entidade (classe) se agrava, exemplo hipotético: em relação às entidades com apenas[2] um tipo de *smell*, a prevalência de *clones* aumenta significativamente quando as entidades apresentam mais de um tipo de *smell* (Large & Complex Class).

Questões de Pesquisa:

RQ2.1 Desconsiderando a intensidade dos *smells*, existe diferença na prevalência de clones entre as entidades que apresentam co-ocorrência de *smells* (LC/CC) e as entidades que apresentam apenas o *smell* CC? Se sim, quanto esse comportamento está associado?

A Figura 10 representa graficamente a RQ2.1, ela descreve a associação de clones entre as entidades com apenas o *smell* CC e aquelas onde os *smells* LC e CC co-existem. Contudo, nessa RQ, a intensidade dos *smells* não é considerada.

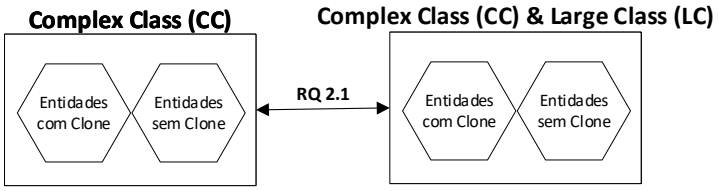


Figura 10 – Diagrama de representação da RQ2.1

[2] Neste estudo empírico as expressões: (i) "entidades/classes com apenas o *smell* LC"; (ii) "entidades/-classes com apenas o *smell* CC", quer dizer que dentre os *smells* estudados neste capítulo, as classes em questão não apresentam a co-ocorrência simultânea dos *smells* LC e CC. Por outro lado, estas entidades podem apresentar *smells* que não são alvos destes estudos (ex. Spaghetti Code). Observe que essas expressões são usadas no restante deste capítulo.

RQ2.2 Desconsiderando a intensidade dos *smells*, existe diferença na prevalência de clones entre as entidades que apresentam co-ocorrência de *smells* (LC/CC) e as entidades que apresentam apenas o *smell* LC? Se sim, quanto esse comportamento está associado?

A Figura 11 representa graficamente a RQ2.2, ela descreve a associação de clones entre as entidades com apenas o *smell* LC e aquelas onde os *smells* LC e CC co-existem. Contudo, nessa RQ, a intensidade dos *smells* não é considerada.

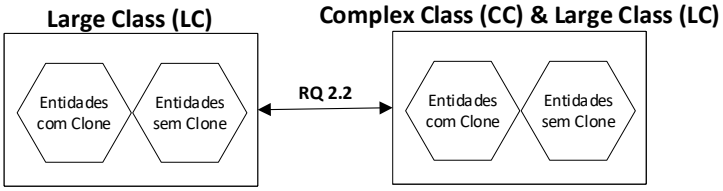


Figura 11 – Diagrama de representação da RQ2.2

4.1.3 Materiais e Métodos

Para responder às questões de pesquisa apresentadas na subseção anterior devemos: (i) coletar os dados necessários e (ii) modelá-los conforme alguma técnica, que possibilite a comparação dos modelos e ainda permita quantificar o impacto dos fatores nos modelos. Portanto, as próximas subseções detalham estes procedimentos.

4.1.3.1 Coleta de Dados

A decisão de quais informações devem ser coletadas é baseada nas questões de pesquisa, assim o primeiro passo é identificar as classes dos projetos alvo do estudo que são acometidas pelos *smells* Duplicate Code, Complex e/ou Large Class. O segundo passo é estruturar e consolidar os dados coletados, permitindo que sua análise responda as questões de pesquisa. Neste ponto, usaremos a ferramenta PMD[3] para detectar os clones e o utilitário DECOR[4] (4) para os demais *smells*, os detalhes das estratégias usadas nessas ferramentas estão descritos nas próximas duas subseções. Escolhemos essas ferramentas porque elas são de código fonte aberto, o que permite verificar a implementação e ainda possibilita a confecção de customizações. Outro aspecto que motivou essa escolha é o seu amplo uso na comunidade de desenvolvedores e pesquisadores.

Detectando *Smells* e Parametrização das Ferramentas

Conforme apresentado na Subseção 2.3.1, a ferramenta PMD pode ser usada para detectar clones e ela possui alguns parâmetros de configuração. O primeiro deles solicita que

[3] <https://pmd.github.io/>
[4] *DEtection & CORrection* — <https://bitbucket.org/ptidejteam/>

o usuário informe um *threshold* mínimo para o tamanho dos clones. Nos experimentos apresentados neste capítulo, utilizamos o valor de 75 *tokens* como limiar de tamanho mínimo dos clones. Esse valor é o padrão definido pela ferramenta, contudo, outros valores (50 e 100) também foram analisados. Os resultados experimentais obtidos para o valor 50 revela que este limiar, em relação ao valor 75, aumenta consideravelmente a quantidade de clones detectados. Isso porque clones pequenos (ex. 8 linhas) passam a ser considerados. Assim, este limiar foi descartado devido à grande quantidade de clones considerados irrelevantes. Por outro lado, quando testamos o limiar no valor 100, observamos que apenas os clones maiores são retornados. Isso diminuiu significativamente o número de classes com clones, o que inviabiliza a execução da técnica estatística detalhada na Subseção 4.1.3.2. Assim, experimentalmente observamos que o limiar mais adequado foi aquele parametrizado na ferramenta (75 *tokens*). Além disso, nos experimentos, configuramos a ferramenta para detectar tanto os clones do tipo I quanto do tipo II. Por fim, também ligamos o parâmetro que descarta as linhas referentes a comentários de código. Assim, os fragmentos considerados clones não consideram seus comentários de código.

A ferramenta PMD produz uma saída com a quantidade de linhas clonadas, as próprias linhas tidas como clones, o número de *tokens* destas linhas e o nome do(s) arquivo(s) onde este fragmento clonado se encontra. O Código 3 apresenta um fragmento desse arquivo de saída. Além das informações explicitamente apresentadas, realizando o mapeamento dos clones, também é possível dizer se os clones são: i) **inter-classes** ocorrendo apenas entre diferentes entidades no nível de classe, como é o caso dos clones entre os arquivos *TestLucene70DocValuesFormat.java* e *TestLucene54DocValuesFormat.java* apresentados no primeiro fragmento de clone do Código 3; ii) **intra-classe** é quando o fragmento de código clonado ocorre apenas dentro de uma entidade, mas em linhas diferentes. Esse é o caso do segundo exemplo apresentado no Código 3, o arquivo *BasePostingsFormatTestCase.java* apresenta 13 linhas de códigos duplicados, que ocorre em duas posições distintas do arquivo *UnicodeUtil.java*; iii) **mix-classes** que refere-se aos clones que ocorrem inter- e intra-classe simultaneamente. Essa classificação dos clones baseado no local de ocorrência será usado em nossas análises.

Além da ferramenta PMD, também usamos a ferramenta DECOR. Ela é usara para detectar os *smells* Large Class e Complex Class. Conforme apresentado na Subseção 2.3.2, a parametrização dos limiares é baseada em uma técnica que dispensa a intervenção manual do usuário, ou seja, não é necessário informar manualmente os limiares das métricas usadas na detecção dos *smells*. Portanto, para nosso estudo, apenas executamos a ferramenta DECOR para obter um lista das classes que apresentam algum dos *smells* de interesse.

Código 3 – Fragmento de saída da detecção de *clones* (ferramenta PMD)

Found a 131 line (972 tokens) duplication in the following files:
Starting at line 153 of
lucene/core/src/test/org/apache/lucene/codecs/lucene70/TestLucene70DocValuesFormat.java
Starting at line 141 of
lucene/backward-codecs/src/test/org/apache/lucene/codecs/lucene54/TestLucene54DocValuesFormat.java

```
1  @Slow
2  public void testSparseDocValuesVsStoredFields() throws Exception {
3   int numIterations = atLeast(1);
4   for (int i = 0; i < numIterations; i++) {
5    doTestSparseDocValuesVsStoredFields();
6   }
7  }
8  private void doTestSparseDocValuesVsStoredFields() throws Exception {
9   final long[] values = new long[TestUtil.nextInt(random(), 1, 500)];
10  for (int i = 0; i < values.length; ++i) {
11   values[i] = random().nextLong();
12  }
13  Directory dir = newFSDirectory(createTempDir());
14  IndexWriterConfig conf = newIndexWriterConfig(new MockAnalyzer(random()));
15  conf.setMergeScheduler(new SerialMergeScheduler());
16  RandomIndexWriter writer = new RandomIndexWriter(random(), dir, conf);
17  ...
```

Found a 13 line (143 tokens) duplication in the following files:
Starting at line 142 of
lucene/core/src/java/org/apache/lucene/util/UnicodeUtil.java
Starting at line 200 of
lucene/core/src/java/org/apache/lucene/util/UnicodeUtil.java

```
1   if (code < 0x80)
2    out[upto++] = (byte) code;
3   else if (code < 0x800) {
4    out[upto++] = (byte) (0xC0 | (code >> 6));
5    out[upto++] = (byte)(0x80 | (code & 0x3F));
6   } else if (code < 0xD800 || code > 0xDFFF) {
7    out[upto++] = (byte)(0xE0 | (code >> 12));
8    out[upto++] = (byte)(0x80 | ((code >> 6) & 0x3F));
9    out[upto++] = (byte)(0x80 | (code & 0x3F));
10  } else {
11   // surrogate pair
12   // confirm valid high surrogate
13   if (code < 0xDC00 && i < end) {
```

Estruturando e Consolidando a Matriz de Dados

Os dados fornecidos pelas ferramentas PMD e DECOR foram organizados em uma matriz de dados (Tabela 4). Essa matriz é formada por diversas linhas, uma para cada classe do sistema que está sendo analisado. Assim, o campo "$ClasseID$" identifica cada classe usando seu respectivo nome. As colunas $Smell_{DC}$, $Smell_{LC}$ e $Smell_{CC}$, descrevem de forma binária (Verdadeiro ou Falso) se determinada classe apresenta, respectivamente, os *smells* Duplicate Code, Large Class e Complex Class. Com relação ao *smell* Duplicate Code, essa matriz também apresenta duas informações complementares: (i) $Tipo_{DC}$ que classifica o(s) clone(s) da classe conforme a dispersão no sistema (Inter-Classe, Intra-Classe e Mix-Classe); (ii) $Métrica_{DC}$ que representa, numericamente e em termos de *tokens*, o tamanho do(s) clone(s) encontrado(s) na classe analisada. Analogamente a este último campo, as colunas $Métrica_{LC}$ e $Métrica_{CC}$ também são informações numéricas que

representam, respectivamente, as métricas NMD[5]$+NAD$[6] e $McCabe$ da classe analisada.

Tabela 4 – Estrutura geral da matriz de dados

Classe ID	**PMD**			**DECOR**			
	[1]$Smell_{DC}$	[2]Tipo_{DC}	[3]$\text{Métrica}_{\text{DC}}$	[1]$Smell_{LC}$	[3]$\text{Métrica}_{\text{LC}}$	[1]$Smell_{CC}$	[3]$\text{Métrica}_{\text{CC}}$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

DC — Duplicate Code; LC — Large Class; CC — Complex Class.
[1] Variável binária (*dummy*).
[2] Variável categórica (Inter-Classe, Intra-Classe e Mix-Classe).
[3] Variável discreta.

Analisando as questões de pesquisa (Subseção 4.1.2), observa-se que elas foram elaboradas para verificar a possível associação entre Duplicate Code e os *smells* Large e Complex Class. Assim, as questões buscam verificar essa associação em relação a dois aspectos: (i) quanto a intensidade dos *smells* e (ii) quanto a quantidade de tipos de *smells*.

Portanto, para responder as questões de pesquisa é necessário reorganizar a matriz de dados (Tabela 4). Para isso, selecionamos apenas aquelas classes que apresentam ao menos um dos seguintes *smells*: Large, Complex Class. Para estes dados, incluímos a coluna $Smell_{DC}$ da Tabela 4, na coluna denominada $Smell_{PMD}$ da Tabela 5. Complementarmente, também incorporamos as colunas $Smell_{LC}$ e $Smell_{CC}$ da Tabela 4, na coluna $Smell_{DECOR}$ (Tabela 5) que é descrita por meio de uma variável categórica. Neste caso, a categorização ocorre usando a representação dos dados, ou seja, uma combinação dos *smells* Large e Complex Class. O resultado é uma coluna que inicialmente apresenta três categorias: (i) LC para as classes que apresentam apenas Large Class; (ii) CC para aquelas que apresentam apenas Complex Class e (iii) $LC\&CC$ para aquelas que apresentam a co-ocorrência dos *smells* Large Class e/ou Complex Class. Isso permitirá verificar o que acontece com os clones quando a prevalência dos outros *smells* varia, ou seja, permite responder as duas últimas questões de pesquisa apresentadas na Subseção 4.1.2.

Para responder as outras questões de pesquisa é necessário que a intensidade dos *smells* Large e Complex Class também seja incorporada na coluna $Smell_{DECOR}$ da Tabela 5. Neste sentido, as colunas $Métrica_{LC}$ e $Métrica_{CC}$ da Tabela 4 são usadas na discretização em n níveis, pois elas refletem a grandeza das métricas destes *smells*. Contudo, como a coluna $Smell_{DECOR}$ é representada por uma variável categórica, os dados numéricos das métricas devem ser discretizados em um número determinado de níveis ($Nível_{1...n}$). A ideia é que o nível inicial acomode as classes onde o *smell* em questão se apresenta em uma intensidade menor do que aquelas classes que estão acomodadas nos níveis subsequentes. Na prática, isso quer dizer que as três categorias $LC, CC, LC\&CC$ devem ser expandidas de forma a acomodar os n níveis de intensidade dos respectivos *smells*. Assim, as

[5] *Number of Methods Declared*
[6] *Number of Attributes Declared*

categorias podem ser representadas como $LC_{1...n}, CC_{1...n}, LC_{1...n}\&CC_{1...n}$, onde o índice representa o respectivo nível de intensidade do *smell*. Observe que a categoria representada pela co-ocorrência de *smells* deve ser desdobrada conforme a quantidade de *smells* que co-ocorrem e a quantidade de níveis de intensidade de cada um deles ($Nº Níveis^{Nº Smells}$). Assim, como exemplo, considerando a co-ocorrência de dois *smells* (LC, CC) e tendo dois níveis de intensidade cada, essa categoria é desmembrada em quatro categorias, que são representadas como: $LC_1\&CC_1$, $LC_1\&CC_2$, $LC_2\&CC_1$, $LC_2\&CC_2$.

Tabela 5 – Matriz de dados consolidado conforme RQs

Classe ID	[1] $Smell_{PMD}$	[2] $Smell_{DECOR}$
⋮	⋮	⋮

[1] Variável binária (*dummy*).
[2] Variável categórica (LC_n — Large Class; CC_n — Complex Class; $LC_n\&CC_n$ — Large & Complex Class).

A discretização do valor contido na coluna $Métrica_{LC}$ e $Métrica_{CC}$ ocorre usando sua distribuição. Para isso usamos os dados da Tabela 4, selecionando apenas as classes acometidas pelo *smell* que está sendo analisado (LC ou CC). Com estes dados, gera-se o *boxplot* da métrica ($NMD + NAD$ *ou* $McCabe$). Assim, por meio dos quartils ocorre a discretização em n níveis (ver Figura 12). Nos experimentos descritos neste capítulo discretizamos a intensidade em dois níveis, neste caso, consideramos que no primeiro nível (baixo) estão aquelas classes em que o valor da métrica está abaixo da mediana e no segundo nível (alto) estão as outras classes. Esta quantidade de níveis foi escolhida experimentalmente pela comparação com os resultados usando três e quatro níveis. Observamos que à medida que o número de níveis aumenta, menor é a quantidade de projetos que satisfazem as condições da técnica estatística de regressão logística. Assim, a quantidade com menor descarte de projetos na fase de análise dos requisitos estatísticos foi de dois níveis e, por isso, esse valor é usado nos experimentos descritos neste capítulo.

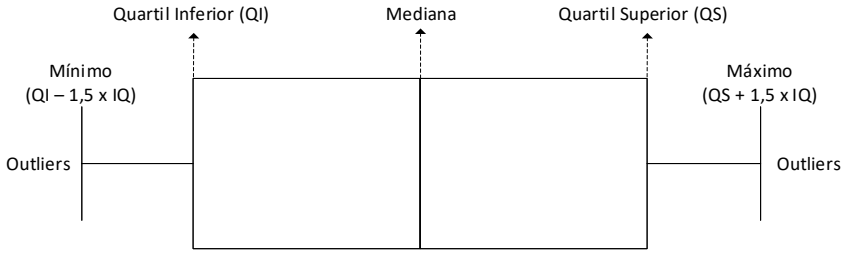


Figura 12 – Representação do Boxplot.

4.1.3.2 Técnica de Modelagem de Dados

A análise de regressão logística é um caso especial do modelo linear generalizado, usado para analisar modelos onde a variável de resposta (ou dependente) é uma variável dicotômica/binária, ou seja, que assume apenas dois possíveis valores (ex. "0" ou "1")

(130). O principal objetivo deste modelo é explorar a relação entre uma ou mais variáveis explicativas (ou independentes/preditoras) e uma variável resposta (ou dependente).

Segundo diversas literaturas, o uso da regressão logística deve satisfazer as seguintes premissas:

- **Linearidade** (*Linearity*). Este critério visa verificar se existe alguma relação linear entre qualquer um dos preditores contínuos e a variável de resposta. Essa suposição pode ser avaliada olhando a significância da interação entre os termos preditores contínuos e suas respectivas transformações *log*[7] (131, pg. 344). Para o estudo de caso apresentado neste capítulo, essa suposição não precisará ser testada pois as variáveis preditoras são categóricas.

- **Independência dos Erros**. Para quaisquer pares de observações, os termos residuais não devem ser correlacionados, algumas vezes essa suposição é descrita como falta de auto-correlação. Essa suposição pode ser verificada pelo teste *Durbin–Watson* (Na ferramenta R® usa-se a função *durbinWatsonTest()*), quando a estatística deste teste estiver entre 1 e 3, quer dizer que a independência dos erros é garantida (131, pg. 292). Ainda segundo Field e Miles (131, pg. 321) caso cada uma das observações da amostra seja independente e não relacionada umas as outras, pode-se assumir a independência dos erros. Assim, considerando que as entidades analisadas no estudo apresentado neste capítulo são independentes e não relacionadas, podemos assumir a independência dos erros.

- **Multicolinearidade**. Em essência, os preditores **não** devem estar altamente correlacionados. Essa suposição pode ser verificada analisando a tolerância do teste estatístico VIF (*Variance Inflation Factor*). Usualmente, os valores acima de 10 são considerados problemáticos (131, pg. 343). Essa condição é necessária apenas quando o modelo usa dois ou mais preditores simultaneamente, o que não é o caso do estudo empírico apresentado neste capítulo.

- **Homocedasticidade**. Isso significa que os resíduos em cada nível do(s) preditor(es) devem ter a mesma variação. Diferentemente do que acontece na análise de regressão convencional, essa suposição **não** precisa ser atendida na regressão logística pois este é um teste não paramétrico (132, pg. 14) (133, pg. 449).

- **Independência das Observações**. Supõe-se que todos os valores da variável de resposta são independentes, ou seja, cada valor dessa variável provém de uma entidade diferente (131, pg. 272).

- **Variável Dependente**. Na regressão logística, os valores da variável dependente **não** precisam estar normalmente distribuídos (133, pg. 449).

[7] Cálculo do logaritmo natural

- **Tabela de Contingência**. Considerando as variáveis categóricas, sua representação nesse tipo de tabela **não** pode apresentar células com valores zerados (133). Complementarmente, o valor esperado das células deve ser maior ou igual a cinco (131). Por outo lado, segundo Dunn e Clark (134) o valor esperado de cada célula deve ser maior que dois ou três. Caso essa suposição limite a aplicação da técnica, Hahs-Vaughn (135) sugere que para contornar este problema podemos: (i) adicionar um valor constante a cada uma das células da tabela ou (ii) realizar uma mesclagem de categorias adjacentes (135, pg. 137). Segundo Field, Miles e Field (131) o valor esperado é calculado pela equação:

$$Valor\ Esperado_{ij} = \frac{Total\ da\ Linha_i \times Total\ da\ Coluna_j}{Total\ Geral} \tag{1}$$

- **Não Separação de Dados**. Adicionalmente, segundo Hahs-Vaughn (135), uma separação completa ocorre quando a variável de resposta é completamente agrupada conforme uma variável preditora ou mesmo uma combinação de variáveis preditoras. O mesmo princípio se aplica ao *"quasi-complete separation"*, entretanto, neste caso variável de resposta é parcialmente agrupada (135, pg. 137) (131, pg. 323). No modelo de regressão logística, essas situações devem ser evitadas.

A regressão logística com uma variável independente é formulada pela Equação 2, onde X_1 representa a variável independente/preditora, β_0 é o coeficiente constante do modelo, também conhecido como *intercept*; β_1 é o coeficiente associado à variável independente e $0 \leq \pi \leq 1$ é o valor da curva de regressão logística (*conditional mean*) que representa a probabilidade de um evento que depende das variáveis independentes, ou seja, $\pi(x) = E(Y|x)$ lido como "o valor esperado de Y dado o valor de x" (136).

$$\pi(x) = \frac{e^{\beta_0+\beta_1 X_1}}{1 + e^{\beta_0+\beta_1 X_1}} \tag{2}$$

Para resolver essa equação, estima-se os valores dos coeficientes ($\beta_{0\ldots n}$) com base nos dados disponíveis das variáveis preditoras. Nos modelos de regressão logística geralmente se usa o princípio de máxima verossimilhança (*maximum-likelihood estimation*) para estimar os coeficientes (131). Geralmente, o estimador de máxima verossimilhança pode ser encontrado seguindo os passos: (i) encontrar a função de verossimilhança; (ii) aplicar a função *ln*; (iii) derivar em relação ao parâmetro; (iv) igualar o resultado a zero e (v) verificar que este estimador é ponto de máximo.

Na regressão logística, as equações resultantes do estimador de máxima verossimilhança são não-lineares em β_0 e β_1 e, portanto, requerem métodos especiais de solução. Esses métodos são de natureza iterativa (ex. Método de Newton-Raphson). Contudo, neste trabalho não iremos abordar e detalhar as técnicas usadas para resolver essas equações,

maiores esclarecimentos podem ser adquiridos nas obras de McCullagh e Nelder (137), Hosmer, Lemeshow e Sturdivant (136).

Uma vez estimados os coeficientes, é necessário verificar a significância do modelo. Uma das formas de realizar essa operação é aplicar o teste da Razão de Verossimilhança (RV) ou mesmo o teste de Wald (136). O método RV usa a distribuição chi-quadrado (χ^2) para comparar o valor da curva de regressão logística do modelo que inclui a variável independente (*New*) e o valor do modelo *Baseline* que apresenta apenas a constante (136, 131). Se o $p-$value deste teste for maior que o nível de significância testado, então, o modelo *Baseline* é descartado, indicando que as variáveis preditoras exercem alguma influência no modelo. Usando os mesmos princípios, também é necessário verificar se cada um dos coeficientes (β_0 e β_1) associados ao modelo de regressão são estaticamente diferentes de nulo (131).

Após a verificação da significância do modelo estatístico e de seus coeficientes, pode-se calcular o coeficiente de determinação (R^2) que é uma medida de ajuste do modelo em relação aos valores observados. Assim, é possível verificar o quanto o modelo consegue explicar a variabilidade dos dados (131).

Segundo Hosmer, Lemeshow e Sturdivant (136), os modelos de regressão logística quantificam a relação entre a variável dependente e as variáveis independentes através da taxa de probabilidade (*Odds Ratio — OR*). A taxa OR é obtida dividindo a probabilidade que um dado evento ocorrer pela probabilidade do evento não ocorrer, ou seja:

$$OR = \frac{P(Sucesso)}{P(Fracasso)} \tag{3}$$

Na regressão logística com apenas uma variável independente dicotômica, o valor OR desta variável está diretamente relacionado ao coeficiente β_1 da regressão (136) e pode ser diretamente obtido pela equação:

$$OR = e^{\beta_1} \tag{4}$$

Semanticamente, o valor OR é amplamente utilizado como uma medida de associação entre um tratamento (preditor/condição) e uma saída (resposta) (136). A métrica OR representa a probabilidade de que uma saída irá ocorrer em consequência do tratamento em comparação com a probabilidade da saída ocorrer sem o tratamento em questão. Para melhor entendimento considere o exemplo: se a saída denota a presença ou ausência do câncer de pulmão, o tratamento X (preditor) denota a característica do indivíduo de fumar (Fumante ou Não Fumante) e observamos um valor OR igual a 2 que é estatisticamente significante, então, considerando a amostra estudada, a interpretação deste valor é que as chances de ocorrer câncer de pulmão entre os indivíduos fumantes é duas vezes maior do que as chances de haver câncer de pulmão entre os indivíduos não fumantes (136). Segundo Field, Miles e Field (131) a métrica OR oferece três possíveis interpretações: (i) $OR > 1$ — indica que à medida que o preditor aumenta, também aumenta-se a

probabilidade de ocorrer a saída; (ii) $OR = 1$ — o preditor não afeta a probabilidade da saída; (iii) $OR < 1$ — à medida que o preditor aumenta, a probabilidade da saída ocorrer diminui.

Assim, o protocolo da execução desta técnica estatística pode ser brevemente resumida nos seguintes passos: (i) estabelecer o modelo com suas respectivas variáveis (independentes e dependentes). (i) verificar as precondições da regressão logística; (ii) estimar os coeficientes do modelo; (iii) verificar a significância do modelo em relação ao *Baseline* e a significância dos respectivos coeficientes; (iv) calcular o coeficiente de determinação (R^2); (v) quantificar a relação entre a variável dependente e as independentes; (vi) fornecer interpretação dos dados do modelo.

4.1.3.3 Equações das RQs Para o Modelo de Regressão Logística

Com base nos passos de execução da regressão logística (ver subseção anterior), para cada uma das RQs descritas na Subseção 4.1.2 definimos as variáveis independentes e dependentes de cada modelo (Equação 5).

A variável dependente (Y_{DC}) é do tipo binária onde cada observação denota a existência ou não existência de clone na entidade analisada. Como as RQs visam investigar a associação da prevalência de clones em relação a outros *smells*, a variável dependente é comum entre todas as perguntas de pesquisa. Entretanto, a variável independente e o conjunto de dados analisados variam conforme o objetivo específico de cada RQ.

Na RQ1.1 a variável independente é a intensidade do *smell* Complex Class (CC) que está categorizado em dois níveis (baixo[8] e alto[9]), por isso ela é representada como: $X_{CC_{1...2}}$. Essa pergunta de pesquisa analisa apenas as observações em que o *smell* CC ocorre isoladamente, ou seja, as observações de co-ocorrência entre LC e CC não são consideradas. A RQ1.2 é semelhante à RQ1.1, contudo, sua variável independente ($X_{LC_{1...2}}$) é a intensidade do *smell* Large Class (LC).

A RQ1.3 investiga a associação de clones entre as entidades acometidas pela co-ocorrência dos *smells* Large e Complex Class, conforme a intensidade de cada *smell*. Portanto, a variável independente é a intensidade em dois níveis destes *smells*. Como estes *smells* co-existem na mesma entidade, sua análise ocorre em dois modelos distintos ($Modelo_A/Modelo_B$): (i) considerando a intensidade do *smell* Large Class ($X_{CC_x \& LC_{1...2}}$) e (ii) tomando como base a intensidade do *smell* Complex Class ($X_{LC_x \& CC_{1...2}}$). Portanto, temos dois modelos que devem ser comparados a fim de identificar aquele que

[8] Indica que a métrica dessa classe é menor ou igual a mediana do *boxplot* — ver Subseção 4.1.3.1
[9] Indica que a métrica dessa classe é maior do que a mediana do *boxplot* — ver Subseção 4.1.3.1

melhor se adéqua aos dados.

$$
\begin{aligned}
\textbf{RQ1.1} &\rightarrow \left\{ Y_{DC} = \beta_0 + \beta_1 X_{CC_{1\ldots2}} \right. \\
\textbf{RQ1.2} &\rightarrow \left\{ Y_{DC} = \beta_0 + \beta_1 X_{LC_{1\ldots2}} \right. \\
\textbf{RQ1.3} &\rightarrow \begin{cases} Modelo_A \Rightarrow Y_{DC} = \beta_0 + \beta_1 X_{CC_x \& LC_{1\ldots2}} \\ Modelo_B \Rightarrow Y_{DC} = \beta_0 + \beta_1 X_{LC_x \& CC_{1\ldots2}} \end{cases} \\
\textbf{RQ2.1} &\rightarrow \left\{ Y_{DC} = \beta_0 + \beta_1 X_{CC_{Isolado\ldots Co-ocorrência}} \right. \\
\textbf{RQ2.2} &\rightarrow \left\{ Y_{DC} = \beta_0 + \beta_1 X_{LC_{Isolado\ldots Co-ocorrência}} \right.
\end{aligned}
\tag{5}
$$

No próximo grupo de perguntas a intensidade dos *smells* é desconsiderada na investigação de associação da prevalência de clones, possibilitando analisar a relação de clones entre entidades que são acometidas por um ou mais *smells*.

Neste sentido, a variável independente da RQ2.1 ($X_{CC_{Isolado\ldots Co-ocorrência}}$) é a categorização dos *smells* existentes nas entidades classificadas como Complex Class. Portanto, nosso conjunto de dados gera dois grupos: (i) aquele onde o *smell* Complex Class ocorre isoladamente e (ii) aquele em que o *smell* Complex Class co-ocorre com Large Class. Por fim, a variável independente da RQ2.2 ($X_{LC_{Isolado\ldots Co-ocorrência}}$) é análoga a esta da RQ2.1. Entretanto, o alvo da análise são as entidades identificadas com o *smell* Large Class.

4.2 Execução Experimental

Essa subseção descreve e caracteriza os projetos de software usados na parte experimental das questões de pesquisa. Para estes softwares, também apresentamos uma caracterização quantitativa dos *smells* relatando a distribuição de suas respectivas métricas. Na sequência, os resultados de cada pergunta de pesquisa são efetivamente descritos conforme o protocolo de execução da regressão logística, apresentado na seção anterior.

4.2.1 Caracterização dos Projetos de Software

Nesta subseção apresentamos, descrevemos e caracterizamos os projetos de software utilizados neste estudo empírico. Usamos cinco projetos de software, a maior parte (quatro) é de projetos que fazem parte do ecossistema gerenciado pelo grupo *Apache Software Foundation* (ASF[10]) que supervisiona o desenvolvimento de mais de 350 projetos de software *open source* e conta com a colaboração de desenvolvedores de grandes empresas (ex.: Google, Microsoft, IBM). O grupo ASF é conhecido, dentre outras coisas, por aplicar

[10] <https://projects.apache.org/>

princípios da engenharia de software (ex.: Modularização). Isso foi um dos fatores que motivaram a análise destes projetos. Também analisamos o código fonte de um projeto que não faz parte do ecossistema gerido pela ASF. Este é um projeto recorrente na literatura de *bad smells*, por ser sabidamente reconhecido por apresentar diversos *smells*. Essas características motivaram a inclusão deste projeto, possibilitando aumentar a diversidade dos sistemas. Nos próximos parágrafos, descrevemos a finalidade de cada um destes projetos.

ArgoUML[11] é uma ferramenta *open source* usada para modelar diagramas UML, compatíveis com os padrões definidos na versão 1.4 destes diagramas. Essa ferramenta permite a criação de nove diagramas: diagrama de classe, diagrama de estado, diagrama de atividade, utilizar diagrama de caso, diagrama de interação, diagrama de distribuição e diagrama de sequências.

Apache™ Cassandra®[12] é um banco de dados distribuído usado para gerenciar grandes volumes de dados estruturados e alocados em múltiplos servidores. Este banco de dados é baseado na tecnologia NoSQL.

Apache™ Lucene®[13] é uma biblioteca de alta performance que possibilita a indexação de documentos e pesquisa de texto nos arquivos indexados. Sua implementação é altamente confiável, escalável e tolerante a falhas, fornecendo indexação distribuída, replicação e consulta com balanceamento de carga. Essa biblioteca é usada em mecanismos de busca e navegação de grandes sites (ex. DuckDuckGo[14], Netflix[15]).

Apache™ Hadoop®[16] é um *framework* de software de código aberto usado para processar grandes conjuntos de dados. Este *framework* possibilita o processamento distribuído dos conjuntos de dados em *clusters* de computadores usando modelos de programação simplificados. Este *framework* é formado por diversos módulos (ex. HDFS, YARN, MapReduce), um deles, o Hadoop Common é um utilitário comum que suporta os outros módulos do Hadoop e o código deste módulo será analisado em nosso estudo.

Apache™ Ant®[17] é um utilitário especialmente desenvolvido para construir/compilar ferramentas e bibliotecas desenvolvidas na linguagem Java.

A Tabela 6 apresenta, em termos numéricos, a caracterização dos projetos de software. Essa tabela apresenta as seguintes informações: (i) versão do sistema; (ii) linguagem usada no desenvolvimento, (iii) quantidade de arquivos Java que compõe o projeto; (iv) quantidade de linhas em branco que os arquivos Java apresentam; (v) número de linhas comentadas e (vi) quantidade de linhas de código que tem a habilidade de serem executadas. As versões listadas nessa tabela foram selecionadas com base no aspecto de ser a

[11] <http://argouml.tigris.org/>
[12] <http://cassandra.apache.org/>
[13] <https://lucene.apache.org/>
[14] <https://duckduckgo.com/>
[15] <https://www.netflix.com/>
[16] <http://hadoop.apache.org/>
[17] <http://ant.apache.org/>

Tabela 6 – Caracterização dos projetos analisados

Projeto	**Versão**	**Linguagem**	**Número de**			
			Arquivos	**Linhas em Branco**	**Comentários**	**Linhas de Código**
ArgoUML	0.34	Java	1233	25581	100193	105795
Cassandra	3.11	Java	2062	64190	081278	333211
Lucene	6.2.1	Java	3848	90570	183106	533926
Hadoop	2.6.0	Java	1417	32985	077373	194518
Ant	1.8.2	Java	1182	26800	099397	127042

Métricas obtidas pela ferramenta Cloc <http://cloc.sourceforge.net/>

versão mais recente do código fonte que conseguimos configurar o *.classpath* e usá-lo para compilar o projeto diretamente na IDE (*Integrated Development Environment*) Eclipse. Pela tabela, observa-se que o Lucene é o maior sistema em todos os quesitos.

4.2.1.1 Caracterização dos Dados de *Smells*

Nessa subseção apresentamos, por meio do *boxplot* de cada projeto de software, a distribuição de cada uma das métricas usadas para detectar os *smells* Duplicate Code, Large Class e Complex Class.

Duplicate Code. Conforme apresentado, este *smell* é detectado usando a métrica *tokens* (ver Subseção 2.3.1) e ele pode ocorrer em qualquer arquivo Java, independente se este apresenta ou não outro(s) *smell(s)*. Isso quer dizer que uma classe que não apresenta qualquer um dos outros *smells* (LC/CC) pode apresentar o *smell* Duplicate Code. Por outro lado, uma classe com algum dos *smells* de interesse pode apresentar n instâncias de Duplicate Code. Assim, a caracterização deste *smell* foi dividida em dois aspectos: (i) **análise do projeto**: considera todas as classes do projeto, independente da ocorrência dos *smells* de interesse (ver Figura 13) e (ii) **análise de interesse**: considera apenas os clones que estão nas classes de interesse (ver Figura 14), ou seja, classes que apresentam pelo menos um dos *smells*: Large Class, Complex Class. Complementarmente, cada um destes aspectos subdividem-se em duas perspectivas: (a) considerando a métrica de número de *tokens* e (b) considerando o número de linhas clonadas. Esse detalhamento em outras duas perspectivas se faz necessário pelo fato da métrica *tokens* ser pouco intuitiva, então uma correspondência em número de linhas de código clonadas favorece a compreensão dos dados.

Duplicate Code: aspecto análise do projeto. No geral, em termos de *outliers*, a perspectiva baseada na métrica *tokens* (Figura 13a) apresenta maior prevalência do que a outra perspectiva baseada em número de linhas clonadas (Figura 13b). Isso é especialmente significativo quando consideramos apenas os dados que estão acima do quartil superior. Esta diferença pode ser explicada pelo fato da métrica *tokens* considerar apenas

os comandos/sintaxe da linguagem e desconsiderar questões de estilo de programação. Na prática, isso quer dizer que dois códigos com a mesma quantidade de linhas certamente contém números diferentes de *tokens* pois essa métrica tem uma granularidade mais fina do que o número de linhas. Por consequência, isso aumenta a variabilidade da métrica *tokens* que gera maior incidência de *outliers*. Considerando os cinco projetos de software, a mediana da métrica *tokens* ficou entre 105 e 97 (Figura 13a). O Lucene foi o projeto de software que apresentou a maior prevalência de clones, 2255 fragmentos de trechos de códigos clonados. Por outro lado, o ArgoUML apresentou apenas 214 trechos de clones. Dado que estes sistemas são, receptivamente, os maiores e menores projetos, isso era esperado. Da perspectiva do número de linhas clonadas, a mediana ficou entre 24 e 16 (Figura 13b). Observa-se que o ArgoUML foi o projeto que apresentou os maiores fragmentos clonados, esperava-se isso pois este projeto é reconhecido pelas deficiências dos princípios de engenharia de software.

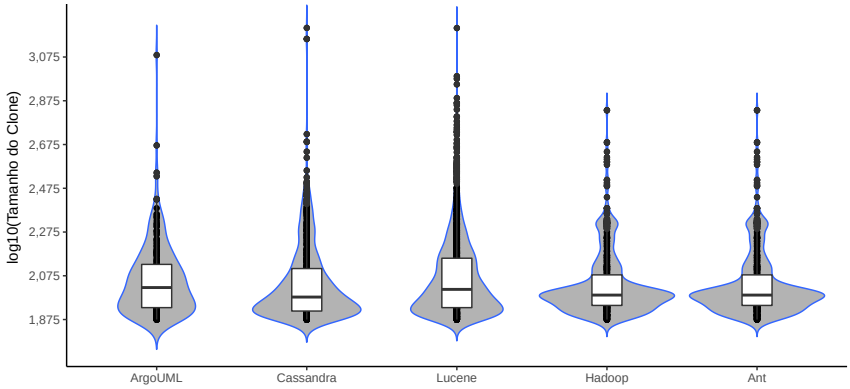
Duplicate Code: aspecto análise de interesse. Ao considerar os clones apenas nas classes de interesse (Figura 14) e compararmos com os dados anteriores (Figura 13), observamos que a mediana da métrica aumenta expressivamente. A mediana na Figura 14a está entre 448 e 183, enquanto que na Figura 14b está entre 124 e 71. Isso quer dizer que em termos da métrica *tokens*, em média, a mediana da Figura 14a é 2,91 vezes maior do que na Figura 13a. Por outro lado, em termos de números de linhas clonadas, em média, a mediana da Figura 14b é 4,02 vezes maior do que na Figura 13b. Isso quer dizer que os maiores fragmentos de clones estão mais concentrados nas classes de interesse, ou seja, aquelas que são Large Class e/ou Complex Class. Esse comportamento era esperado, pois conforme apresentado, Fowler e Beck (15) e outros trabalhos conjecturaram analiticamente a possível relação entre clones e *smells* relacionados à estruturas grandes e/ou complexas. Também observamos que o Lucene ainda é o projeto de software com a maior quantidade de fragmentos de clones. Diferentemente do que aconteceu na Figura 13b, a distribuição dos clones do projeto ArgoUML não é tão diferente dos demais projetos (Figura 14b). Isso quer dizer que independente do nível de interesse pelos princípios de engenharia de software demonstrado nos projetos, os clones nas classes de interesse seguem uma distribuição relativamente semelhante.

Large Class (LC). Este *smell* é detectado pelo DECOR usando uma combinação de métricas, em especial, o número de métodos (NMD) e atributos (NAD). Essas métricas são usadas pelo DECOR pois este *smell* está relacionado à quantidade de responsabilidades atribuídas à classe, sendo que quanto maior o valor dessas métricas, maior é a quantidade de atribuições da classe. A Figura 15 apresenta a distribuição dessas métricas apenas nas classes que o DECOR considera apresentar o *smell* Large Class. Observe que neste caso, essas classes não apresentam a co-ocorrência dos *smells* Large Class e Complex Class (No Apêndice, a Figura 35 considera todas as classes com LC, independente da co-ocorrência com outros *smells*). Interessante notar que o projeto ArgoUML apresenta

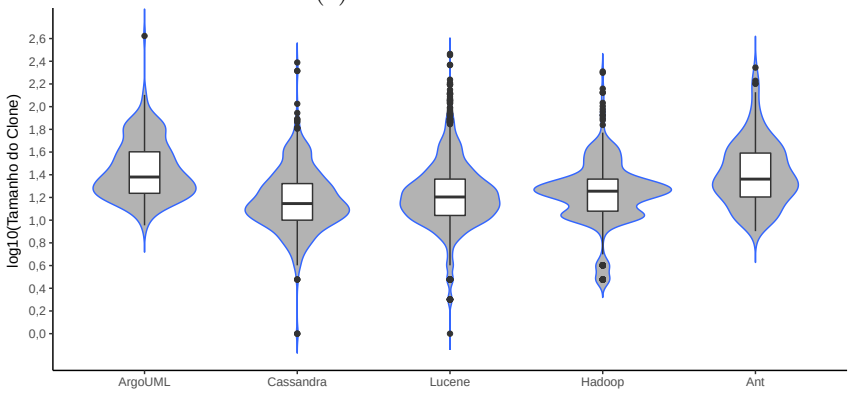
valores menores nos quartils do que os outros projetos que participam do ecossistema Apache. Possivelmente, isso indica que estas classes do ArgoUML apresentam menor número de responsabilidades do que as classes dos outros projetos. Considerando que em termos de número de arquivos Java e número de linhas de código (ver Tabela 6) o projeto ArgoUML e o Ant são semelhantes, acreditamos que essa diferença nos quartils pode ser resultado de decisões de projeto ou mesmo devido às características inerentes ao domínio das aplicações (Ant — utilitário de compilação, ArgoUML — ferramenta de modelagem). Em geral e em termos numéricos, a mediana dos projetos ficou entre 21 e 36.

Complex Class (CC). Este *smell* é caracterizado no DECOR pelo valor da métrica *McCabe* que na sua essência quantifica o número de estruturas de desvio de fluxo. A Figura 16 denota a distribuição dessa métrica nas classes consideradas CC, observe que essa figura não considera as classes em que este *smell* co-ocorre com LC, isso é apresentado na Figura 36 do Apêndice. Nota-se que o projeto Lucene foi o que apresentou os maiores valores, com picos que atingem o valor de 1262. Em geral, a mediana desta métrica ficou entre 38 e 52. Complementarmente, os projetos Hadoop, Ant e Cassandra são os que apresentaram as menores variabilidades desta métrica. Por outro lado, os projetos Lucene e ArgoUML são aqueles com os maiores *outliers*.

Co-ocorrência de *Smells* (LC/CC). A Figura 17 é análoga às anteriores, contudo, ela apresenta a distribuição das métricas *McCabe* e *NMD+NAD* para as classes em que os *smells* Large e Complex Class co-ocorrem na mesma entidade. Em relação à Figura 16, a métrica *McCabe* da Figura 17 apresenta um aumento nos valores da mediana. Em termos percentuais, este aumento ocorre em taxas que variam de 50% a 65%. Por outro lado e em relação à Figura 15, a métrica *NMD+NAD* da Figura 17 também apresenta um aumento nos valores da mediana. Contudo, os percentuais de aumento estão entre 38% e 52%. Este fato possivelmente indica que a co-ocorrência destes *smells*, gera uma interação entre eles e isso é refletido pelo aumento no valor das métricas. Pela Figura 17 também se observa que o projeto ArgoUML apresenta os menores valores da mediana para ambas as métricas, isso é consistente com o comportamento apresentado nas Figuras 15 e 16. Isso, possivelmente denota que as características dos sistemas (ex. decisões de projeto, domínio da aplicação) são razoavelmente semelhantes nas classes analisadas e, em certo grau, independe a existência de determinados *smells*.

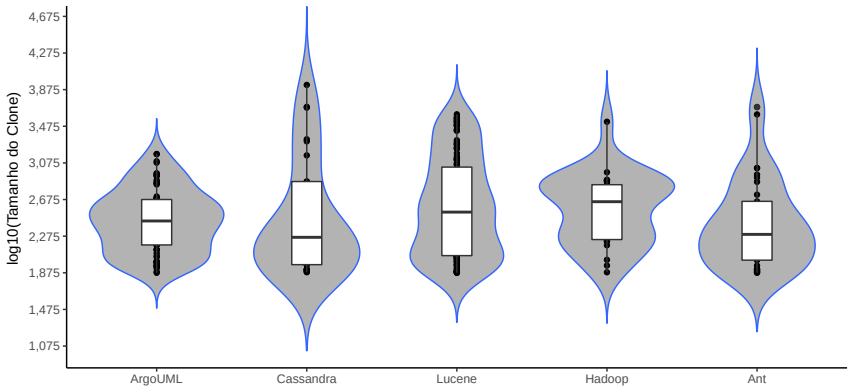


(a) Métrica: *Tokens*

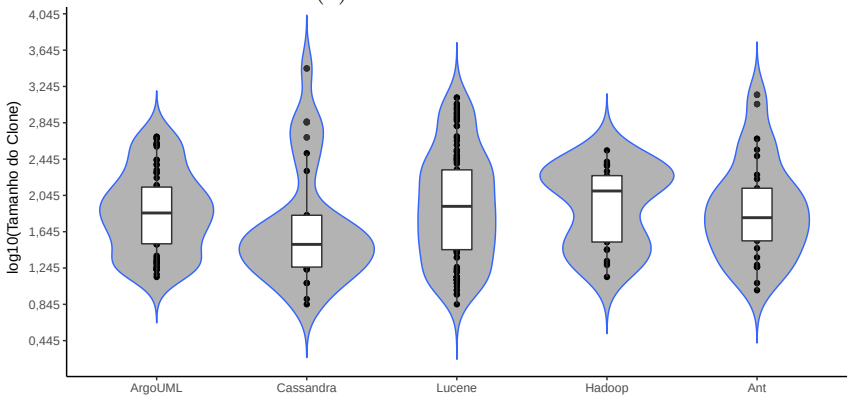


(b) Métrica: Número de linhas

Figura 13 – Boxplot do tamanho dos clones em todas as classes do sistema.



(a) Métrica: *Tokens*



(b) Métrica: Número de linhas

Figura 14 – Boxplot do tamanho dos clones apenas nas classes LC e/ou CC.

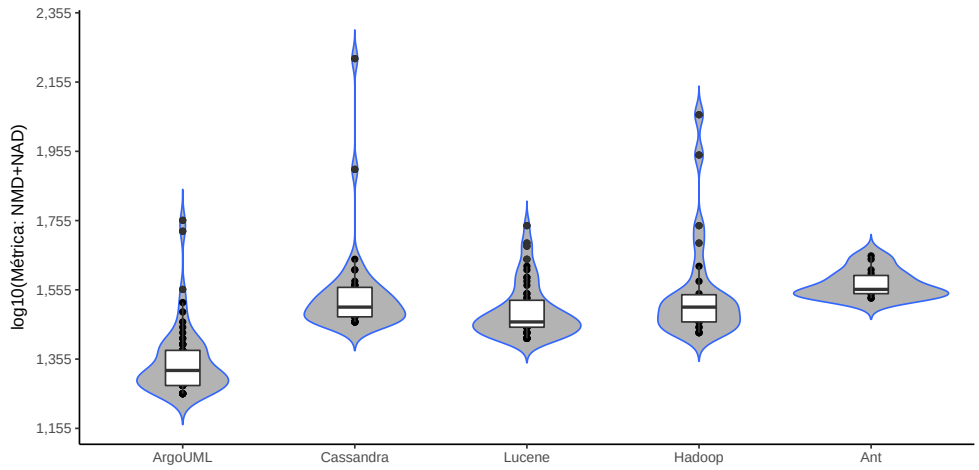


Figura 15 – Boxplot da métrica $NMD + NAD$ nas classes com o *smell* LC.

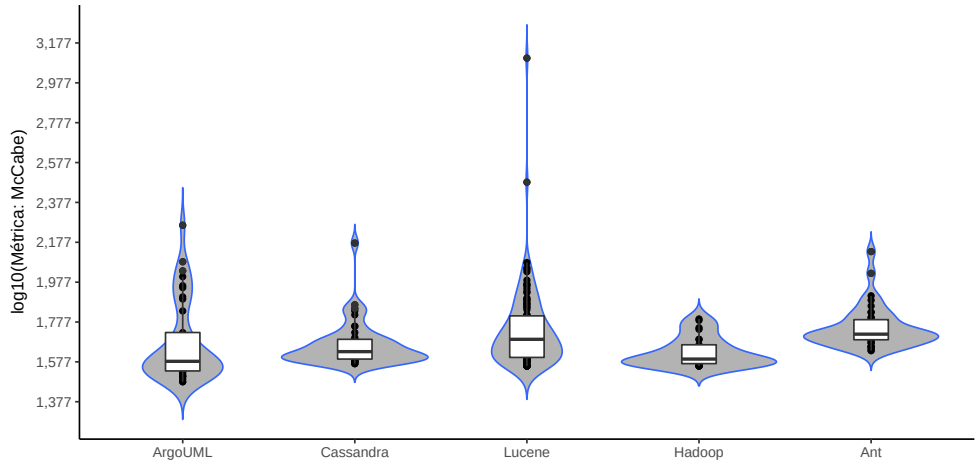


Figura 16 – Boxplot da métrica $McCabe$ nas classes com o *smell* CC.

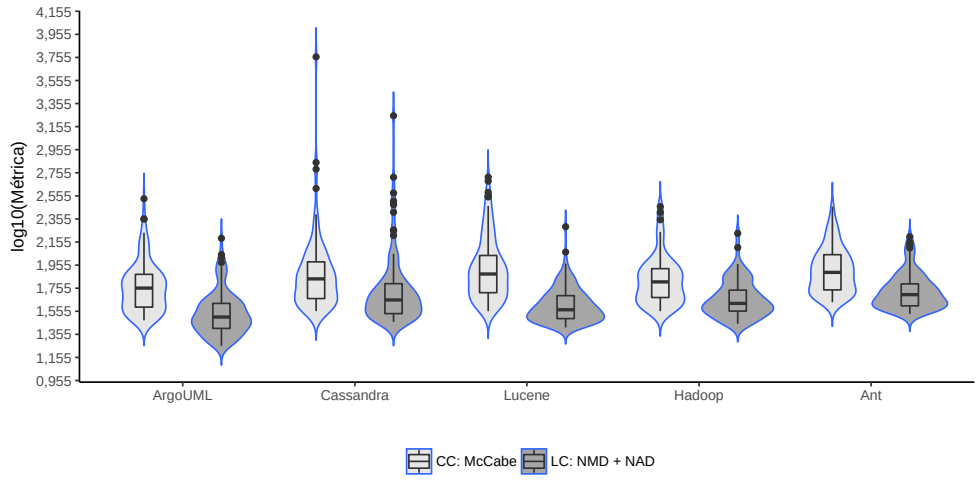


Figura 17 – Boxplot das métricas de LC e CC nas classes de co-ocorrência.

4.2.2 Resultados Experimentais

Essa subseção apresenta os resultados estatísticos para cada questão de pesquisa apresentada na subseção 4.1.2. A subseção está subdividida em duas partes, a primeira apresenta os resultados da relação entre prevalência de clones e a intensidade dos *smells* de interesse e a segunda destina-se a detalhar a relação entre prevalência de clones e a prevalência de tipos de *smells*, sem considerar a intensidade.

4.2.2.1 Associação da Intensidade de *Smells* a Clones

Os resultados das três questões de pesquisa relacionadas à intensidade de *smells* são apresentados nesta subseção.

RQ1.1 — A intensidade do *smell* Complex Class está associada à prevalência de *clones*? Se sim, quanto ela está associada?

A Tabela 7 apresenta os dados de cada projeto de software organizados em uma tabela de contingência. Dentre os *smell* que estudamos, essa tabela detalha como as classes afetadas apenas pelo *smell* Complex Class estão quantitativamente agrupadas em relação à existência de clones. Observe que a complexidade das classes está dividida em dois níveis. No primeiro nível (N_1) estão as classes em que a métrica *McCabe* se apresenta em valores que estão abaixo da mediana. Por outro lado, o segundo nível apresenta as classes com a métrica *McCabe* acima da mediana. Observe que para os estudos de casos apresentados neste capítulo, a mediana usada para desmembrar os níveis de intensidade é obtida usando os dados das classes que são acometidas pelo *smell* de interesse, independente da co-ocorrência com outros *smells*, exemplo: o projeto ArgoUML apresenta 156 classes CC, sendo 37 ocorrendo isoladamente (Tabela 7) e 119 ocorrendo concomitantemente com o *smell* Large Class (Tabela 12), neste caso, a mediana dessa métrica foi de 57 (ver Figura 36).

A tabela de contingência ajuda a compreender a relação entre duas variáveis categóricas (131) e ainda demonstra se os dados satisfazem uma das premissas necessárias à estatística de regressão logística (ver Subseção 4.1.3.2). Pela Tabela 7 observamos que a maior parte das classes com clones apresenta complexidades que estão agrupadas no primeiro nível. Complementarmente, observamos que a maior parte das classes complexas do primeiro nível não apresentam fragmentos clonados. Isso induz a conclusão de que não há associação entre a intensidade do *smell* Complex Class e a prevalência de clones. Contudo, é necessário executar métodos estatísticos que consolidem esse pensamento. Neste caso, usamos a regressão logística.

A execução da regressão logística necessita que as células da tabela de contingência não apresentam valores zerados e que o valor esperado das células sejam maiores que dois. Para os dados desta questão de pesquisa, observa-se que os projetos Hadoop e Ant são

os únicos que não satisfazem as condições necessárias do método estatístico em questão. Portanto, os dados da regressão logística destes projetos devem ser desconsiderados.

Tabela 7 – Tabela de Contingência — RQ1.1: Variando a intensidade do *smell* CC

		ArgoUML			**Cassandra**			**Lucene**			**Hadoop**			**Ant**		
		Clone		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**
		Não	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**	
CC	**N.$_1$**	18	8	**26**	30	6	**36**	43	27	**70**	18	3	**21**	22	4	**26**
	N.$_2$	4	7	**11**	6	2	**8**	16	14	**30**	4	0	**4**	5	2	**7**
Total		**22**	**15**	**37**	**36**	**8**	**44**	**59**	**41**	**100**	**22**	**3**	**25**	**27**	**6**	**33**

A Tabela 8 detalha os dados da regressão logística para cada um dos projetos analisados. Essa tabela apresenta os parâmetros de cada modelo, detalhando a constante (*Intercept*) e o coeficiente β_1 que representa a intensidade do *smell* CC (variável preditora). Para cada um destes parâmetros, também apresentamos os valores estatísticos do erro padrão (SE) e a significância estatística destes termos. O coeficiente de determinação (R^2) que é uma medida de ajuste do modelo em relação aos valores observados, também é apresentado nessa tabela. O R^2 indica o quanto o modelo consegue explicar os valores observados. Complementarmente apresentamos o valor AIC (*Akaike Information Criterion*), que é usado para comparar o ajuste entre modelos, neste caso, quanto menor o valor melhor é o ajuste do modelo. Por fim, essa tabela também detalha o intervalo de confiança (IC), *Odds Ratio* (OR) e a estatística do teste χ^2 usada para avaliar a significância estatística do modelo com a variável preditora e do modelo que considera apenas o *Intercept* (*baseline*).

Os dados da Tabela 8 demonstram que dos cinco projetos analisados, apenas no ArgoUML a inclusão da variável preditora intensidade do *smell* Complex Class é estatisticamente significante em relação ao modelo alternativo, ou seja, o modelo que a considera é melhor do que o modelo com apenas o *Intercept*. Complementarmente, para este projeto, a intensidade da variável preditora está significativamente associada com a chance de ocorrer clones (p-value é menor que o nível de significância de 10%). Em termos da métrica OR, a interpretação deste valor é que as chances de ocorrer clones entre as entidades mais complexas é 3,93 vezes maior do que as chances de haver clones entre as entidades menos complexas. Contudo, analisando o R^2, observamos que apenas 6,9% da variabilidade dos dados pode ser explicada pelo modelo. Na prática, isso quer dizer que além da intensidade deste *smell* há outros fatores que ajudam a explicar a prevalência de clones e provavelmente algum destes fatores deve explicar melhor a prevalência de clones.

Tabela 8 – Modelos de Regressão — RQ1.1: Variando a intensidade do *smell* CC

Projeto	**Constante (SE)**	β_1 **(SE)**	**AIC**	$\mathbf{R^2}$	**IC para** β_1			***Baseline vs. New***
					Inferior (5%)	**OR**	**Superior (95%)**	
ArgoUML	-0,810$^{\triangleright}$ (0,42)	1,370$^{\triangleright}$ (0,75)	50,51	0,069	1,166	3,937	14,512	$\chi^2(1)$=3,44, p=0,064$^{\otimes}$
Cassandra	-1,609$^{\triangledown}$ (0,44)	0,510$^{\blacklozenge}$ (0,93)	45,44	0,007	0,307	1,666	7,251	$\chi^2(1)$=0,29, p=0,592
Lucene	-0,465$^{\triangle}$ (0,24)	0,331$^{\blacklozenge}$ (0,44)	138,81	0,004	0,672	1,393	2,882	$\chi^2(1)$=0,57, p=0,452
Hadoop	-1,791$^{\triangleleft}$ (0,62)	-16,774$^{\blacklozenge}$ (3261,3)	21,22	0,061	—	$5,19E-08$	$6,19E+139$	$\chi^2(1)$=1,12, p=0,290
Ant	-1,704$^{\triangleleft}$ (0,54)	0,788$^{\blacklozenge}$ (0,99)	34,70	0,019	0,375	2,2	11,079	$\chi^2(1)$=0,59, p=0,441

R^2=Hosmer–Lemeshow (131)
Significância dos Coeficientes (R®): 0,001$^{\triangledown}$; 0,01$^{\triangleleft}$; 0,05$^{\triangleright}$; 0,1$^{\triangle}$; 1$^{\blacklozenge}$
$^{\otimes}$Para $\alpha = 0,10$, o modelo com a variável β_1 é significativamente melhor do que aquele com apenas a constante ($p < \alpha$).

Analiticamente, considerando o baixo valor do R^2 obtido nos dados do projeto ArgoUML e ainda que em 80% dos projetos analisados a variável preditora não foi melhor do que o modelo *baseline*, podemos presumir que a prevalência de clones está fracamente e esporadicamente associada à intensidade do *smell* Complex Class, quando este ocorre isoladamente.

RQ1.2 — A intensidade do *smell* Large Class está associada à prevalência de *clones*? Se sim, quanto ela está associada?

A Tabela 9 descreve, para cada projeto de software, os dados das entidades com o *smell* Large Class (LC) na forma de uma tabela de contingência. Para essa questão de pesquisa, investigamos a relação entre a prevalência de clones e a intensidade do *smell* LC nas entidades que apresentam apenas o *smell* LC, o qual a intensidade foi agrupada em dois níveis. O desmembramento destes níveis ocorreu de forma análoga ao realizado na subseção anterior (RQ1.1), entretanto, a mediana foi obtida pela métrica $NMD + NAD$ de todas as classes que apresentam o *smell* Large Class (ver Figura 35).

Tabela 9 – Tabela de Contingência — RQ1.2: Variando a intensidade do *smell* LC

		ArgoUML			**Cassandra**			**Lucene**			**Hadoop**			**Ant**		
		Clone		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**
		Não	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**	
LC	**N.$_1$**	33	22	**55**	30	3	**33**	26	16	**42**	15	5	**20**	5	5	**10**
	N.$_2$	6	2	**8**	3	2	**5**	7	5	**12**	6	0	**6**	0	0	**0**
Total		**39**	**24**	**63**	**33**	**5**	**38**	**33**	**21**	**54**	**21**	**5**	**26**	**5**	**5**	**10**

Análogo ao observado na RQ1.1, pela Tabela 9 observamos que a maior parte das classes com clones estão agrupadas no primeiro nível de tamanho do *smell* LC. Complementarmente, observamos que a maior parte das classes grandes no primeiro nível de tamanho não apresentam fragmentos de clones. Para averiguar a associação dos fatores, é necessário a execução da regressão logística (ver Tabela 10). Contudo, essa técnica necessita que as células da tabela de contingência não apresentam valores zerados e que o valor esperado das células deve ser maior que dois. Assim, para os dados desta questão de pesquisa, observa-se que os projetos Cassandra, Hadoop e Ant não satisfazem as condições necessárias da regressão logística. Portanto, os dados da regressão logística destes projetos devem ser desconsiderados (ver Tabela 10).

Tabela 10 – Modelos de Regressão — RQ1.2: Variando a intensidade do *smell* LC

Projeto	**Constante (SE)**	β_1 **(SE)**	**AIC**	$\mathbf{R^2}$	**IC para β_1**			***Baseline vs. New***
					Inferior (5%)	**OR**	**Superior (95%)**	
ArgoUML	-0,405$^{\blacklozenge}$ (0,27)	-0,693$^{\blacklozenge}$ (0,86)	87,03	0,008	0,099	0,5	1,883	$\chi^2(1)$=0,70, p=0,402
Cassandra	-2,302$^{\triangledown}$ (0,60)	1,897$^{\triangle}$ (1,09)	30,83	0,093	1,019	6,666	42,026	$\chi^2(1)$=2,76, p=0,097$^{\otimes}$
Lucene	-0,485$^{\blacklozenge}$ (0,31)	0,149$^{\blacklozenge}$ (0,66)	76,12	0,001	0,375	1,16	3,456	$\chi^2(1)$=0,05, p=0,823
Hadoop	-1,098$^{\triangleright}$ (0,51)	-17,467$^{\blacklozenge}$ (2662,85)	26,49	0,116	—	$2,59E-08$	$3,85E+87$	$\chi^2(1)$=2,96, p=0,085$^{\otimes}$
Ant	-1,550$^{\triangledown}$ (0,41)	0,545$^{\blacklozenge}$ (0,51)	106,18	0,011	0,755	1,724	4,158	$\chi^2(1)$=1,16, p=0,281

R^2=Hosmer–Lemeshow (131)
Significância dos Coeficientes (R$^{®}$): 0,001$^{\triangledown}$; 0,01$^{\triangleleft}$; 0,05$^{\triangleright}$; 0,1$^{\triangle}$; 1$^{\blacklozenge}$
$^{\otimes}$Para $\alpha = 0,10$, o modelo com a variável β_1 é significativamente melhor do que aquele com apenas a constante ($p < \alpha$).

A Tabela 10 demonstra que os projetos Cassandra e Hadoop são os únicos em que a inclusão da variável preditora intensidade do *smell* Large Class é estatisticamente significante em relação ao modelo alternativo *baseline*. Contudo, como apresentado, estes projetos não satisfazem certas condições da regressão logística. Portanto, não podem ser considerados na análise. Por outro lado, para os outros três projetos (ArgoUML, Lucene e Ant), o modelo na ausência da variável preditora (*baseline*) foi considerado melhor, indicando que essa variável não é relevante ao modelo. Assim, para os cinco projetos analisados, a variável preditora **não** é estatisticamente associada à prevalência de clones. Na prática, isso quer dizer que há outros fatores não relacionados à métrica em questão que expliquem a prevalência de clones nas entidades LC.

Considerando os dados estatísticos apresentados, para os cinco projetos analisados, a variável preditora intensidade do *smell* Large Class ocorrendo isoladamente **não** está estatisticamente associada à prevalência de clones em nenhum dos projetos de software que analisamos.

RQ1.3 — A intensidade da co-ocorrência, na mesma entidade (classe), dos *smells* Complex Class e Large Class está associada à prevalência de *clones*? Se sim, quanto ela está associada?

Essa questão de pesquisa é análoga às anteriores, ou seja, investiga a relação entre a intensidade dos *smells* e a prevalência de clones. Contudo, as entidades analisadas são aquelas onde os *smells* Large e Complex Class co-existem na mesma entidade (ver Figura 17). Isso quer dizer que o agrupamento dos dados em dois níveis de intensidade pode ocorrer de duas formas: (i) considerando a intensidade do *smell* Complex Class (Tabela 11) ou (ii) tomando como base a intensidade do *smell* Large Class (Tabela 12). Portanto, para cada projeto de software, temos dois modelos (ver Tabela 13) que devem ser comparados a fim de identificar aquele mais adequado aos dados.

Tabela 11 – Tabela de Contingência — RQ1.3: Variando a intensidade do CC* na co-ocorrência

	ArgoUML			**Cassandra**			**Lucene**			**Hadoop**			**Ant**		
	Clone		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**
	Não	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**	
CC_1&LC_x	39	12	**51**	39	2	**41**	45	12	**57**	21	4	**25**	26	7	**33**
CC_2&LC_x	40	28	**68**	53	16	**69**	40	63	**103**	30	13	**43**	48	15	**63**
Total	**79**	**40**	**119**	**92**	**18**	**110**	**85**	**75**	**160**	**51**	**17**	**68**	**74**	**22**	**96**

* Referência ao modelo * da Tabela 13

A Tabela 11 denota a tabela de contingência tendo como base a intensidade da métrica *McCabe* agrupada em dois níveis. Essa tabela e sua forma de estruturar os dados são análogas ao procedimento detalhado na subseção 4.2.2.1 que descreve os resultados da RQ1.1. Contudo, diferentemente do que foi demonstrado na Tabela 7 da RQ1.1, observa-se que a maior parte das entidades com clones também são aquelas que apresentam complexidade no maior nível (Tabela 11). Os dados de todos os cinco projetos de software atendem à necessidade do método estatístico de não apresentar valores zerados nas células da tabela de contingência e que os valores esperados são maiores que dois. Isso quer dizer que todos os modelos representados por "*" que estão descritos na Tabela 13 devem ser analisados.

A Tabela 12 descreve os mesmos dados apresentados na Tabela 11, contudo, neste caso, a representação dos dados na forma de tabela de contingência usa a métrica $NMD+NAD$ para agrupar as classes em dois níveis de intensidade. A forma de estruturar os dados nessa tabela é análoga ao procedimento detalhado na subseção 4.2.2.1, onde os resultados da RQ1.2 são descritos. Similarmente ao demonstrado na Tabela 11, a atual tabela de contingência também demonstra que todos os cinco projetos de software atendem à necessidade do método estatístico de não apresentarem valores zerados nas células da

tabela de contingência e que o valor esperado das células é maior que dois. Portanto, todos os modelos representados por "$^{\oslash}$" que estão descritos na Tabela 13 também devem ser analisados.

Tabela 12 – Tabela de Contingência — RQ1.3: Variando a intensidade do LC$^{\oslash}$ na co-ocorrência

	ArgoUML			**Cassandra**			**Lucene**			**Hadoop**			**Ant**		
	Clone		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**
	Não	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**	
CC_x&LC_1	20	14	**34**	36	4	**40**	40	24	**64**	20	4	**24**	33	7	**40**
CC_x&LC_2	59	26	**85**	56	14	**70**	45	51	**96**	31	13	**44**	41	15	**56**
Total	**79**	**40**	**119**	**92**	**18**	**110**	**85**	**75**	**160**	**51**	**17**	**68**	**74**	**22**	**96**

$^{\oslash}$ Referência ao modelo $^{\oslash}$ da Tabela 13

A Tabela 13 descreve dois modelos estatísticos para cada um dos cinco projetos de software analisados. Isso porque nas entidades analisadas temos a co-ocorrência de dois *smells*, portanto, os dados podem ser analisados sobre duas perspectivas de *smells*. Assim, os modelos representados pelo símbolo "*" foram gerados usando a intensidade do *smell* Complex Class (Tabela 11). Por outro lado, os modelos representados pelo símbolo "$^{\oslash}$" foram gerados usando a intensidade do *smell* Large Class (Tabela 12). Ao prosseguir com a análise, devemos decidir qual destes modelos é mais adequado aos dados. Assim, devemos: (i) verificar se inclusão da variável preditora intensidade do *smell* é estatisticamente relevante em relação ao modelo *baseline* e (ii) caso seja relevante nos dois modelos ("$^{\star}/^{\oslash}$"), usar a métrica AIC para comparar os modelos.

Analisando os dados da Tabela 13 observamos que em 80% dos modelos "$^{\oslash}$" a variável preditora intensidade do *smell* Large Class **não** é estatisticamente relevante em relação ao modelo *baseline*. Isso quer dizer que essa variável preditora é relevante em apenas um modelo, aquele do projeto Lucene. Por outro lado, apenas 40% dos modelos "*" a variável preditora intensidade do *smell* Complex Class **não** é estatisticamente relevante em relação ao modelo *baseline*. Assim, temos três modelos em que essa variável preditora é relevante e estes são os modelos dos projetos ArgoUML, Cassandra e Lucene. Observe que houve apenas um caso (projeto Lucene), onde ambos modelos ("$^{\star}/^{\oslash}$") apresentam variáveis preditoras como sendo estatisticamente relevante em relação ao modelo *baseline*. Contudo, ao analisar a métrica AIC, observamos que o modelo "*" é o que melhor se adere aos dados, pois este tem o menor AIC (200,28). Portanto, para 60% dos casos analisados, a variável preditora intensidade do *smell* Complex Class foi relevante ao modelo e **não** houve casos onde a intensidade do *smell* Large Class foi considerada relevante ao modelo. Este resultado é consistente com os resultados das questões de pesquisa RQ1.1 e RQ1.2, discutidos na subseção 4.2.2.1. Indicando que a complexidade da entidade está mais associada há prevalência de clones do que ao tamanho da entidade.

Tabela 13 – Modelos de Regressão — RQ1.3: Variando a intensidade na co-ocorrência

Projeto		**Constante (SE)**	β_1 **(SE)**	**AIC**	$\mathbf{R^2}$	**IC para β_1**			***Baseline vs. New***
						Inferior (5%)	**OR**	**Superior (95%)**	
ArgoUML	⋆	-1,178$^{\triangledown}$ (0,33)	0,822$^{\triangleright}$ (0,41)	151,79	0,027	1,169	2,275	4,561	$\chi^2(1)$=4,16, p=0,041$^{\otimes}$
	⊘	-0,356$^{\blacklozenge}$ (0,35)	-0,462$^{\blacklozenge}$ (0,42)	154,75	0,008	0,315	0,629	1,265	$\chi^2(1)$=1,20, p=0,274
Cassandra	⋆	-2,970$^{\triangledown}$ (0,72)	1,772$^{\triangleright}$ (0,77)	94,715	0,075	1,886	5,886	26,881	$\chi^2(1)$=7,33, p=0,007$^{\otimes}$
	⊘	-2,197$^{\triangledown}$ (0,52)	0,810$^{\blacklozenge}$ (0,60)	100,06	0,02	0,877	2,25	6,663	$\chi^2(1)$=1,98, p=0,159
Lucene	⋆	-1,321$^{\triangledown}$ (0,32)	1,776$^{\triangledown}$ (0,38)	200,28	0,113	3,206	5,906	11,35	$\chi^2(1)$=24,90, p=0,000$^{\otimes}$
	⊘	-0,510$^{\triangleright}$ (0,25)	0,636$^{\triangle}$ (0,32)	221,39	0,017	1,103	1,888	3,267	$\chi^2(1)$=3,79, p=0,051$^{\otimes}$
Hadoop	⋆	-1,658$^{\triangleleft}$ (0,54)	0,822$^{\blacklozenge}$ (0,63)	78,686	0,023	0,834	2,275	7,043	$\chi^2(1)$=1,79, p=0,181
	⊘	-1,609$^{\triangleleft}$ (0,54)	0,740$^{\blacklozenge}$ (0,63)	79,04	0,019	0,767	2,096	6,499	$\chi^2(1)$=1,44, p=0,230
Ant	⋆	-1,312$^{\triangleleft}$ (0,42)	0,149$^{\blacklozenge}$ (0,51)	107,26	0,001	0,503	1,16	2,814	$\chi^2(1)$=0,08, p=0,773
	⊘	-1,550$^{\triangledown}$ (0,41)	0,545$^{\blacklozenge}$ (0,51)	106,18	0,011	0,755	1,72	4,158	$\chi^2(1)$=1,16, p=0,281

R^2=Hosmer–Lemeshow (131)
Significância dos Coeficientes (R®): 0,001$^{\triangledown}$; 0,01$^{\triangleleft}$; 0,05$^{\triangleright}$; 0,1$^{\triangle}$; 1$^{\blacklozenge}$
$^{\otimes}$Para $\alpha = 0,10$, o modelo com a variável β_1 é significativamente melhor do que aquele com apenas a constante ($p < \alpha$).
*Modelo variando a intensidade do *smell* Complex Class na co-ocorrência de Large & Complex Class
$^{\oslash}$Modelo variando a intensidade do *smell* Large Class na co-ocorrência de Large & Complex Class

Tomando como base os modelos representados por "⋆" dos projetos ArgoUML, Cassandra e Lucene, observa-se que o R^2 está entre 2,7% e 11,3%. Isso quer dizer que a intensidade do *smell* Complex Class explica até 11,3% da variabilidade dos dados. Indicando que além da intensidade deste *smell* há outros fatores que ajudam a explicar a prevalência de clones. Além disso, a métrica OR varia entre 2,2 e 5,9, indicando que na co-ocorrência dos *smells* LC/CC as chances de ocorrer clones entre as entidades mais complexas é de 2,2 a 5,9 vezes maior do que as chances de haver clones entre as entidades menos complexas. Novamente, essas considerações são consistentes com aquelas apresentadas nos resultados da RQ1.1.

Os resultados das RQs deste grupo indicam que a intensidade da complexidade da entidade está mais associada à prevalência de clones do que a intensidade de tamanho das entidades. Também observamos que a co-ocorrência dos *smells* Large e Complex Class pode potencializar as chances de ocorrer clones, especialmente nas entidades em que o *smell* CC ocorre em maior intensidade. Outro fato revelador é que independente da forma com que estes *smells* ocorrem, a existência de clones explicam no máximo 11,3% da prevalência dos *smells* estudados. Indicando que outros fatores também estão associados à prevalência de clones.

4.2.2.2 Associação da Quantidade de *Smells* a Clones

Os resultados das duas questões de pesquisa relacionadas à prevalência de clones em relação à quantidade tipos de *smells*, desconsiderando-se a intensidade dos *smells*, são descritos nas próximas subseções.

RQ2.1 — Desconsiderando a intensidade dos *smells*, existe diferença na prevalência de clones entre as entidades que apresentam co-ocorrência de *smells* (LC/CC) e as entidades que apresentam apenas o *smell* CC? Se sim, quanto esse comportamento está associado?

Nessa questão de pesquisa avaliamos a relação da prevalência de clones entre as entidades que apresentam apenas o *smell* CC e aquelas que apresentam a co-ocorrência dos *smells* de interesse (LC/CC). Deste modo, a variável preditora desta questão de pesquisa é a quantidade de *smells* de interesse que ocorrem nas entidades (classes). Para avaliar isso, os dados usados nas tabelas de contingências da RQ1.1 e RQ1.3 foram reorganizados de modo a desconsiderar a intensidade dos seus respectivos *smells*. Assim, a linha de saldo total da Tabela 7 está representada pela linha denominada CC na Tabela 14 e, analogamente, a linha de totalização da Tabela 11 está representada pela linha denominada LC&CC nessa mesma tabela.

Tabela 14 – Tabela de Contingência — RQ2.1: Entidades CC *vs.* LC&CC

	ArgoUML			**Cassandra**			**Lucene**			**Hadoop**			**Ant**		
	Clone		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**
	Não	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**	
CC	22	15	**37**	36	8	**44**	59	41	**100**	22	3	**25**	27	6	**33**
LC&CC	79	40	**119**	92	18	**110**	85	75	**160**	51	17	**68**	74	22	**96**
Total	**101**	**55**	**156**	**128**	**26**	**154**	**144**	**116**	**260**	**73**	**20**	**93**	**101**	**28**	**129**

A execução da regressão logística necessita que o valor esperado nas células da tabela de contingência sejam maiores que dois. Para os dados da Tabela 14 observa-se que todos os cinco projetos de software analisados satisfazem essa condição. Portanto, os dados da regressão logística destes projetos devem ser analisados (ver Tabela 15).

Analisando a Tabela 15 observa-se que a variável preditora **não** é estatisticamente relevante em nenhum dos cinco projetos de software, ou seja, o modelo *baseline* é melhor que aquele onde a variável preditora é considerada. Portanto, para nossa amostra de software, **não** há evidência estatística significante de que a prevalência de clones nas entidades que apresentem apenas o *smell* CC seja diferente da prevalência de clones ocorrida nas entidades onde os *smells* LC&CC co-ocorrem.

Tabela 15 – Modelos de Regressão — RQ2.1: Entidades CC *vs.* LC&CC

Projeto	**Constante (SE)**	β_1 **(SE)**	**AIC**	$\mathbf{R^2}$	**IC para** β_1			***Baseline vs. New***
					Inferior (5%)	**OR**	**Superior (95%)**	
ArgoUML	-0,383$^{\blacklozenge}$ (0,33)	-0,297$^{\blacklozenge}$ (0,38)	205,91	0,003	0,394	0,742	1,415	$\chi^2(1)$=0,59, p=0,444
Cassandra	-1,504$^{\triangledown}$ (0,39)	-0,127$^{\blacklozenge}$ (0,46)	143,77	0,001	0,415	0,880	1,962	$\chi^2(1)$=0,07, p=0,787
Lucene	-0,364$^{\triangle}$ (0,20)	0,238$^{\blacklozenge}$ (0,25)	360,55	0,002	0,832	1,269	1,944	$\chi^2(1)$=0,86, p=0,353
Hadoop	-1,992$^{\triangleleft}$ (0,61)	0,893$^{\blacklozenge}$ (0,67)	98,82	0,021	0,873	2,444	8,496	$\chi^2(1)$=2,00, p=0,157
Ant	-1,504$^{\triangledown}$ (0,45)	0,291$^{\blacklozenge}$ (0,51)	138,64	0,002	0,595	1,337	3,274	$\chi^2(1)$=0,33, p=0,564

R^2=Hosmer–Lemeshow (131)
Significância dos Coeficientes ($\mathsf{R}^{®}$): 0,001$^{\triangledown}$; 0,01$^{\triangleleft}$; 0,05$^{\triangleright}$; 0,1$^{\triangle}$; 1$^{\blacklozenge}$
$^{\otimes}$Para $\alpha = 0,10$, o modelo com a variável β_1 é significativamente melhor do que aquele com apenas a constante ($p < \alpha$).

A análise estatística da amostra de cinco software, desconsiderando a intensidade de cada *smell* de interesse, indica que **não** há nenhuma evidência estatisticamente significante de que a prevalência de clones nas entidades que apresentam apenas o *smell* CC seja diferente da prevalência de clones ocorrida nas entidades onde os *smells* LC&CC co-ocorrem.

RQ2.2 — Desconsiderando a intensidade dos *smells*, existe diferença na prevalência de clones entre as entidades que apresentam co-ocorrência de *smells* (LC/CC) e as entidades que apresentam apenas o *smell* LC? Se sim, quanto esse comportamento está associado?

Essa questão de pesquisa é semelhante a RQ2.1 apresentada na subseção anterior e, portanto, sua análise segue os mesmos princípios e procedimentos descritos na subseção anterior. Observe que nessa RQ estamos interessados em analisar prevalência de clones entre as entidades apenas LC e aquelas com a co-ocorrência dos *smells* de interesse (LC/CC). Portanto, a variável preditora dos modelos ainda é a quantidade de *smells* de interesse que ocorrem nas entidades, contudo, o conjunto de dados é diferente.

Com o intuito de caracterizar o conjunto de dados analisados, os mesmos foram tabulados na tabela de contingência 16. Os dados dessa representação são os mesmos apresentados na Tabela 9 e 11, entretanto, eles foram reorganizados de forma a desconsiderar a intensidade dos respectivos *smells*.

Analisando a Tabela 16, observa-se que o valor mínimo de todas nas células da tabela de contingência atendem à condição necessária da regressão logística. Assim, todos os modelos da Tabela 17 devem ser analisados.

Tabela 16 – Tabela de Contingência — RQ2.2: Entidades LC *vs.* LC&CC

	ArgoUML			**Cassandra**			**Lucene**			**Hadoop**			**Ant**		
	Clone		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**	**Clone**		**Total**
	Não	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**		**Não**	**Sim**	
LC	39	24	**63**	33	5	**38**	33	21	**54**	21	5	**26**	5	5	**10**
LC&CC	79	40	**119**	92	18	**110**	85	75	**160**	51	17	**68**	74	22	**96**
Total	**118**	**64**	**182**	**125**	**23**	**148**	**118**	**96**	**214**	**72**	**22**	**94**	**79**	**27**	**106**

Tabela 17 – Modelos de Regressão — RQ2.2: Entidades LC *vs.* LC&CC

Projeto	**Constante (SE)**	β_1 **(SE)**	**AIC**	R^2	**IC para β_1**			***Baseline vs. New***
					Inferior (5%)	**OR**	**Superior (95%)**	
ArgoUML	-0,485$^{\triangle}$ (0,25)	-0,195$^{\blacklozenge}$ (0,32)	239,68	0,002	0,483	0,822	1,407	$\chi^2(1)$=0,36, p=0,548
Cassandra	-1,887$^{\triangledown}$ (0,47)	0,255$^{\blacklozenge}$ (0,54)	131,64	0,002	0,550	1,291	3,387	$\chi^2(1)$=0,23, p=0,633
Lucene	-0,452$^{\blacklozenge}$ (0,27)	0,326$^{\blacklozenge}$ (0,32)	297,35	0,004	0,821	1,386	2,367	$\chi^2(1)$=1,05, p=0,306
Hadoop	-1,435$^{\triangleleft}$ (0,49)	0,336$^{\blacklozenge}$ (0,57)	105,93	0,004	0,568	1,400	3,807	$\chi^2(1)$=0,36, p=0,549
Ant	$7,249E-15^{\blacklozenge}$ (0,63)	-1,213$^{\triangle}$ (0,67)	121,21	0,026	0,095	0,297	0,922	$\chi^2(1)$=3,09, p=0,079$^{\otimes}$

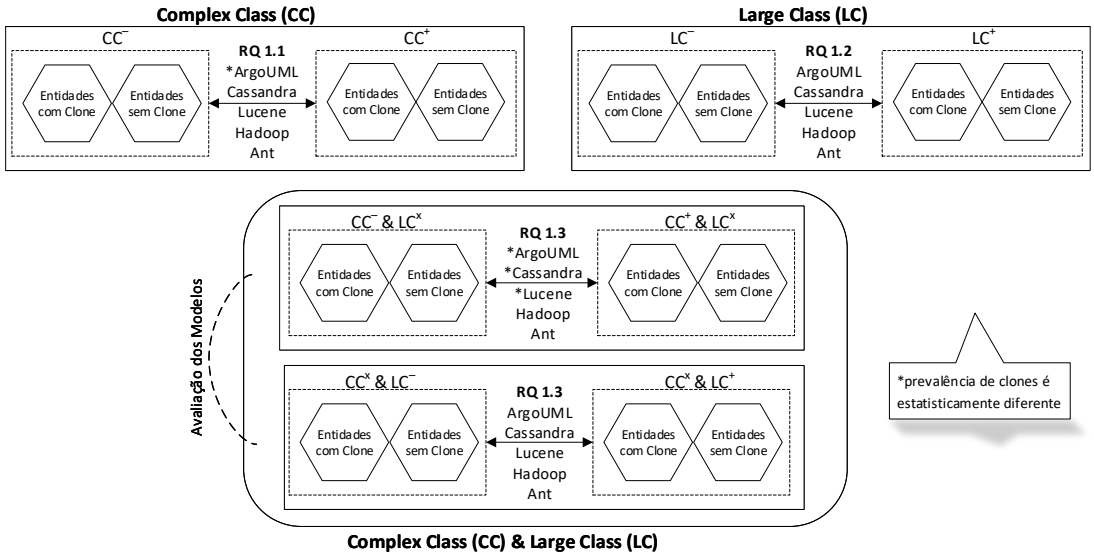
R^2=Hosmer–Lemeshow (131)
Significância dos Coeficientes (R$^{®}$): 0,001$^{\triangledown}$; 0,01$^{\triangleleft}$; 0,05$^{\triangleright}$; 0,1$^{\triangle}$; 1$^{\blacklozenge}$
$^{\otimes}$Para $\alpha = 0,10$, o modelo com a variável β_1 é significativamente melhor do que aquele com apenas a constante ($p < \alpha$).

Os modelos da Tabela 17 revelam que a inclusão da variável preditora relativo à quantidade de *smells* foi estatisticamente relevante em apenas um caso, aquele que modela os dados do projeto Ant. Para este modelo, observa-se que a métrica OR é menor do que um (0,29), indicando que a ocorrência de clones está mais associada à ocorrência isolada do *smell* Large Class do que a co-ocorrência dos *smells* Large & Complex Class. Analisando o R^2, observa-se que apenas 2,6% da variabilidade dos dados pode ser explicada pelo modelo. Na prática, isso indica que além da existência deste *smell* há outros fatores que ajudam a explicar a prevalência de clones.

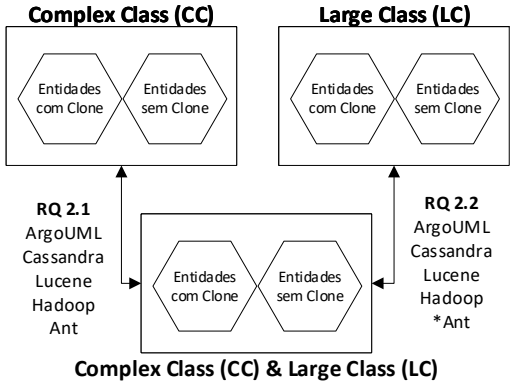
Encontramos evidência de que entre as entidades acometidas pelo *smell* Large Class (LC) e aquelas acometidas pela co-ocorrência dos *smells* Large & Complex Class, ao desconsiderar a intensidade dos *smells*, clones estão mais associados à existência/prevalência singular de *smells* (LC). Em especial, o modelo do projeto Ant indica que para cada incremento no tamanho da entidade, a prevalência de clones aumenta em uma taxa de 0,29.

4.3 Discussão dos Resultados

Essa subseção discute qualitativamente os resultados estatísticos apontados nas subseções anteriores. Para cada uma das RQs, a Figura 18 apresenta resumidamente quais modelos dos projetos de software analisados foram estaticamente significantes segundo os procedimentos executados na regressão logística. Observe que os modelos significantes estão marcados com o símbolo "*" e os próximos parágrafos discutem objetivamente esses modelos segundo a análise qualitativa da prevalência de clones.



(a)



(b)

Figura 18 – Resumo dos modelos avaliados nas RQs. (a) Considerando a intensidade dos *smells* (b) Desconsiderando a intensidade dos *smells*

A análise qualitativa dos resultados das RQs que apresentam no mínimo um dos modelos de regressão logística estatisticamente significante ocorre verificando-se aquelas classes acometidas por clones e pelo(s) respectivo(s) *smell(s)* abordados na questão de pesquisa em apreço. Adicionalmente, são analisadas apenas as classes daqueles projetos em que o modelo estatístico se mostrou significante, exemplo: a RQ1.1 analisa-se todas

as classes com clones que apresentam apenas o *smell* Large Class, isto apenas para o projeto ArgoUML.

4.3.1 Com Intensidade de *Smells*

Essa subseção apresenta a discussão das três primeiras perguntas de pesquisa, a saber: RQ1.1, RQ1.2 e RQ1.3.

RQ1.1 — Considerando a intensidade de *smells* apenas nas entidades CC (excluise as entidades de co-ocorrência CC/LC): dos cinco projetos analisados, os dados do ArgoUML foram os únicos estatisticamente significantes. Para este projeto, observamos que as chances de ocorrer clones entre as entidades mais complexas é 3,93 vezes maior do que as chances de haver clones entre as entidades menos complexas. Os dados deste projeto revelam 15 classes com clones que também sofrem do *smell* Complex Class (ver Tabela 7). Destas classes, oito estão classificadas no menor nível de intensidade do *smell* Complex Class (CC_1) e o restante está no maior nível (CC_2). Nos próximos parágrafos as classes com clones destes níveis de intensidade do *smell* CC são examinadas.

Código 4 – Clone contido na classe CrUnconventionalAttrName do ArgoUML.

```
1  ...
2  @Override
3  public boolean stillValid (ToDoItem i, Designer dsgr) {
4   if (!isActive()) {
5    return false;
6   }
7   ListSet offs = i.getOffenders();
8   Object f = offs.get(0);
9   if (!predicate(f, dsgr)) {
10   return false;
11  }
12  ListSet newOffs = computeOffenders(f);
13  boolean res = offs.equals(newOffs);
14  return res;
15 }
```

Analisando os fragmentos de clones de todas as classes que estão no nível CC_1 observamos que a maior parte (91%) refere-se a clones simples e/ou relativamente pequenos. Como exemplo, a classe CrUnconventionalAttrName apresenta três fragmentos de clones que são considerados simples e um destes está parcialmente apresentado no Código 4. Observe que este clone é de uma funcionalidade que apresenta baixa complexidade. Ainda para os clones do nível CC_1, também observamos que o *template* de código(s) usado pelo(s) desenvolvedor(es) parece exercer alguma influência no tamanho dos clones, neste caso intensificando o tamanho dos clones. A título de exemplo, o Código 5 demonstra

um fragmento de clone em que o tratamento de exceção e/ou condição indesejada têm tratamentos semelhantes. Esse tipo de observação é especialmente prevalente nos clones em que o comando *throw/try/catch* ocorre de forma aninhada. Notamos casos em que o aninhamento destes comandos representou até 43% do referido clone.

Código 5 – Clone contido na classe ModelElementNameNotationJava do ArgoUML.

```
1   ...
2       if (name != null) {
3        String msg = "parsing.error.model-element-name.twin-names";
4        throw new ParseException(Translator.localize(msg),
5        st.getTokenIndex());
6       }
7       name = token;
8      }
9     }
10  } catch (NoSuchElementException nsee) {
11   String msg = "parsing.error.model-element-name.unexpected-name-element";
12   throw new ParseException(Translator.localize(msg),
13   text.length());
14  } catch (ParseException pre) {
15   throw pre;
16  }
```

Por outro lado, examinando os fragmentos de clones de todas as classes que estão no nível CC_2 apresentam forma, tamanho e/ou complexidade diferente daqueles do nível CC_1. Neste conjunto de dados, não encontramos clones de funcionalidades completas como aquele apresentado no Código 4, aqui, os clones são fragmentos de código dentro de uma funcionalidade mais extensa. A exemplo, a classe AttributeNotationUml apresenta o método parseAttribute que possui 229 linhas (característica do *smell* Long Method), internamente esse método apresenta dois clones diferentes: (i) um na região que realiza o *parse* quando o *token* é o símbolo "{" e por fim, (ii) temos o clone na região de tratamento de exceção. Além disso, os clones no nível CC_2 apresentam mais estruturas de desvio de fluxo e portanto, são mais complexos. Nos fragmentos de clones, notamos que dentre as estruturas sujeitas a desvio de fluxo do código (ex. *if-then-else*, *while*, *for*, *do-while*, *switch-case*), a estrutura *if* e suas derivações é a mais recorrente. Em comparação com os fragmentos de clones do nível CC_1 os fragmentos de clones do nível CC_2 são maiores em relação à média de linhas de código e estão concentrados nos maiores métodos das classes.

Na RQ1.1 os resultados da análise qualitativa do ArgoUML demonstra que a complexidade exerce um papel na forma, tamanho e complexidade com que os desenvolvedores "criam" clones.

RQ1.2 — Considerando a intensidade de *smells* apenas nas entidades LC (exclui-se as entidades de co-ocorrência CC/LC): Pela análise estatística observa-se que a intensidade

do *smell* LC **não** é um fator relevante na prevalência de clones para nenhum dos projetos de software analisados. Independente do projeto de software, a Tabela 9 da RQ1.2 denota poucas classes classificadas no maior nível de intensidade do *smell* Large Class (LC_2), e a maior parte das classes está no grupo LC_1. Comparando-se a quantidade de classes que está no maior nível de intensidade dos *smells* apresentados na Tabela 7 (Apenas CC_2) e 9 (Apenas LC_2) nota-se uma diferença significativa, em termos proporcionais, dos valores totais, exemplo: no projeto ArgoUML, a Tabela 7 denota 29,7% das classes classificadas no maior nível de intensidade do *smell* Complex Class e na Tabela 9 observa-se que apenas 12,6% das classes estão classificadas no maior nível de intensidade do *smell* Large Class. Esse comportamento se repete em praticamente todos os dados da RQ1.2.

Segundo Palomba et al. (48), o tamanho das entidades (classes/métodos) geralmente é um fator visto pelos desenvolvedores como uma importante ameaça ao software. O baixo número de classes no grupo LC_2 indica que provavelmente os desenvolvedores e/ou engenheiros de software atuam ativamente para evitar essa situação e certamente isso contribui para os resultados estatísticos desta RQ. Por outro lado e, ainda, segundo Palomba et al. (48), a complexidade também é vista como um fator de ameaça. Contudo, na prática, supomos que os desenvolvedores têm mais dificuldade para controlar a complexidade das classes do que o tamanho (número de responsabilidades) delas propriamente dito. Talvez, isso possa estar relacionado às práticas/métodos estabelecidos na engenharia de software: (i) quando uma classe é grande, ou seja, acumula muitas responsabilidades pouco relacionadas, é intuitivo que as responsabilidades que são mais relacionadas entre si sejam movidas para outra(s) classe(s). Na prática, geralmente certos métodos da classe grande são movidos para outra classe, neste caso, a ação é realizada assegurando-se a compatibilidade com as entidades que são clientes destes métodos movidos. (ii) Por outro lado, aplicar esse tipo de técnica para controlar a complexidade de uma classe pode resultar apenas em mover a complexidade de uma classe para outra. Então o controle da complexidade exige que o desenvolvedor faça uma análise minuciosa no nível de linhas de código, o que lhe permite decidir qual parte do código deve ser movida para outro método e/ou outra classe. Isso torna o controle da complexidade uma tarefa dispendiosa em termos de tempo e mais arriscada, pois, ao refatorar o código, o comportamento da funcionalidade não deve ser modificada. Portanto, o controle da complexidade se torna menos frequente do que o controle de tamanho.

Essa hipótese de que os desenvolvedores e/ou engenheiros de software atuam controlando o tamanho das classes explica porque não há diferença estatística na prevalência de clones entre as classes LC_1 e LC_2. Neste caso, poucas classes no grupo LC_2 também significa que há poucos clones nestas entidades e, portanto, não há diferença estatística na proporção de clone entre estes grupos. Essa hipótese pode ser observada empiricamente pela análise histórica destas entidades ao longo das diversas versões de cada projeto de software, o que será realizado nos próximos passos desta pesquisa.

Assim, o resultado da análise estatística de que a intensidade do *smell* LC não se relaciona à prevalência de clones é congruente com o comportamento dos desenvolvedores, pois estes provavelmente atuam mantendo o controle do tamanho das classes. Portanto, era esperado não haver relacionamento entre a intensidade isolada do *smell* Large Class e a prevalência de clones destas entidades.

RQ1.3 — Considerando a intensidade de *smells* apenas nas entidades onde LC e CC co-ocorrem: Os modelos estatísticos baseados na intensidade do *smell* CC, são os únicos relevantes em comparação com o *baseline*, neste caso, em três projetos de software. Os resultados da métrica OR destes modelos varia entre 2,2 e 5,9, indicando que na co-ocorrência dos *smells* LC/CC as chances de ocorrer clones entre as entidades mais complexas é de 2,2 a 5,9 vezes maior do que as chances de haver clones entre as entidades menos complexas. Em comparação com os resultados da RQ1.1, observa-se que a maior intensidade do *smell* CC na co-ocorrência com o *smell* LC pode potencializar as chances de clones. Nos próximos parágrafos analisamos as classes com clones e que apresentam a co-ocorrência dos *smells* de interesse (Tabela 11). Neste caso, investigamos como os fragmentos de clones que ocorrem nestas classes estão distribuídos ao longo das entidades de cada projeto. Essa análise é realizada apenas para os projetos ArgoUML, Cassandra e Lucene, pois seus modelos de regressão foram estatisticamente significantes.

Por definição, a existência de clones exige que no mínimo dois fragmentos de códigos sejam semelhantes entre si conforme algum critério. Estes fragmentos de código clonados podem estar na mesma classe (arquivo) ou em classes distintas, o que permite realizar uma análise considerando a classificação dos *smells* que é atribuída a cada classe do projeto de software. Assim, os clones podem ser analisados conforme a classificação dos pares de classes em que eles ocorrem. Neste estudo empírico, as classes podem ser classificadas conforme a existência de determinados *smells*: (i) **Co-ocorrência**, os *smells* Large e Complex Class existem simultaneamente na mesma classe; (ii) **Apenas LC**, a classe não apresenta o *smell* CC, contudo exibe o *smell* LC; (iii) **Apenas CC**, similar ao anterior mas do ponto de vista do *smell* CC e (iv) **Outros**, a classe não exibe qualquer um dos *smells* de interesse (LC e/ou CC).

Pela Tabela 11, o projeto ArgoUML apresenta 40 classes com clones, sendo que 12 classes demonstram a co-ocorrência de *smells* e apresentam o *smell* CC no menor nível de intensidade. Por outro lado, 28 classes com clones têm a co-ocorrência de *smells*, mas apresentam o *smell* CC no maior nível de intensidade. Ainda por esta tabela, observa-se que o projeto Cassandra e Lucene apresentam, respectivamente, 18 e 75 classes com clones e que têm a co-ocorrência de *smells*. No contexto de clones, cada uma destas classes podem apresentar mais de um fragmento de código clonado. Como exemplo, a classe FigPackage do projeto ArgoUML apresenta dois fragmentos de clones: o primeiro surge entre as linhas 1028 e 1054, o segundo entre as linhas 820 e 844. O primeiro destes clones é entre os pares de classes: FigPackage e FigObject, que são respectivamente classificadas

como: **co-ocorrência** onde o *smell* CC se apresenta no maior nível de intensidade e **apenas LC** onde o *smell* LC se manifesta no menor nível de intensidade. Portanto, a ocorrência específica deste clone é manifestada pela interação de código entre duas classes com classificações distintas (CC_2&LC_x e LC_1). Assim, cada ocorrência de clone pode ser agrupada conforme a classificação de *smells* dos pares de classes em que eles ocorrem. A Tabela 18 detalha esse agrupamento considerando apenas as classes com clones dos projetos ArgoUML, Cassandra e Lucence que são apresentadas na Tabela 11.

Os dados da Tabela 18 demonstram que os fragmentos de clones das classes existem, em sua maior parte, entre entidades onde os *smells* LC e CC co-existem. Proporcionalmente, observamos poucos casos em que um fragmento de clone ocorre entre entidades onde uma delas **não** apresenta qualquer um dos *smells* de interesse (LC e/ou CC). Isso indica que os fragmentos de clones das classes grandes e complexas são restritos a elas mesmas, ou seja, aquelas classes onde os *smells* LC e CC co-ocorrem, especialmente naquelas em que o *smell* CC ocorre com maior intensidade. Esse comportamento pode explicar porque a prevalência de clones em relação à co-ocorrência destes *smells* é estatisticamente significante nestes projetos, ou seja, a maior incidência de clones nas classes que têm determinada característica aumenta as chances de que os grupos de classes com característica distintas sejam estatisticamente diferentes. Isso também explica porque a maior intensidade do *smell* CC na co-ocorrência com o *smell* LC pode potencializar as chances de clones.

Tabela 18 – Ocorrência de clones conforme a classificação dos pares de classes.

		ArgoUML		**Cassandra**		**Lucene**	
		Co-ocorrência		**Co-ocorrência**		**Co-ocorrência**	
		CC_1&LC_x	CC_2&LC_x	CC_1&LC_x	CC_2&LC_x	CC_2&LC_x	
Co-ocorrência	CC_1&LC_x	8	5	0	0	6	5
	CC_2&LC_x	5	27	0	23	5	124
Apenas CC	CC_1	1	2	0	0	0	2
	CC_2	0	5	0	0	1	5
Apenas LC	LC_1	14	7	0	2	8	0
	LC_2	2	0	0	0	1	1
Outros Tipos de Classes		2	6	1	2	14	6
Total Geral de Pares		**84**		**28**		**178**	

Com o intuito de verificar o tipo de clone (inter-, intra- e mix-classe) que mais ocorre, nos próximos parágrafos, analisamos as classes com clones dos projetos: ArgoUML, Cassandra e Lucene, pois seus modelos de regressão foram estatisticamente significantes. Neste caso, analisamos cada fragmento de clone das classes que manifestam um ou mais

smells de interesse, ou seja, as classes com clones que estão apresentadas nas Tabelas 7, 9 e 11.

Analisando as classes acometidas por clones do projeto ArgoUML, temos um total de 15 classes que também sofrem apenas do *smell* CC (ver Tabela 7), 24 classes que apresentam apenas o *smell* LC (ver Tabela 9) e, por fim, temos 40 classes com a co-ocorrência dos *smell* CC/LC (ver Tabela 11). Para o projeto ArgoUML, os clones das classes que sofrem de apenas um *smell* (CC ou LC) são clones inter-classe em 95,8% das vezes. Por outro lado, os clones das classes de co-ocorrência dos *smells* CC e LC são clones inter-classe em 72,3% dos casos. Fazendo essa mesma análise para o projeto Cassandra, obtivemos resultados semelhantes aos encontrados no ArgoUML. Por fim, o projeto Lucene apresentou diferenças percentuais mais significativas: realizando a mesma análise observa-se que os clones das classes que sofrem de apenas um *smell* (CC ou LC) são clones inter-classe em 70,9% das vezes e os clones nas classes com co-ocorrência de *smells* são clones inter-classe em apenas 36,2% dos casos. Isso demonstra um indício de que a co-ocorrência destes *smells* influencia na forma com que os clones são "criados" pelos desenvolvedores e reforça a observação de que os clones que ocorrem nas classes com a co-ocorrência dos *smells* de interesse são clones restritos à entidades com estas características. Supomos que alterar recursos/funcionalidades implementadas em classes grandes e complexas seja uma tarefa complicada, devido à quantidade de responsabilidades implementadas, esse tipo de entidade, geralmente, se relaciona a muitas outras e, complementarmente, o entendimento da funcionalidade a ser alterada demanda tempo, bem como conhecimento de certas partes do sistema. Assim, para classes grandes e complexas, acreditamos que os desenvolvedores preferem realizar a alteração da funcionalidade em uma cópia de si mesma do que efetivamente alterar a funcionalidade original. Isso poderia explicar porque a taxa de clones do tipo inter-classe é sempre menor nas entidades de co-ocorrência dos *smells* Large e Complex Class do que nas entidades em que estes *smells* ocorrem isoladamente.

Nota-se que, independente do sistema analisado e dos *smells* envolvidos na classe (LC e/ou CC), as classes que se apresentam como mix-classe são pouco frequentes. Em regra, uma classe do tipo mix-classe ocorre quando, no mínimo, um de seus clones são fragmentos de código que também está em outra(s) classe(s) e que um outro fragmento de código clonado aparece em locais diferentes da mesma classe. Isso quer dizer que uma classe do tipo mix-classe apresenta no mínimo dois fragmentos de clones independentes entre si e cada um tem sua própria classificação (inter- e intra-classe). Esse fenômeno pode ser observado no Código 6: no lado esquerdo temos um clone que ocorre entre as classes FigNodeModelElement e FigEdgeModelElement, do lado oposto, temos um clone que ocorre nas linhas 980 e 1002 da mesma classe FigNodeModelElement. Portanto, a classe FigNodeModelElement se apresenta como clones mix-classe. Considerando que um dado fragmento de clone pode ocorrer entre múltiplas classes, existe a possibilidade que um mesmo fragmento de clone possa ser simultaneamente um clone que ocorre inter- e intra-classe. Contudo, analisando

as classes do tipo mix-classe dos projetos ArgoUML, Cassandra e Lucene, não encontramos um mesmo fragmento de clone que seja, simultaneamente, Intra-classe e Inter-classe. Encontramos apenas classes com mais de um fragmento de clone onde cada um tem sua própria classificação específica, como o exemplo apresentado no Código 6.

Código 6 – Classe FigNodeModelElement do tipo mix-classe (ArgoUML).

Clone Intre-Classe

```
1  protected void addElementListener(Object
       element) {
2   listeners .add(new Object[] {element, null
       });
3   Model.getPump().addModelEventListener(
       this, element);
4  }
5  protected void addElementListener(Object
       element, String property) {
6   listeners .add(new Object[] {element,
       property});
7   Model.getPump().addModelEventListener(
       this, element, property);
8  }
9  protected void addElementListener( Object
       element, String[] property) {
10  listeners .add(new Object[] {element,
       property});
11  Model.getPump().addModelEventListener(
       this, element, property);
12 }
13 ...
```

Clone Intra-Classe.

```
1  List<ToDoItem> items = tdList.
       elementListForOffender(getOwner());
2  for (ToDoItem item : items) {
3   Icon icon = item.getClarifier();
4   int width = icon.getIconWidth();
5   if (y >= getY() - 15 && y <= getY() +
       10 && x >= iconX && x <= iconX +
       width) {
6    return item;
7   }
8   iconX += width;
9  }
10 for (ToDoItem item : items) {
11  Icon icon = item.getClarifier();
12  if (icon instanceof Clarifier) {
13   ((Clarifier) icon).setFig(this);
14   ((Clarifier) icon).setToDoItem(item);
15   if (((Clarifier) icon).hit(x, y)) {
16    return item;
17   }
18  }
19 }
```

A discussão dos resultados desta RQ demonstra que: (i) os clones que ocorrem nas entidades que apresentam simultaneamente os *smells* LC e CC, são clones restritos às entidades com essa característica; (ii) a co-ocorrência dos *smells* de interesse influencia na forma com que os clones são "criados"; (iii) há indícios de que os clones nas classes acometidas pelos *smells* Large Class e/ou Complex Class são operações oportunísticas.

4.3.2 Sem Intensidade de *Smells*

Essa subseção apresenta a discussão das duas últimas perguntas de pesquisa, a saber: RQ2.1 e RQ2.2.

RQ2.1 — Desconsiderando a intensidade de *smells* nas entidades que apresentam CC (inclui-se as de co-ocorrência CC/LC): Dentre os *smells* analisados, a quantidade de

tipos de *smells* que co-existem nas entidades acometidas pelo *smell* CC **não** é relevante na prevalência de clones. Nessa RQ dois grupos de entidades são analisados, um grupo que sofre apenas do *smell* CC e o outro grupo que apresenta a co-ocorrência LC e CC. Então, em termos de *smells*, a única diferença entre esses grupos é que no segundo grupo as entidade também são acometidas pelo *smell* LC. Assim, se considerarmos o que foi estabelecido na discussão da RQ1.2 observa-se que o *smell* LC não tem grandes chances de exercer um papel significativo na prevalência de clones. Portanto, efeito atribuído à ocorrência do *smell* LC existente no segundo grupo se torna estatisticamente nula. Assim, para o caso desta RQ o resultado relatado já era esperado e está completamente alinhado à discussão realizada nas outras RQs.

RQ2.2 — Desconsiderando a intensidade de *smells* nas entidades que apresentam LC (inclui-se as de co-ocorrência CC/LC): Para este caso, a quantidade de tipos de *smells* que co-existem nas entidades é inversamente relevante na prevalência de clones em pelo menos um caso, o projeto Ant. Assim, a ocorrência de clones está mais associada à ocorrência isolada do *smell* Large Class do que à co-ocorrência dos *smells* Large & Complex Class ($\beta_1 = -1,213, OR = 0,29$). Analisando o projeto Ant, observa-se que este apresenta uma peculiaridade nos dados relativos às entidades acometidas apenas pelo *smell* Large Class: este projeto manifesta um total de 10 entidades com apenas LC (ver Tabela 9 e 16) e, deste valor, 50% apresentam o *smell* Duplicate Code (DC). Nota-se que 80% destas classes com DC são "variações" da mesma entidade, exemplo: a classe CCMklbtype apresenta atributos diferentes daqueles contidos na classe CCMkattr; contudo elas manifestam: (i) métodos idênticos (ex. getCommentCommand, getVersionCommand), (ii) métodos parcialmente clonados (ex. execute, checkOptions) e (iii) métodos próprios (ex. getVOB, getTypeValueCommand). Isso demonstra que estes clones são praticamente os mesmos e estão distribuídos em diversas entidades, indicando que estas entidades fornecem pouca variabilidade de dados para análise. Estes clones poderiam ser facilmente evitados caso certos princípios da programação orientada a objetos (ex. herança, polimorfismo) tivessem sido aplicados corretamente nestas quatro entidades. Analisando os *commits* destas classes observamos que as alterações realizadas em uma destas classes são, muitas vezes, replicadas nas outras e, algumas vezes, as alterações são ligeiramente diferentes. Isso reforça que certos princípios da programação orientada a objetos não foram corretamente aplicados. Outro aspecto interessante é que dentre as classes acometidas apenas pelo *smell* LC (análise da RQ1.2) este projeto foi o único que não apresentou classes LC no maior nível de tamanho (Tabela 9), inclusive esse comportamento eliminou seu modelo da análise realizada na RQ1.2. Estas características demonstram que este projeto apresenta baixa variabilidade de dados e certamente isso limita os resultados da RQ2.2 o que permite a seguinte interpretação: Os dados estatísticos demonstram que desconsiderando a intensidade dos *smells* de interesse e para as entidades acometidas pelo *smell* LC, incluindo aquelas de co-ocorrência (LC/CC), existe a possibilidade de que a ocorrência

de clones está mais associada à ocorrência isolada do *smell* LC do que a co-ocorrência de LC&CC. Contudo, a análise semântica dos dados revela que essa possibilidade é remota, especialmente nos dados do projeto Ant.

4.3.3 Possíveis Implicações Práticas

Em geral, as possíveis implicações estão relacionadas aos padrões dos clones apontados no estudo empírico descrito no capítulo anterior. Alguns destes padrões e suas aplicações estão detalhadas nos próximos parágrafos.

Na discussão da RQ1.3 mostramos que os fragmentos de clones das classes grandes e complexas são em grande parte restritos a elas mesmas, ou seja, os clones que surgem nas classes onde os *smells* LC e CC co-ocorrem são clones confinados a classes que seguem este comportamento. Isso quer dizer que há poucas chances de ocorrer clones entre uma classe que é grande e complexa, e uma classe que não apresenta essa característica. Esse comportamento, descoberto neste estudo empírico, pode ser usado para otimizar a detecção de *smells*, a exemplo Duplicate Code. Em geral, como não se sabe de antemão quais fragmentos de código estão duplicados e onde estão, então, a ferramenta de detecção de clones inevitavelmente precisa comparar todos os fragmentos de código com todos os outros fragmentos de código. Na prática, um fragmento de código pode ser definido por um dado número de linhas de código e quanto menor o tamanho do fragmento de código a ser comparado com todos os outros, maior é o espaço de busca que a ferramenta deve percorrer para identificar se aquele fragmento é ou não um clone. Portanto, dependendo do tamanho do fragmento de código e/ou do tamanho do projeto de software, a ferramenta pode levar um tempo considerável para identificar os clones do projeto. Considerando que a maior parte dos clones que ocorrem nas entidades com os *smells* LC e CC são restritos a elas mesmas, o espaço de busca de clones nestas entidades pode ser reduzido. Portanto, para esta situação, podemos diminuir o tempo de busca. Contudo, este tipo de aplicação requer estudos complementares sobre este comportamento, em especial, usando um universo maior de projetos.

Outra possível implicação prática se refere ao planejamento das atividades de refatoração. Pela discussão, observa-se que grande parte dos fragmentos de clones que ocorrem nas entidades acometidas por apenas um *smell* (CC ou LC) são clones do tipo inter-classe. Isso implica que a remoção destes clones passa pela aplicação das técnicas de refatoração: *Extract Superclass* ou *Extract Class*. Por outro lado, apenas uma parcela dos clones ocorridos nas entidades acometidas pela co-ocorrência dos *smells* Large Class e Complex Class são clones do tipo inter-classe. Para os clones destas entidades, a maior parte pode ser removido usando a técnica de refatoração *Extract Method* pois, proporcionalmente, predomina-se os clones do tipo intra-classe. Em comparação com os clones do primeiro caso, a refatoração dos clones do segundo caso requerem a execução de operações mais interna à classe em questão, ou seja, o desenvolvedor precisa examinar e entender uma

quantidade menor de classes para então realizar a operação de refatoração. Na prática, a refatoração dos clones do segundo caso podem ser executadas mais rapidamente e por desenvolvedores que não são experientes, em especial, por aqueles que têm conhecimento limitado dos subsistemas do projeto de software, bem como de suas inter-relações. Contudo, essa conjectura necessita de averiguação através de estudos empíricos que envolvam sujeitos humanos.

4.4 Limitações e Ameaças à Validade

Algumas ameaças devem ser consideradas na análise dos resultados apresentados. A validade interna verifica se as conclusões de uma investigação são corretas para a população de estudo. Por outro lado, a validade externa refere-se à dimensão com que os resultados da pesquisa podem ser generalizados para outras condições ou populações.

Uma das ameaças à validade externa ocorre porque os resultados encontrados não podem ser generalizados para outras linguagens de programação orientada a objetos, isso porque todos os projetos analisados foram desenvolvidos em Java. Para generalizar este estudo, é necessário avaliar projetos de outras linguagens que também utilizam o paradigma de orientação a objetos. Além disso, analisamos apenas projetos *open source* e a maior parte deles (80%) são mantidos pela *Apache Foundation*. No entanto, para estudos exploratórios, a amostra é suficientemente representativa.

Em termos de ameaça interna, as classes analisadas são entidades de produção, ou seja, não consideramos as entidades usadas para testar as classes de produção (ex. classes derivadas da *JUnit*). Isso porque as entidades de teste têm peculiaridades diferentes daquelas usadas em produção (uma análise preliminar dos nossos dados revelou que elas têm clones maiores em termos de LOC). Também consideramos que os desenvolvedores/pesquisadores estão mais preocupados com os *smells* nas classes de produção do que *smells* nas classes de teste, inclusive há estudos dedicados exclusivamente a *smells* que surgem nas classes de teste (138, 139). Portanto, nossa amostra está alinhada aos interesses da comunidade.

Outra ameaça interna é o tamanho da amostra, apenas cinco projetos de software. No entanto, a amostra de três ou mais software é usual em estudos exploratórios. Isso porque este tipo de estudo tem como característica a produção de evidências que suportem os estudos mais abrangentes e extensos.

A detecção dos *smells* é outra ameaça à validade interna. Conforme apresentado na Subseção 2.3.2, a identificação automatizada e/ou semi-automática apresenta um componente de subjetividade e por isso este é um processo impreciso. Inclusive, esse fator ocorre quando os desenvolvedores discutem sobre a existência ou não de determinado *smell* no código (45). Uma forma de minimizar este problema é usar um banco de dados onde diversos desenvolvedores independentes avaliam e votam se determinada entidade

realmente apresenta o *smell* acusado pela ferramenta. A ideia é que quanto mais desenvolvedores concordarem, maior é a chance de estarem corretos. Neste sentido, Palomba et al. (140) criaram o banco de dados *Landfill*[18] com 243 instâncias de cinco tipos de *smells* que foram identificados em 20 projetos de software *open source*. Contudo, este banco de dados não considera os *smells* do estudo detalhado neste capítulo. Assim, devido à limitação de tempo para manualmente avaliar todas as instâncias de *smells* que estudamos e, ainda, devido à subjetividade dos *smells*, confiamos nos resultados da implementação do DECOR que é considerada uma ferramenta estado da arte e, que segundo Moha et al. (4), apresenta até 88% de *Precision* e 100% de *Recall*.

Por fim, outra ameaça à validade interna é que o escopo deste estudo considerou apenas os clones que ocorreram nas classes que sofrem do *smell* Large Class e/ou Complex Class. Entretanto, há clones em entidades que não foram caracterizadas com estes *smells* e estes clones podem se apresentar de forma totalmente diferente destes que analisamos. Como o estudo detalhado neste capítulo visa investigar casos de interação entre *smells* e devido à limitação de tempo/recursos, não seria possível analisar toda e qualquer combinação de *smells* ocorridos com clones. Portanto, a investigação destes outros clones e da possível ocorrência deles com outros *smells* deve ser realizada em trabalhos futuros.

4.5 Considerações Finais

Neste capítulo investigamos a influência de alguns fatores na prevalência de clones, em especial, analisamos como os *smells* Large Class e Complex Class se relacionam com a ocorrência de fragmentos de códigos clonados. Essa investigação considerou diversas formas de interação entre os clones e estes *smells*, a saber: (i) a ocorrência de clones nas classes acometidas apenas pelo *smell* Large Class; (ii) de forma análoga, também consideramos os clones das classes onde o *smell* Complex Class ocorre de forma isolada; (iii) além disso, examinamos a associação da prevalência de clones nas classes onde os *smell* Large Class e Complex Class co-existem simultaneamente.

A literatura apresenta diversos estudos sobre o tema *bad smell*, alguns destes apresentam técnicas usadas para priorizar a refatoração dos *smells* detectados (141, 128), exemplo: a ferramenta inFusion (extensão do iPlasma) fornece aos desenvolvedores um índice de severidade/intensidade do *smell* definido como: *"computed by measuring how many times the value of a chosen metric exceeds a given threshold"* (141). O interesse em estudar a priorização dos *smells* decorre do fato de que os projetos de software geralmente apresentam um grande volume de *smells* (24, 25) e muitas vezes, este volume está distribuído em vários tipos de *smells* distintos (ex. Large Class, Long Method, Data Class). Além disso, ao considerar que há indícios empíricos de que muitos *smells*

[18] <http://www.sesa.unisa.it/landfill/>

interagem entre si (30), a tarefa de refatorar o código que é atribuída ao desenvolvedor se torna dispendiosa e muitas vezes arriscada. Neste contexto e considerando que nosso estudo investiga a associação da prevalência de clones em relação a dois *smells*, investigamos como a intensidade dos *smells* LC e/ou CC influenciam na prevalência de clones.

Em geral, os resultados indicam que a ocorrência de clones está muito mais associada à existência do *smell* Complex Class do que com a existência isolada do *smell* Large Class. Ao considerar a intensidade dos *smell* nas entidades em que existe a co-ocorrência dos *smells* Large Class (LC) e Complex Class (CC), os dados indicam que a prevalência de clones está associada ao mais alto nível de intensidade do *smell* CC. Por outro lado, independentemente da forma com que os *smells* estudados ocorrem, eles explicam apenas uma pequena parte da existência dos clones, ou seja, há outros fatores que ajudam a explicar a incidência de clones nas classes com os *smells* LC e/ou CC. Também observamos que a complexidade da classe e a intensidade dos *smells* de interesse (LC e/ou CC) exercem um papel na forma (clone de uma funcionalidade completa — método — ou apenas um fragmento da funcionalidade), tipo (intra-, inter- e mix-classe), localidade (clones restritos à classes com determinados *smells*) e tamanho (LOC) dos clones.

Capítulo 5

Estudo Empírico: Cronologia de *Smells* e Co-ocorrência

O capítulo anterior demonstrou que, ao menos para alguns dos sistemas analisados, há padrões nos clones entre as entidades que apresentam os *smells* Large Class e/ou Complex Class. Contudo, o estudo apresentado não investiga como e quando esses padrões foram introduzidos no código fonte. Assim, no presente capítulo estudamos a cronologia dos *smells*: Duplicate Code (DC), Large Class (LC) e Complex Class (CC), bem como suas inter-relações. A cronologia de *smells* é caracterizada pelo rastreamento da ancestralidade de uma dada entidade que apresenta algum *smell*. A literatura apresenta diversos estudos relacionados à cronologia de *smells* (142, 143, 144, 145), geralmente referenciados como estudo genealógico de *smells*. Segundo Kim et al. (145) a genealogia de clones investiga a criação, modificação e remoção de clones durante as revisões de um software. A tarefa de traçar a ancestralidade deve ser planejada com cuidado, pois os desenvolvedores podem mover os fragmentos de código de um local para outro ou ainda alterar significativamente o código (ex. incluindo/removendo linhas) ao longo das revisões.

Kim et al. (145), realizou o primeiro estudo sobre genealogias de clones. Suas descobertas indicam que a maioria dos clones são de curta duração e, portanto, a refatoração agressiva pode ser um exagero. Eles também relatam que, muitas vezes, devido às limitações da linguagem de programação, clones longevos são difíceis de refatorar.

Chatzigeorgiou e Manakos (55), estudaram a cronologia de quatro *bad smells* (Long Method, Feature Envy, State Checking, God Class). Os resultados indicam que os *smells* se acumulam à medida que o sistema amadurece, sendo que, na maioria dos casos, os *smells* persistem no código até a última versão examinada. Também relatam que uma vez introduzidos, os *smells* permanecem no código por um grande número de versões. Além disso, uma porcentagem significativa dos *smells*, foi introduzida no momento em que o método/classe em que eles residem foi adicionado ao sistema. Poucos *smells* são removidos e na grande maioria destes casos, o seu desaparecimento não foi o resultado de

refatoração, mas sim um efeito colateral da manutenção do software.

Tufano et al. (78), conduziram um estudo em larga escala (200 sistemas) que investigou quando e porque os *smells* (Blob Class, Class Data Should be Private, Complex Class, Functional Decomposition, Spaghetti Code) são introduzidos. Uma das descobertas indica que muitas vezes as entidades com *smells* já nascem com algum *smell*, reforçando as descobertas de Chatzigeorgiou e Manakos (55). Também descrevem que a implementação de novos recursos no sistema e o aprimoramento daqueles já existentes, são as principais atividades dos desenvolvedores que tendem a introduzir *smells*. Contudo, encontraram quase 400 operações de refatoração que introduziram *smells*. Em um estudo complementar Tufano et al. (146), descobriu que 80% das instâncias dos *smells* analisados continuam a existir no sistema e apenas uma pequena porcentagem (9%) são removidos através de refatoração. Entretanto, não forneceram possíveis motivos para tal descoberta, e ressaltam a necessidade de estudos para entender porque os *smells* não são refatorados pelos desenvolvedores.

A evolução dos *smells* também foi estudada por Olbrich et al. (147). Analisando a evolução dos *smells* God Class e Shotgun Surgery, observaram que há períodos onde o número de *smells* aumenta e períodos em que esse número diminui. Também mostram que o aumento/diminuição do número de instâncias não depende do tamanho do sistema.

Após extensa revisão da literatura (39), não foram encontrados estudos genealógicos da co-ocorrência/inter-relação de *smells*, bem como da análise cronológica conjunta dos *smells*: Duplicate Code (DC), Large Class (LC) e Complex Class (CC). Este tipo de estudo é importante para confirmar e/ou contrapor a sabedoria comum sobre a evolução do software e a manifestação dos *smells* (ex., declínio da qualidade e aumento da complexidade (148)). Do ponto de vista prático, os resultados desse estudo poderiam ajudar a distinguir as diferentes situações que podem ocorrer nos sistemas, em especial, nos casos: i) a introdução de *smells* na concepção das entidades (ex., classes/métodos) do software; ii) os *smells* surgem subitamente à medida que as entidades de software são alteradas ou incrementadas com novas funcionalidades; iii) a pre-existência de determinado *smell* contribui para o surgimento de outro(s) *smell(s)* ao longo do tempo. No primeiro caso, os detectores de *smells* podem ajudar a identificar possíveis problemas, como aqueles relacionados à escolha inadequada do *design* das entidades do sistema que pode exigir um *redesign* para evitar problemas futuros. No segundo caso, as ferramentas de apoio à manutenção do software podem usar essa informação para recomendar quais entidades devem passar por manutenção de emergência, considerando a realização de atividades de refatoração. Por fim, no último caso e considerando que refatorar um dado *smell* pode não ser simples, o surgimento da co-ocorrência deles pode ser um fator usado para apoiar à manutenção do software recomendando certas entidades ao processo de refatoração.

5.1 Planejamento Experimental

O objetivo deste estudo é analisar o histórico de *commits* dos sistemas com o propósito de investigar cronologicamente quando os *smells* são introduzidos e/ou removidos pelos desenvolvedores. Mais especificamente, o estudo visa abordar as seguintes perguntas (RQs):

RQ1 Quando os smells ocorrem nas entidades do software, sejam isolados ou em co-ocorrência? Existe relação temporal entre eles?

Essa questão de pesquisa visa investigar até que ponto podemos considerar a ideia de que os *smells* são introduzidos como consequência das atividades contínuas de manutenção e evolução (15). Para isso, estudamos quando os *smells* ocorrem no sistema, investigando, por exemplo, se o estágio de amadurecimento (ex. momento no tempo) do sistema contribui para uma incidência maior de *smell(s)*. Estudos anteriores (55, 146), investigaram se os *smells* são introduzidos assim que uma entidade é criada ou se são repentinamente introduzidos no contexto de atividades específicas. Entretanto, não consideraram o estágio de amadurecimento em que os *smells* ocorrem. Além disso, não consideraram os *smells*: Duplicate Code, Large Class e Complex Class, bem como a co-ocorrência deles. Os trabalhos anteriores também não investigaram aspectos relacionados à relação temporal dos *smells* e a remoção deles e/ou de suas classes do sistema. Portanto, essa questão de pesquisa visa complementar o conhecimento do estado da arte.

RQ2 Entidades com a simples ocorrência de *smell* podem se tornar entidades com a co-ocorrência deles?

Lehman et al. (149), foram os pioneiros no estudo da evolução de software e formularam um conjunto de observações, das quais podemos destacar: i) um sistema se tornará progressivamente mais complexo, a menos que alguma coisa seja feita para reduzir explicitamente sua complexidade; ii) um sistema perderá sua qualidade ao longo do tempo, a menos que seu *design* seja cuidadosamente mantido e adaptado a novos requisitos operacionais. Esse tipo de observação nos conduz a pensar que se uma entidade apresenta algum *smell* e nada é feito para resolver, com a evolução do sistema, possivelmente o *smell* dessa entidade se torna mais crítico ou o mesmo passa a despenhar um papel de proporcionar o surgimento concomitante de outro(s) *smell(s)*. Portanto, essa questão de pesquisa visa investigar até que ponto podemos considerar a ideia de que um *smell* não removido/resolvido subsidia o surgimento de outro(s).

RQ3 Existe algum *smell* com comportamento cíclico, sejam eles isolados ou em co-ocorrência? Se sim, quais padrões se manifestam?

O comportamento cíclico de *smell* ocorre quando os desenvolvedores removem uma instância de *smell*, mas por algum motivo, ela retorna à classe. Tufano et al. (146), revelaram que grande parte (80%) das instâncias de *smell* continuam a existir até a última versão do código fonte analisado, ou seja, *smells* não são removidos pelos desenvolvedores. Entretanto, não investigaram esse fenômeno considerando a co-ocorrência de *smells*, tão pouco avaliaram o fenômeno considerando os *smells* usados neste estudo. Além disso, o resultado do estudo (146) não permite afirmar que os desenvolvedores não estão interessados em remover *smells*. Isso porque um dado *smell* pode ter um comportamento cíclico. Portanto, essa questão permite identificar se esse comportamento existe, bem como, identificar os padrões que estão entre a remoção e o retorno do *smell*. Assim, essa questão de pesquisa complementa as questões investigadas por estudos anteriores.

5.1.1 Materiais e Métodos

Para responder às questões de pesquisa apresentadas na subseção anterior devemos: i) selecionar um conjunto de repositórios de software, ii) coletar dados dos códigos fonte contidos nos repositórios, iii) viabilizar o processo de análise dos dados. As próximas subseções detalham esses procedimentos

5.1.1.1 Enumeração de Repositórios

Este estudo consiste na análise do histórico de *commits*, ou seja, usaremos os registros de alteração de cada um dos arquivos fonte que compõem um dado sistema. Este tipo de informação é controlada pelo sistema de controle de versão do código fonte (ex.: Git, Subversion) e são mantidas em repositórios públicos e/ou privados.

Um dos maiores repositórios públicos é denominado Github[1], possuindo aproximadamente 31 milhões de usuários e 100 milhões de repositórios. Devido a necessidade de coletar dados em tempo hábil, pesquisadores desenvolveram o projeto BOA, que consiste em uma linguagem de domínio específico acessada por uma infraestrutura de *cloud computing* escalável (150). O projeto BOA, permite efetuar a coleta rápida de dados em *snapshot* do Github que possibilitarão a seleção dos repositórios que usaremos neste estudo. Portanto, usaremos os dados do repositório Github pelos seguintes motivos: i) pela sua dimensão e difusão na comunidade de desenvolvedores; ii) possibilidade de coletar dados preliminares de forma rápida e eficiente. Por outro lado, a própria integração o projeto BOA[2] com o Github já diminui o escopo de busca por repositórios, pois sistemas com determinadas características não são considerados (ex. sistemas que não usam a linguagem Java).

[1] <https://github.com/>
[2] <http://boa.cs.iastate.edu/boa/?q=content/dataset-notes-september-2015>

Uma vez definido onde estão os repositórios (Github) e como coletar informações deles (BOA), o próximo passo é selecionar os repositório alvo deste estudo empírico. Em uma análise preliminar, o BOA retornou do Github um total de 380011 repositórios Java. Como o BOA usa os dados do Github do ano de 2015, provavelmente alguns desses repositórios não estão mais ativos o que impossibilita o download do código fonte. Portanto, o primeiro critério para incluir um repositório no conjunto de dados a ser analisado é que o mesmo esteja ativo no Github (Outubro de 2018).

O segundo critério está relacionado à linguagem usada para desenvolver o sistema, pois existem sistema desenvolvidos em Java e outras linguagens. Assim, para evitar a análise de sistemas com predominância de arquivos fonte de outras linguagens, selecionamos apenas os repositórios que apresentam no mínimo 75% dos arquivos desenvolvidos em Java.

Com o intuito de analisar apenas sistemas ativos, o próximo critério considera o ano em que último *commit* foi realizado. Nesse caso, repositórios onde o último *commit* não foi realizado no ano de 2018 são descartados.

Como o objetivo deste estudo é analisar a evolução histórica dos *smells* e/ou do código fonte, no quarto critério consideramos apenas os repositórios que têm idade mínima de 5 anos e possuem um total mínimo de 500 *commits*.

Por fim, para que nosso estudo seja focado em grandes sistemas, no processo de inclusão de repositórios incluímos mais dois filtros: i) o número de arquivos Java deve ser maior ou igual a 1000 e ii) o sistema deve contar com no mínimo 20 desenvolvedores. A decisão de analisar apenas sistemas grandes advém da ideia de que nesses sistemas a equipe de desenvolvimento é, até certo ponto, bem estruturada e preocupada com a qualidade do código/sistema, portanto a existência de *smells* não deve ser negligenciada.

Ao final dessa etapa foram selecionados 89 repositórios, alguns fazem parte de grandes ecossistemas de software (ex., Hive[3] da Apache, JDT[4] do Eclipse). O conjunto final de dados destes 89 repositórios possui um total de 295164 arquivos Java e 669353 *commits*.

5.1.1.2 Coleta de Dados

Conforme anteriormente apresentado, este estudo visa responder as questões de pesquisa sobre a cronologia dos *smells*: Duplicate Code (DC), Large Class (LC) e Complex Class (CC). A escolha deles se deve ao fato de que são *smells* largamente estudados e difundidos na comunidade, bem como são amplamente e facilmente encontrados no código fonte dos sistemas (39). Aliado a isso, no capítulo anterior, encontramos evidências de que para alguns sistemas estes *smells* apresentam certo grau de associação. Portanto, o estudo da cronologia deles se faz necessária para compreender o fenômeno.

Ao estudar a cronologia de *smells* devemos identificar se e quando um dado *smell* é introduzido na classe do sistema, bem como por quanto tempo ele permaneceu nessa

[3] <https://github.com/apache/hive>
[4] <https://github.com/eclipse/eclipse.jdt.debug>

entidade. Contudo, conforme apresentado em nosso estudo (39), há poucas ferramentas de detecção de *smells* capazes de detectar *smells* por entre os diversos *commints* que compõem os sistemas, e destas, não encontramos uma capaz de fazer isso para os *smells* de interesse deste estudo. Além disso, muitas dessas ferramentas são estritamente acadêmicas onde o seu código fonte não é disponibilizado à comunidade. Assim, para o estudo deste capítulo, decidimos construir nossa própria ferramenta.

Para compreender o funcionamento da ferramenta projetada é necessário conhecer certos elementos do sistema de controle de versão do código fonte (ex., repositório Git). Ao descarregar uma cópia do código fonte de um dado sistema armazenado nesses repositórios, além dos arquivos de código fonte, também descarregamos uma estrutura que mantém informações de alteração/criação/remoção de arquivos deste sistema. Essa estrutura é organizada em unidades conhecidas como *commits*, que podem ser vistas como o registro de tudo que é feito no sistema em um dado instante no tempo. Então, a versão atual do código fonte é basicamente o resultado acumulado de todas as alterações realizadas pelos desenvolvedores. Partindo desse princípio podemos criar uma organização cronológica dos registros de *commits* que, a partir da versão atual, pode ser percorrida em ordem inversa para rastrear e analisar qualquer alteração do código fonte. Isso possibilita coletar informações dos *smells* de cada classe do sistema por entre as diversas alterações do código fonte. Entretanto, muitas vezes os repositórios são organizados em ramificações (*branches*). Isso significa que a organização cronológica dos *commits* não é necessariamente linear, ou seja, alterações do código fonte podem ocorrer em paralelo e quando elas divergem de alguma forma (ex., objetivo) da linha de desenvolvimento principal (*master*) uma ramificação é criada, mas essa em algum momento retorna ao curso da linha de desenvolvimento principal. Um exemplo desse tipo de situação ocorre quando, uma alteração longa e complexa deve ser feita no código (ex., inclusão de novas funcionalidades), sendo que ao mesmo tempo outras de curta duração devem ocorrer e serem entregues aos clientes (ex. correção de erros). A Figura 19 representa essa organização dos *commits*, neste caso, cada círculo representa um dado *commit* ocorrido no instante N de seu fluxo de desenvolvimento (*master* ou *branch*).

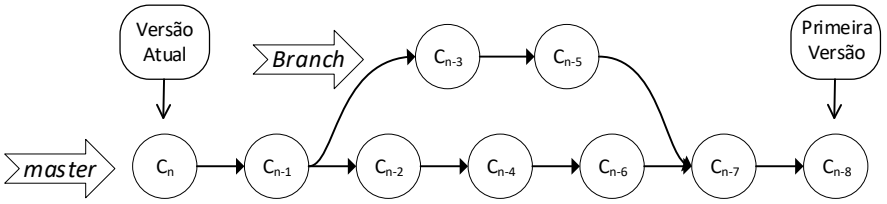


Figura 19 – Diagrama de *commits* do código fonte.

Nossa ferramenta descarrega uma cópia do código fonte hospedado em um repositório do Github e organiza cronologicamente os registros de *commits*, sendo esses analisados em ordem inversa (do presente para o passado) para coletar dados relacionados aos *smells*

de interesse deste estudo. Para evitar a coleta de informação de alguma ramificação irrelevante (ex., *test branch*) e também para otimizar o processo computacional, nossa ferramente identifica a ramificação principal e percorre apenas os *commits* dela. Ainda em relação aos *commits*, observamos a existência de sistemas que apresentam um elevado número de *commits* e a análise de cada um deles pode ser, em termos computacionais, um fator limitante em estudos com um grande número de repositórios/*commits*. Neste caso, nossa ferramenta permite a análise usando o conceito de janela de *commits* deslizante, em que o usuário informa um coeficiente que indica quantos *commits* devem ser descartados até que o próximo seja analisado. Para coletar os dados deste estudo usamos como coeficiente um valor que representa 1% dos *commits* do sistema que está em análise, isso porque o numero de *commits* dos sistemas analisados são razoavelmente diferentes (*Mean: 7521, StDev: 6625, Min: 1085, Med: 6152 Max: 41435*). Desse modo, será possível efetuar determinadas observações entre sistemas que tenham a quantidade de *commits* diferente (ex., um dado *smell* é mais propenso de ser introduzido no início do sistema, ou seja, são incluídos na janela de *commit* que representa, por exemplo, os primeiros 10% dos *commits*). O funcionamento da janela de *commits* é apresentado na Figura 20, observe que temos um total de 100 janelas, o que permite analisar os dados usando o conceito percentis (1) e para cada instante no tempo k, as alterações do código fonte são revertidas conforme o primeiro *commit* da janela$_k$.

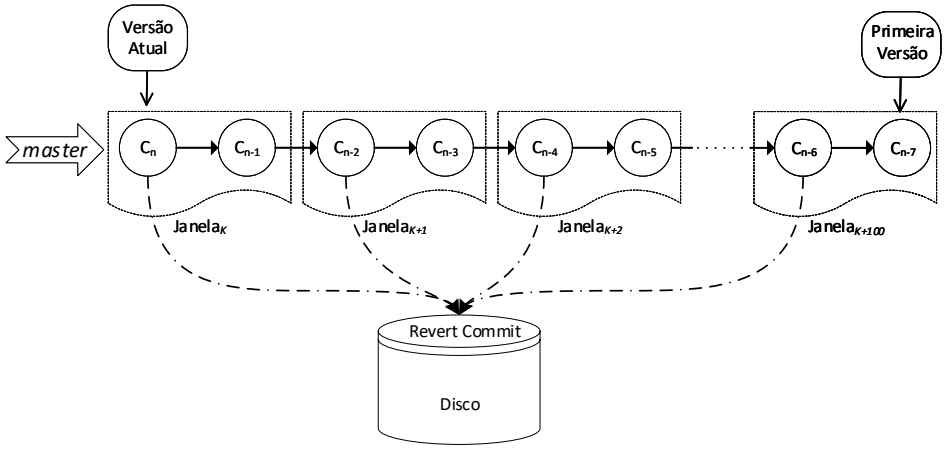


Figura 20 – Diagrama de funcionamento da janela de *commits*.

O processo de análise dos *commits* implementado em nossa ferramenta, consiste em reverter todas as alterações do código fonte para o estado/momento em que o n-ésimo *commit* foi criado. Na sequência, cada uma das classes que compõem o sistema que está sendo analisado passa por um exame que verifica a existência ou não dos *smells* de interesse (diagnóstico), no caso da ocorrência simultânea de algum deles, uma co-ocorrência é registrada. De forma análoga, um registro também é criado ao identificar a ocorrência isolada de um dado *smell*. A implementação da etapa de diagnóstico dos *smells* Duplicate Code, Large Class e Complex Class, segue rigidamente a implementação

usada na ferramenta PMD[5] e no utilitário DECOR[6] (ver Subseção 2.3). Para este estudo, também configuramos os valores dos parâmetros de detecção igual aqueles usados e detalhados no capítulo anterior. Escolhemos usar a implementação dessas ferramentas porque elas são de código fonte aberto, o que possibilita a completa integração do seu código à ferramenta que projetamos. Outro aspecto que motivou essa escolha é o seu amplo uso na comunidade de desenvolvedores e pesquisadores, bem como para manter a consistência dos dados, pois a implementação original foi usada no estudo descrito no capítulo anterior. Nessa etapa é importante destacar que no diagnóstico as classes de teste são descartadas, ou seja, aquelas criadas para testar as classes de produção não são analisadas. Isso porque os *smells* existentes nas classes de teste são diferentes e têm características próprias/específicas (139, 138). Em termos de implementação, classes de teste são aquelas que têm *imports* originados de sistemas teste (ex., *junit*) e que satisfazem alguma das seguintes condições: i) a classe que está sendo examinada possui algum parentesco (ex., herança direta ou indireta) com classes de sistemas teste (ex., *junit*); ii) ou apresenta alguma entidade (ex., método) com *annotation(s)* específicas de sistemas teste (ex., *@Before, @Test, @After*).

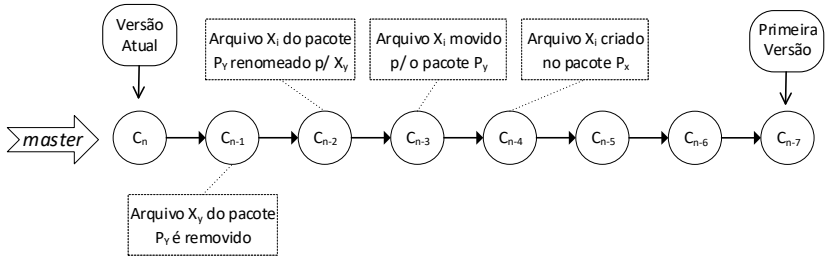


Figura 21 – Exemplo de operações de refatoração ocorridas em um dado arquivo.

O estudo da cronologia de *smells* exige o mapeamento das classes do sistema por entre os diversos *commits*, então o processo de coletar dados não pode se restringir apenas à análise da existência dos *smells* de interesse. Em nossa implementação e para este estudo, toda classe do sistema pertence e existe em um ou mais *commit(s)*, tal que estes estão organizados em k janelas onde k varia de 1 a 100 (ver Figura 20). Entretanto, para maior assertividade do mapeamento entre uma dada classe e os *commits* em que ela existe é necessário considerar certas operações de refatoração possivelmente executadas pelos desenvolvedores. Considere a Figura 21, e imagine que cada um dos *commits*, bem como seus conteúdos são analisados de forma individual e independente. Nessa situação, ao analisar o código fonte resultante em cada um dos *commits* $C_{n-1,\ldots,n-4}$ podemos concluir que existem quatro arquivos independentes. Entretanto, se analisarmos esses mesmos *commits* considerando o que foi feito na vizinhança de cada um deles, podemos concluir

[5] <https://pmd.github.io/>
[6] *DEtection & CORrection* — <https://bitbucket.org/ptidejteam/>

que existe apenas um arquivo que sofreu diversas operações de refatoração.

Algoritmo 1: Coletando dados sobre a cronologia de *smells*.

```
1  DataCollect (url, coefficient ← 1%)
       input  : url of repository.
       input  : coefficient is a value (0%...100%) to get the size of commit window.
       output: set of all classes[7], each one then mapped to a set of commit window.
2      SourceCode ← DownloadRepository(url);
3      RawCommits ← GetCommits(SourceCode);
4      SortedCommits ← GetCommitsOnMasterAndSort(RawCommits);
5      WindowSize ← GetWindowSize(SortedCommits, coefficient);
6      AllClasses ← ∅; PrevCommit ← ∅; k ← 0;
7      while (commit_n ← NextCommit(WindowSize, SortedCommits)) ≠ ∅ do
8          Refactorings ← GetMovesRenames(commit_n, PrevCommit) ;
            // Tsantalis implementation - RefactoringMiner
9          RevertedSource ← RevertToCommit(commit_n);
10         PrevCommit ← commit_n; k ← k + 1;
11         foreach SourceCode class_j ∈ RevertedSource do
12             if not isProductionClass(class_j) then
13                 continue;
14             class_j.PrevObj ← GetObjRefactored(class_j, Refactorings);
15             class_j.LC ← isLargeClass(class_j) ;     // DECOR's implementation
16             class_j.CC ← isComplexClass(class_j) ; // DECOR's implementation
17             class_j.DC ← HasDuplicateCode(class_j) ; // PMD's implementation
18             if class_j.PrevObj ≠ ∅ then
19                 update the class_j on AllClasses_list;
20             else
21                 add the class_j on AllClasses_list;
22             Map the class_j to the commit window k on AllClasses_list;
23     return AllClasses;
```

Tsantalis et al. (151), desenvolveram uma ferramenta denominada *RefactoringMiner*[8] que tem a capacidade de rastrear e identificar diversas operações de refatoração (ex., arquivos movidos e/ou renomeados). *RefactoringMiner* demonstrou 98% de *precision* e 87% de *recall*, no estudo empírico ela se demonstrou significativamente melhor do que as demais ferramentas já existentes. Por esse motivo ela é considerada uma ferramenta de estado da arte, sendo assim usaremos o código fonte dela para rastrear as operações

[7] It means a set that includes the classes that currently exist on the project and/or those that was removed by some commit.

[8] <https://github.com/tsantalis/RefactoringMiner>

de refatoração ocorridas no sistema que está sendo analisado. Assim, ao considerar o caso apresentado na Figura 21, nossa ferramenta permite identificar se o arquivo X_y do *commit* C_{n-1} é o mesmo arquivo denominado X_i existente no *commit* C_{n-4}.

O Algoritmo 1, resume os passos detalhados nessa subseção e usados na coleta de dados do estudo da cronologia de *smells*. Em termos de execução, a coleta de dados executada por esse algoritmo para os 89 sistemas selecionados na Subseção 5.1.1.1 levou cerca de 2,5 semanas em um servidor Linux que tem 15 núcleos de 2,40 GHz e 28 Gb de memória RAM.

5.1.1.3 Detalhamento e Representação dos Dados

Na subseção anterior, descrevemos em detalhes o procedimento efetuado para coletar os dados do estudo da cronologia de *smells*, em especial relatamos como coletar os dados e quais informações foram reunidas. Contudo, não especificamos estruturalmente a organização desse conjunto de informações. Assim, essa subseção irá detalhar e exemplificar esta organização.

Estruturalmente os dados estão distribuídos em k janelas de *commits*, onde a cada janela corresponde a um *snapshot* do sistema referente momento do primeiro *commit* da janela em questão (k_0 é a versão atual do sistema e k_{99} é a primeira versão do sistema), e cada janela contém um conjunto de classes do sistema, sendo que uma dada classe j pertencente a uma dada janela k que apresenta um estado relacionado a existência dos *smells* de interesse ou a própria existência. Em outras palavras, e usando um exemplo hipotético, a $\text{Classe}_{j_{10},k_0}$ na presente versão k_0 do código fonte apresenta a existência apenas do *smell* Complex Class e na versão anterior ($\text{Classe}_{j_{10},k_1}$) ela pode nem mesmo existir no código fonte desse sistema. Portanto, para este estudo, podemos enumerar os seguinte estados das classes:

- **Sem *smell* (SS)**: Essa certamente é a situação ideal pois, possivelmente, indica boa saúde da classe (*smells* são usados como indicadores da qualidade do código (152)). Nesse caso, as ferramentas de detecção de *smells* não identificaram nenhuma instância deles na classe em questão.

- **Classe não encontrada (NE)**: Este caso emerge quando consideramos a longevidade dos arquivos que compõem um dado sistema. Como apresentado (Figura 21), eles não são eternos/imutáveis e podem sofrer diversos tipos de alterações. O presente estado ocorre quando uma dada classe existe em uma janela de *commit* mas não existe em outra. Portanto, essa situação gera esse estado específico.

- **Apenas Large Class (LC)**: A implementação da ferramenta DECOR identificou em uma determinada classe, de uma certa janela, a existência exclusiva do *smell* Large Class. Note que a exclusividade é fruto da checagem da presença simul-

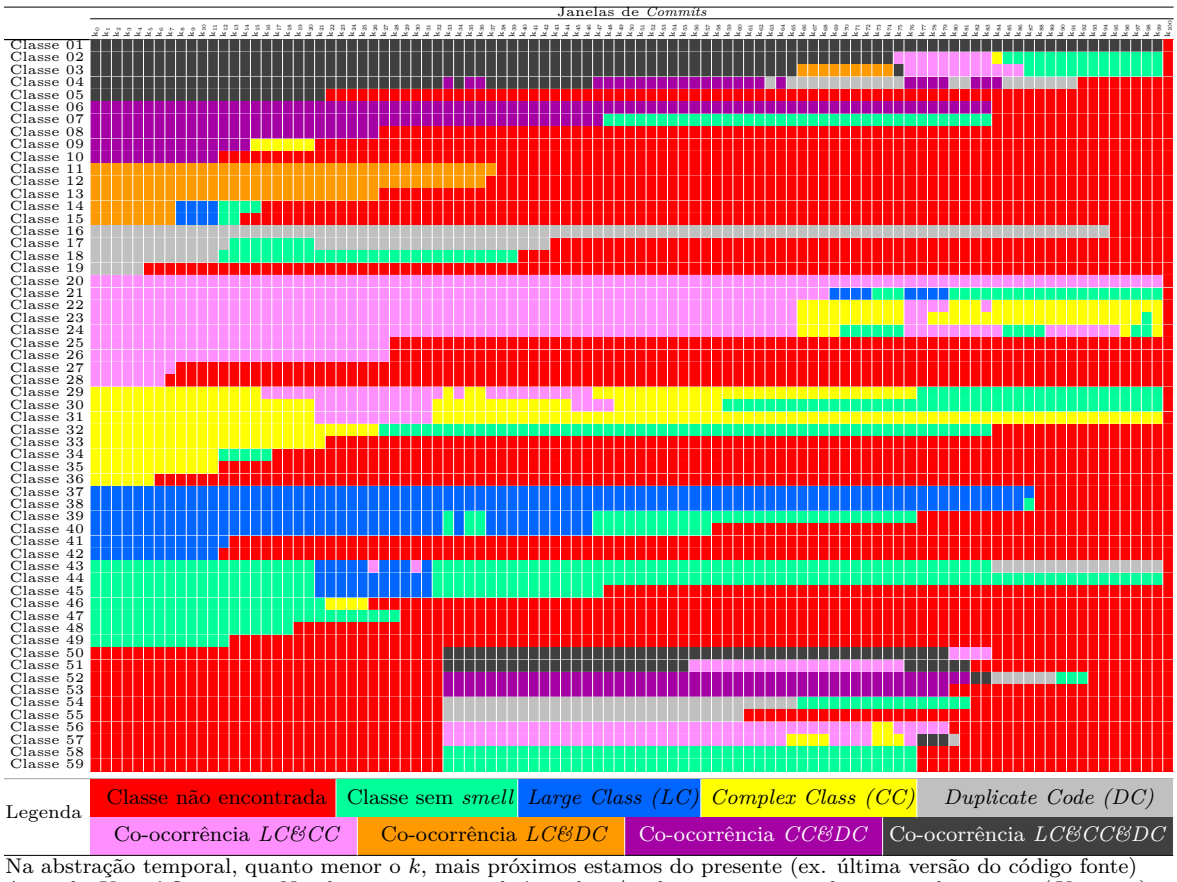
tânea de pelo menos um dos outros dois *smells* de interesse deste estudo, então a classe pode conter outro *smell* que não estamos estudando.

- ❑ **Apenas Complex Class (CC)**: Assim como no estado anterior, esse também se refere à existência exclusiva do *smell* Complex Class.
- ❑ **Apenas Duplicate Code (DC)**: Análogo aos últimos dois estados, mas se trata do *smell* Duplicate Code.
- ❑ **Co-ocorrência LC&CC**: Este estado denota a existência concomitante dos *smells* Large Class e Complex Class.
- ❑ **Co-ocorrência LC&DC**: Aqui se registra a ocorrência concomitante dos *smells* Large Class e Duplicate Code.
- ❑ **Co-ocorrência CC&DC**: De forma semelhante aos dois anteriores, nesse estado temos a existência concomitante dos *smells* Complex Class e Duplicate Code.
- ❑ **Co-ocorrência LC&CC&DC**: Este estado denota a ocorrência concomitante dos três *smells*, a saber: Large Class, Complex Class e Duplicate Code.

Uma vez estabelecido nove possíveis estados das classes que compõem um sistema, podemos usar o conceito de janelas de *commits* para estabelecer uma representação/organização dos dados que nortearão o estudo de *smells* na sua forma cronológica. Isso pode ser feito porque cada janela de *commit* pertence a um dado instante no tempo (Figura 20), em particular aquele conferido ao *commit* usado para reverter as alterações realizadas no código fonte. Partindo desta ideia, criamos um modelo visual de representação que considera as classes, janelas de *commits* e seus respectivos estados. Este modelo permite observar cronologicamente cada um de seus elementos, bem como auxiliará na análise do grande volume de dados produzido pelos 89 sistemas e ele está representado na Figura 22. Nessa representação[9], cada linha é uma classe do sistema e as k-ésimas colunas representam as janelas de *commits*, sendo que a primeira (mais a esquerda) é a versão atual do código fonte e a última (mais a direita) é a primeira versão desse mesmo sistema que coincide com o primeiro *commit*. Observe que atribuímos cores distintas para cada um dos nove possíveis estados das classes: Vermelho para o estado de classe não encontrada; Verde quando a classe não tem *smell*; Azul para representar a simples ocorrência do *smell* Large Class; Amarelo para o *smell* Complex Class; Cinza para Duplicate Code; Rosa na co-ocorrência de Large Class & Complex Class; Laranjado quando existe os *smells* Large Class & Duplicate Code; Roxo na simultaneidade dos *smells* Complex Class & Duplicate Code e finalmente Preto na tríplice ocorrência de todos os *smells*.

[9] Note que neste gráfico o tempo aumenta da direita para esquerda

Figura 22 – Fragmento do estudo da cronologia de *smells* — Software Activemq.



Na abstração temporal, quanto menor o k, mais próximos estamos do presente (ex. última versão do código fonte)
A janela K_{100} é figurativa. Na abstração temporal, é usada p/ indicar a transição de criação do sistema ($K_{99...100}$).

Pela inspeção visual dos dados representados na Figura 22, a análise da vizinhança temporal de uma dada célula (ponto de intersecção entre uma classe e uma janela), fornece uma nova perspectiva que pode ser traduzida pelos conceitos de transição ou estabilidade de estado(s). Transições ocorrem quando o estado de uma dada classe se altera entre duas ou mais janelas de *commits* que são adjacentes no tempo. Por outro lado, a estabilidade de estado é essencialmente o caso oposto, ou seja, o estado não se altera entre uma janela de *commits* e outra adjacente. Nos próximos parágrafos vamos examinar algumas destas situações extraídas da análise realizada no sistema Activemq[10] e que está parcialmente representada na Figura 22.

Verificando a Figura 22, observamos algumas classes com estabilidade absoluta de estado, ou seja, uma dada classe nasce em um estado e permanece nele até o fim (ex. $\text{Classe}_{01,\ 06,\ ...,\ 59}$). Neste fragmento, observe que todos os nove estados (DC, LG, CC, ...) têm alguma classe que segue esse comportamento. Além disso, temos casos em que os desenvolvedores removeram[11] classes com esse comportamento (ex. $\text{Classe}_{53,\ 59}$). Também temos alguns casos de estabilidade absoluta tão longevas quanto o próprio sistema (ex.

[10] <https://github.com/apache/activemq>

[11] Uma classe removida do sistema, implica que as janelas de *commits* subsequentes e mais próximas da versão atual do sistema estarão sinalizadas de vermelho, representado o estado de classe não encontrada.

Classe_{20}), ou seja, a classe nasceu com o sistema em um dado estado e permanece nele até os dias atuais no mesmo estado. Também se pode observar estabilidade relativa, ou seja, para um determinado subintervalo das janelas de *commits* não há variação de estado (ex. Classe_{07} entre as janelas $k_{48...83}$). A estabilidade relativa implica na existência de uma ou mais transições, contudo elas ocorrem em pontos diferentes daqueles observados na estabilidade relativa, exemplo: a Classe_{07} tem uma transição entre as janelas $k_{47...48}$ e $k_{83...84}$. Em termos de transição, também podemos observar algumas situações em que um dado estado é cíclico. A Classe_{45} é um destes casos, até a janela k_{20} seu estado era sem *smell*, entre as janelas $k_{21...30}$ ela alterou seu estado para Large Class e finalmente voltou ao estado sem *smell*.

Pelas situações apresentadas nos parágrafos anteriores, é possível imaginar diversas questões de pesquisa que podem ser respondidas pela análise de padrões dos estados. Contudo, neste estudo, restringimos nossas análises apenas naqueles padrões necessários para responder às questões de pesquisa apresentas no início desta Subseção (5.1). Contudo, apresentamos a Figura 23, que detalha visualmente cada um dos 89 sistemas selecionados na Subseção 5.1.1.1. Nela é possível obter uma visão geral de todos os padrões encontrados no código fonte dos sistemas.

Analisando a representação visual dos dados exibida na Figura 23, podemos destacar três grandes grupos: i) Classes com *smell(s)* (cores: Azul, Amarelo, Rosa, Cinza, Laranjado, Roxo e Preto); ii) Classes sem *smell(s)* (cor Verde); iii) Classes removidas do sistema (cor Vermelha próximo à última versão do código). No primeiro grupo estão as entidades em que as ferramentas de detecção apontaram a existência de algum *smell* em alguma janela e a entidade em questão ainda existe na versão atual do sistema (a maior parte delas estão representadas na parte superior de cada recorte da figura Figura 23). O próximo grupo é formado pelas classes que não apresentaram *smell(s)* na k-ésima janela e também ainda permanecem na versão atual do código fonte. Por fim, no último grupo temos aquelas classes que já não fazem mais parte da última versão do código, ou seja, aquelas que foram removidas. Esse grupo é formado tanto por classes com e/ou sem *smell(s)*, mas que foram removidas do sistema. Elas são identificadas na Figura 23, pelo fato de terem sua representação de ausência marcadas pela cor vermelha nas janelas de *commits* que estão nas últimas versões do sistema, isto é, a(s) coluna(s) mais a esquerda são vermelhas. A Figura 24, apresenta a evolução do comportamento médio (número de classes) daqueles três grupos. Ela foi concebida da seguinte forma: para cada sistema e em cada janela de *commit*, contamos a quantidade de classes pertencentes a cada um dos grandes grupos; por fim, obtemos seus valores médios. Destacamos que a reta usada para representar o grupo das classes removidas, descreve o número médio de classes que já foram removidas até a k-ésima janela de *commit*, ou seja, ela é uma representação acumulada do número de classes removidas.

A Figura 24, detalha quatro linhas de tendência que têm comportamento linear e por

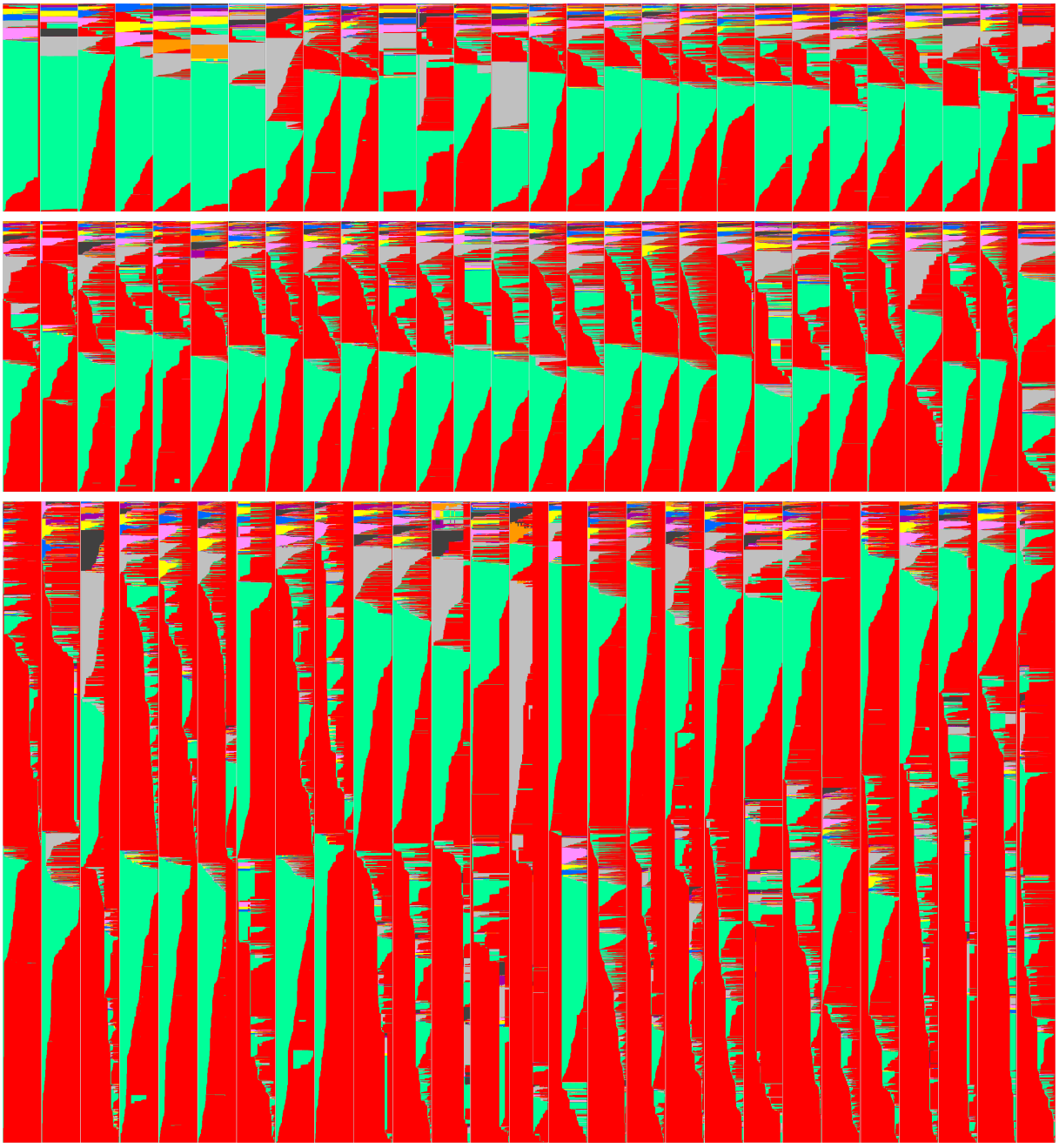


Figura 23 – Dados dos sistemas: a) parte superior — sistemas do 1 a 29, b) parte do meio — sistemas do 30 a 59 e c) parte inferior — sistemas do 60 a 89.

isso também apresentamos os valores dos coeficientes de regressão dessas linhas. A reta com menor coeficiente angular/linear é aquela pertencente ao grupo de classes com algum *smell*. Isso indica que, em média, a quantidade de classes com algum *smell* é menor do que a quantidade de classes pertencentes aos outros grupos. Também notamos que a quantidade média de classes que já foram removidas dos sistemas é maior do que a quantidade de classes que apresenta algum *smell*, isso para grande parte das janelas de *commits*, pois no início do sistema ($Janelas_{87...99}$) esse comportamento é oposto/antagônico. Isso demonstra que atividades de remoção no nível de classe são frequentemente executadas pelos desenvolvedores. Por fim, e na sequência de magnitude temos a linha

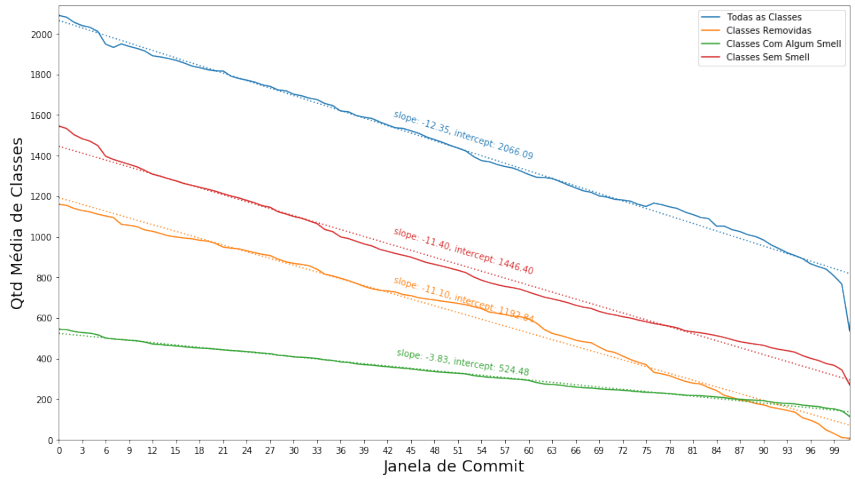


Figura 24 – Evolução histórica, agrupada pelos principais estados das classes dos sistemas.

de tendência do grupo das classes sem *smell*, ela demonstra que os sistemas sempre têm em termos numéricos médio um maior número de classes saudáveis (sem *smell*). A linha azul da Figura 24, mostra o número médio do total de classes existentes sem realizar qualquer distinção entre os três grandes grupos. Nota-se que, os sistemas começam com um número relativamente pequeno de classes ($\approx$ 600), na sequência experimentam um crescimento súbito ($\approx$ 800) e finalmente entram em um regime de crescimento linear. Em geral, pela Figura 24 observamos que em relação ao total de classes, a proporção de classes com algum *smell* é maior no início do sistema. Em outras palavras, à medida que o sistema cresce (número de classes), a proporção de classes afetadas por algum *smell* também cresce mas em uma taxa menor. Isso implica que, de certa forma, à medida que o sistema cresce, os desenvolvedores tendem a controlar a proliferação de *smells*.

5.2 Resultados e Discussão

Essa subseção apresenta os resultados para cada questão de pesquisa apresentada na Subseção 5.1.

RQ1 — Quando os smells ocorrem nas entidades do software, sejam isolados ou em co-ocorrência? Existe relação temporal entre eles?

Essa questão de pesquisa pode ser respondida e investigada tendo em vista diferentes perspectivas: i) em relação a quando os estados de uma classe ocorrem em cada janela de *commits*; ii) em relação a quando as transições entre estados ocorrem nas classes; iii) em relação ao momento em que os estados de *smells* surgem pela primeira vez. Assim, para cada uma dessas perspectivas, elaboramos uma subquestão de pesquisa (RQ1.1; RQ1.2; RQ1.3). Essas serão examinadas nas próximas subseções.

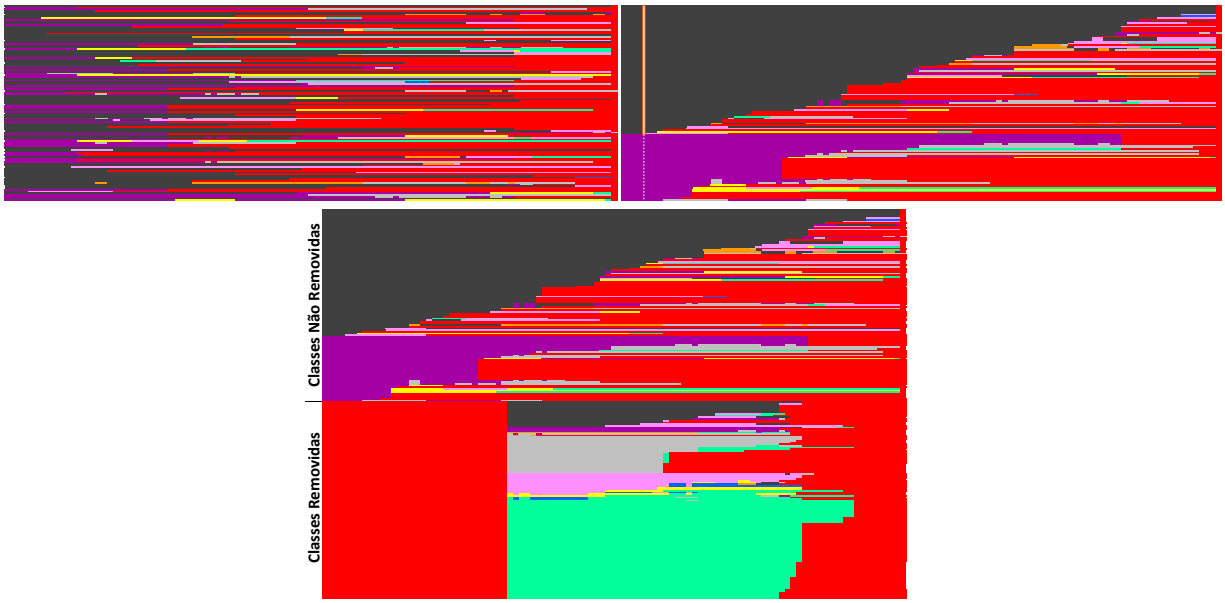


Figura 25 – Estados das classes em cada janela de *commit*: a) superior esquerdo — sem ordenar, b) superior direito — ordenado conforme estados, c) inferior — ordenado por estados e agrupado pela atual existência ou não da classe.

RQ1.1 Quando os estados de uma classe ocorrem ao longo do tempo de vida da classe?

Nessa questão vamos analisar diversas relações: a) se existe alguma relação entre um dado estado (ex. Large Class) e o momento no tempo (Janela de *commit*) em que ele se encontra. Seja um exemplo hipotético: o estado Large Class ocorre de forma contínua e acumulativa ao longo do ciclo de vida dos sistemas, ou seja, o número de classes com o *smell* Large Class cresce com o tempo. b) se existe alguma relação temporal entre dois estados distintos. Seja um exemplo hipotético: o estado Large Class & Duplicate Code, que é um dos tipos de co-ocorrência, tem seu coeficiente angular de regressão similar ao apresentado no estado Complex Class & Duplicate Code.

Para examinar essas relações, devemos quantificar o número de classes em cada janela de *commit*, agrupadas pelos seus respectivos estados. Portanto, o primeiro passo é ordenar os estados conforme o seu tipo, a Figura 25a-b representa esse procedimento. Observe que na primeira figura, as classes (cada linha) não estão ordenadas conforme o estado das classes vizinhas de uma dada janela de *commit*, isso porque ao percorrer os *commits* para coletar os dados não há como saber apriori a posição da classe em relação as demais classes da mesma janela de *commit*. Assim, se faz necessário ordenar os dados (estados) das classes, como demonstrado na Figura 25b. Na sequência, para cada janela de *commit*, contabilizamos a quantidade de classes agrupando por estado: na Figura 25b, a janela k_3 têm 76 classes no estado LC&CC&DC (cor preta) e 39 classes no estado CC&DC (cor roxa). Então, para cada sistema analisado, executa-se esse procedimento para obter uma matriz que descreve a quantidade de classes existente no n-ésimo estado de cada janela

de *commit*. Por fim e a partir dessa matriz, extraímos seus valores médios para produzir os gráficos de linha representados na Figura 26.

Nos gráficos de linha representados na Figura 26, para cada estado, temos duas linhas: uma descreve o comportamento de um dado estado (ex. LARGE CLASS) para as classes que ainda existem na versão atual do código fonte e outra para detalhar a conduta desse mesmo estado nas classes que foram removidas do sistema. Esse novo agrupamento é feito tendo como base o estado atribuído a uma dada classe na versão atual do código (janela de *commit* mais a esquerda). A Figura 25c demonstra esses dois grupos, no primeiro o estado de todas as classes na versão atual do código (janela K_0) é diferente de *"Classe não encontrada"* e portanto, nessa janela, elas estão marcadas com uma cor que não seja vermelha. Por outro lado, o segundo grupo é justamente a situação oposta e por isso a cor assinalada na janela k_0 é justamente a cor vermelha.

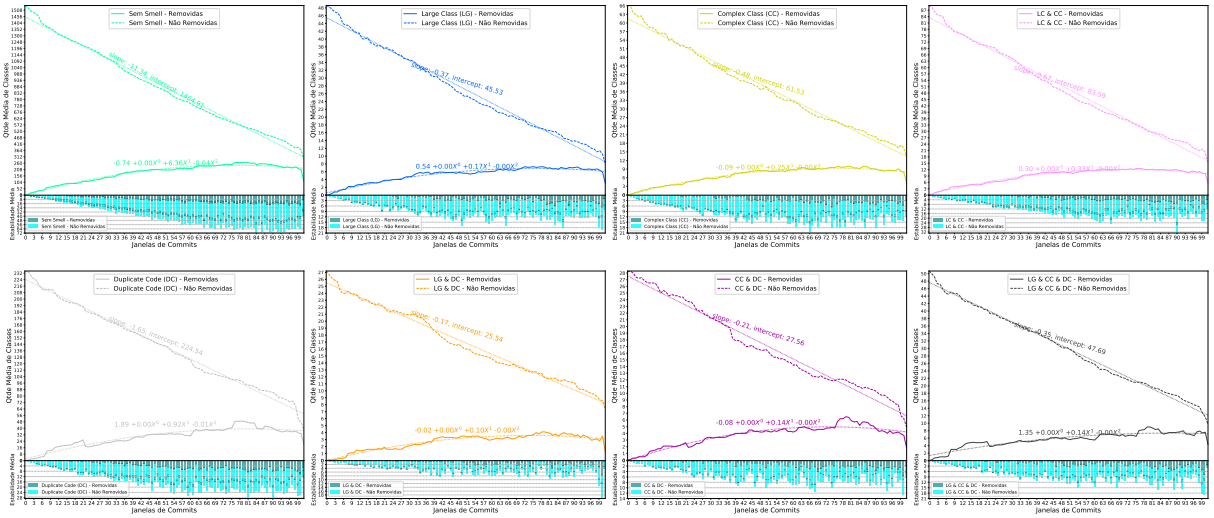


Figura 26 – Evolução histórica de cada estado e estabilidade média dos estados, desmembrado conforme classes removidas e não removidas dos sistemas.

Na Figura 26, além dos gráficos de linhas também temos gráficos de barra. Esses demonstram a estabilidade média de cada estado conforme a ocorrência de transição de estados em uma dada janela de *commit*. Nesse caso, a estabilidade é uma métrica obtida a partir do número médio de janelas de *commits* que um dado estado se manteve sem efetuar uma transição para outro estado. Para compreender, considere duas classes (A e B) que existem no sistema em 9 janelas de *commits* consecutivas ($K_{0..8}$) e para cada uma delas, as classes apresentam determinado estado (destacados por cores), veja: ■■■■■■■■■■ (Classe A) e ■■■■■■■■■■ (Classe B). Uma estabilidade ocorre quando não há diferença de estado entre duas janelas de *commits* consecutivas, exemplo: há uma estabilidade de estado nas três primeiras janelas ($K_{0...2}$) da Classe A, pois o estado da quarta janela não é o mesmo estado na janela anterior, ou seja, temos uma transição de estado entre as janelas três e quatro. Ao considerar as diversas janelas de *commits* e a estabilidade de um dado estado, podemos calcular a estabilidade média deles em relação à

k-ésima janela de *commit*, veja: considerando a janela de *commit* K_2 de ambas as classes e o estado ■, a estabilidade média é de 2,5 janelas, pois para a Classe A a estabilidade desse estado para essa mesma janela de *commit* é 3 e na Classe B o cálculo da estabilidade com os mesmo parâmetros é 2, portanto a estabilidade média é 2,5. O Algoritmo 2 detalha o procedimento para calcular a estabilidade média. Observe que pelo algoritmo é possível efetuar o cálculo de estabilidade média tanto para o grupo de classes removidas, quanto para aquelas que ainda residem na versão atual do código fonte. Assim, na Figura 26, os gráficos de barra detalham estas informações dos dois grupos separadamente (*Removidas* e *Não Removidas*).

Algoritmo 2: Coletando dados sobre a estabilidade dos estados conforme ocorrência de transições nas k-ésimas janelas de *commit*.

```
1  AvgStability (Project, AllDS, cType ← [Removed, NotRemoved])
       input  : Project is the ID of a software project.
       input  : AllDS is the dataset collected (Projects, Classes and Status) and
                sorted according to the projects and the status of each class.
       input  : cType is a filter based on the class existence on the last version of
                the source code.
       output: Average stability of each status according to their commit window.
2      DataSet ← GetDataSetOfProject(AllDS, Project);
3      DataSet ← FilterClassType(DataSet, cType);
4      SetResult ← HashMap<Status,HashMap<Window,int[]{0,0}>>();
5      AvgResult ← HashMap<Status,HashMap<Window,int>>();
6      foreach class ∈ DataSet do
7          k ← 0; Prev_k ← k;                  // The last version of source code is in commit window 0
8          while k < GetNumberOfWindows(class) − 1 do
9              StatusK ← GetStatus(class, k) ;                          // LC, CC, DC, ...
10             StatusNext ← GetStatus(class, k + 1);
11             if StatusK ≠ StatusNext then
12                 SetResult_[StatusK,k,0] ← SetResult_[StatusK,k,0] + 1;
13                 SetResult_[StatusK,k,1] ← SetResult_[StatusK,k,1] + (k − Prev_k) + 1;
14                 Prev_k ← k + 1;
15             k ← k + 1;                                  // Get the previous commit window
16     foreach status ∈ SetResult do
17         foreach window ∈ status do
18             AvgResult_[status,window] ← SetResult_[status,window,1] ÷ SetResult_[status,window,0];
19     return AvgResult;
```

❑ Análise e discussão dos gráficos representados na Figura 26.

Considerando as definições anteriores, iremos examinar a existência de relação entre um dado estado e o momento no tempo (janela de *commit*) em que ele ocorre. Para isso, primeiro vamos considerar os dados das classes pertencentes ao grupo

"Não Removidas"[12], observamos uma relação linear entre o tempo e a quantidade de classes em cada estado, que cresce à medida que nos aproximamos da janela de *commit* que representa a última versão do código fonte. Esse comportamento é nítido e se repete para todos os estados analisados neste estudo.

Em termos gerais de *"quando"* ocorrem, o fato acima indica que todos os *smells*, bem como a co-ocorrência deles, são continuamente introduzidos no código fonte, seja por displicência ou por falta de controle/conhecimento dos desenvolvedores. Isso implica que sistemas com pouco tempo de vida têm numericamente menos incidência de *smell(s)* do que aqueles sistemas mais longevos. Para constatar essa afirmação, considere o gráfico de linha do estado Large Class na Figura 26, em que usamos as marcações do eixo x (janelas de *commits*) como *proxy* para a longevidade do sistema. Considerando que todos os sistemas analisados em nosso estudo são longevos, se considerar apenas as 50 primeiras janelas de *commits* ($K_{50...100}$), teremos um período que denota um sistema mais jovem. Nesse caso, observamos um total médio de 24 classes com o *smell* Large Class. Quando consideramos um período que reflete um sistema com longevidade maior ($K_{0...100}$), esse valor cresce linearmente e praticamente dobra. Portanto, em pesquisas científicas de *smell(s)*, é importante considerar a longevidade dos sistemas usado no conjunto de dados.

Olbrich, Cruzes e Sjoberg (51), estudaram se os *smells* God Class e Brain Class têm efeito negativo na frequência com que os códigos são alterados e se eles influenciam no tamanho das alterações realizadas pelos desenvolvedores. Seus resultados indicam que, esses *smells* têm um impacto negativo nesses fatores, mas ao normalizar os dados pelo tamanho do sistema se observou o contrário, ou seja, classes com esses *smells* foram menos alteradas do que as outras classes. Com base nesse tipo de resultado e pautado no comportamento apresentado no parágrafo anterior, concluímos que o fator "longevidade do sistema" também deve ser considerado nos estudos empíricos. Em outras palavras, sistemas jovens possuem proporcionalmente maior quantidade de *smells* do que sistemas mais longevos, portanto, o fator "longevidade do sistema" também deve ser considerado nos estudos empíricos. Isso pode ajudar a mitigar resultados "contraditórios", como aqueles relatados em nosso estudo (39).

Em termos de estados distintos, ao examinar uma possível relação temporal entre eles, observamos que para o grupo de classes não removidas todos os estados são linearmente relacionados, mas seguem ritmos/proporções diferentes: $1454.91 + 11.34x^1$ (Sem Smell – SS), $224.54 + 1.65x^1$ (Duplicate Code – DC), $83.59 + 0.67x^1$ (LC&CC), $61.53 + 0.48x^1$ (Complex Class – CC), $47.69 + 0.35x^1$ (LC&CC&DC), $45.53 + 0.37x^1$ (Large Class – LC), $27.56 + 0.21x^1$ (CC&DC), $25.54 + 0.17x^1$ (LC&DC). Interessante notar que o estado *Duplicate Code* tem o maior coeficiente

[12] São aquelas classes que existem na última versão do código fonte.

angular, bem como o maior intercepto, indicando que esse *smell* é o mais comum dentre aqueles estudados e possui características próprias (ex. taxa de crescimento). O segundo estado mais comum é uma co-ocorrência dos *smells* Complex Class e Duplicate Code. Também observamos que as equações de alguns estados são numericamente similares, como é o caso dos estados CC&DC e LC&DC.

Do ponto de vista das classes pertencentes ao grupo *"Removidas"*[13] (ver Figura 26), observamos uma relação polinomial entre as janelas de *commit* e a quantidade de classes desse grupo. Esse polinômio tem a concavidade direcionada para baixo, e o ponto mínimo converge para a janela de *commit* que representa a última versão do código fonte. Esse comportamento se repete em todos os estados analisados e significa que no início do sistema, quando a quantidade de classes existente é significativamente menor do que na última versão, os desenvolvedores têm um grande trabalho removendo certas classes dos sistemas. Vale destacar que em nosso estudo, à medida do possível, as classes que passaram por atividades relacionadas à renomear a classe e/ou movê-las para outros pacotes do mesmo sistema, foram rastreadas entre as janelas de *commits* (*RefactoringMiner* (151)) e não fazem parte do grupo *"Removidas"*, elas estão no grupo *"Não Removidas"*. Portanto, as atividades de renomear a classe e/ou mové-las entre pacotes do mesmo sistema não contribui para explicar a relação polinomial apresentada na Figura 26.

Para entender o comportamento polinomial apresentado pelas classes do grupo *"Removidas"* (ver Figura 26), decidimos fazer uma análise das mensagens dos *commits*. Essas mensagens, que estão no formato de texto, são geralmente usadas pelos desenvolvedores para resumir o que foi feito no código fonte. Portanto, elas podem fornecer algum sentido da semântica humana usada na remoção de classes dos sistemas. Isso pode ser feito basicamente de duas formas: i) inspeção manual — cada mensagem passa por uma análise humana que resulta na extração da semântica e/ou classificação dela, exemplo: *thematic analysis* (153); ii) automatizada — utiliza técnicas de *Information Retrieval* (154) e/ou *Data Mining* (155) que não necessitam de interação humana ou essa interação é significativamente menor do que na forma manual. Em nosso estudo, vamos fazer uma combinação dessas duas formas conforme detalhado nos próximos parágrafos.

O primeiro passo é identificar quais *commits* remove(m) arquivo(s) do código fonte. Usando nossos dados, isso pode ser feito comparando o estado de uma dada classe entre a janela k_i e k_{i+1}. Quando o estado da classe na janela k_{i+1} for *"Classe não encontrada"* e o estado dela na janela anterior for diferente de *"Classe não encontrada"*, podemos dizer que algum dos *commits* que estão entre as janelas k_i e k_{i+1} remove uma dada classe do sistema. Então eles são analisados para descobrir

[13] São aquelas classes que **não** existem mais na última versão do código fonte.

qual *commit* remove o arquivo referente a classe. Ao final desse passo, temos um conjunto de mensagens dos *commits* que removeram classes dos sistemas. Em termos numéricos, essa base de dados conta com 11,476 mensagens de *commits* que removem classes.

Seguindo, tal conjunto de mensagens passa por um processo de limpeza e normalização. Isso porque alguns registros são irrelevantes e mensagens como *["...", "*** empty log message ***"]* ou que usam um idioma diferente do inglês, foram removidas do conjunto de dados. Na normalização, termos irrelevantes e muito frequentes foram retirados das sentenças, exemplo: *["git-svn-id", "Change-Id", "Signed-off-by","Also-by"]*. Também removemos palavras que têm comprimento muito grande (>25), e/ou muito pequenas (<3), e/ou têm algum caractere que se repete sequencialmente muitas vezes (>2), e/ou são consideradas *stopwords* do inglês. Por outro lado, alguns termos altamente correlatos que têm grafias diferentes foram normalizados em um único termo, exemplo: *["delete", "remove","drop"]* $\rightarrow$ *"remove/delete/-drop"*. Também usamos a técnica de *Stemmer*[14] para remover os afixos morfológicos das palavras, exemplo: *["removing", "removed","removes"]* $\rightarrow$ *"remove"*. Após esse pré-processamento inicial, o conjunto de dados é formado por 10,717 mensagens de *commits*.

Por fim, usamos uma técnica de *Data Mining* no conjunto de mensagens para extrair termos/tópicos. Uma técnica muito usada no contexto de informação textual é a *Latent Dirichlet Allocation* (LDA) (156), que tem diversas aplicações na engenharia de software e mais recentemente no estudo de *smells* (157, 155). Nessa técnica, cada documento (mensagem de *commit*) pode ser visto como uma mistura de vários tópicos. Sendo ela baseada em modelo probabilístico Bayesiano, que usa a mistura probabilística de tópicos latentes (*latent topics*) e a ligação entre pares de documentos como uma variável binária (155). No contexto do LDA, cada documento é representado por uma distribuição multinomial correspondente ao conjunto de tópicos e cada tópico é representado por uma distribuição multinomial, proveniente do conjunto de palavras pertencentes ao vocabulário do corpus.

Em nosso conjunto de dados (corpus), que são as mensagens dos *commits* de remoção de arquivos do código fonte, aplicamos a técnica LDA para extrair tópicos e palavras associadas a eles. Usando a biblioteca Gensim[15], inicialmente geramos um modelo para nossos dados e extraímos 15 tópicos, cada um contendo 8 palavras que fornecem o seu contexto. Na técnica LDA, uma das dificuldades é dimensionar a quantidade de tópicos. Ao minerar um numero grande de tópicos, os mesmos estarão interrelacionados, dificultando a interpretação dos contextos. Assim, para auxiliar no dimensionamento da quantidade de tópicos, usamos uma representação espacial

[14] <www.nltk.org/api/nltk.stem.html>
[15] https://pypi.org/project/gensim/

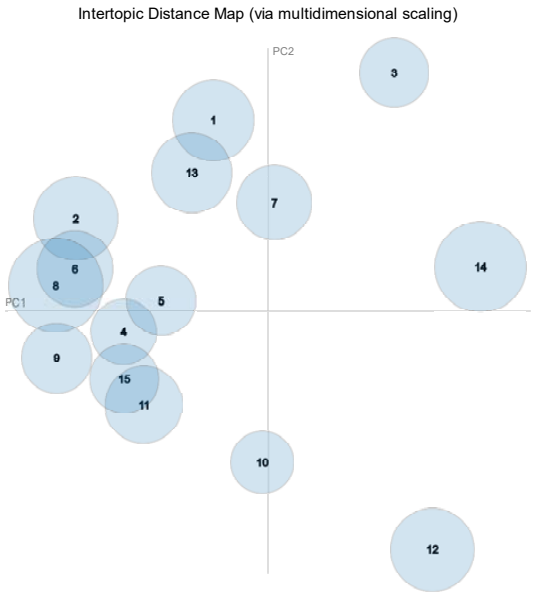


Figura 27 – Visualização espacial (pyLDAvis) dos 15 tópicos extraídos dos *commits* que removem arquivo(s) do sistema.

do nosso modelo LDA produzido pela ferramenta pyLDAvis[16]. A Figura 27, é o resultado da representação espacial produzida pela ferramenta pyLDAvis para os 15 tópicos inicialmente modelados. Com essa representação, é possível visualizar o quanto cada tópico se relaciona com os demais, exemplo: os tópicos 2, 6 e 8 estão se sobrepondo, indicando que podem ser modelados como um único tópico. Assim, usando esse tipo de representação espacial, conseguimos reduzir o número de tópicos modelados para 7. Na visão espacial, esses 7 tópicos têm uma boa separação, o que auxiliará na interpretação de seus contextos.

A Figura 28, detalha as top-5 palavras associadas a cada um dos 7 tópicos minerados. Observa-se que as palavras de um dado tópico têm pesos distintos, exemplo: no tópico 5, os termos *"remove/delete/drop"* têm maior relevância (0,301) do que as outras palavras. Além disso, algumas palavras têm participação em mais de um tópico, exemplo: a palavra *test* participa de 4 tópicos distintos. Isso é um exemplo de como os tópicos podem ser e/ou estar inter-relacionados. Entretanto, no caso desses sete tópicos, não observamos uma palavra que participe de dois ou mais tópicos que tenha um grande peso em cada um deles. Deste modo, em nosso caso, a participação de uma palavra em mais de um tópico não causa grandes efeitos (ex. sobre posição de tópicos). Também observa-se que algumas palavras são *hubs* na formação dos tópicos, especialmente aquelas que apresentam pesos maiores, exemplo: *bugfix/issue* do tópico zero.

A Tabela 19, apresenta a semântica dos sete tópicos listados na Figura 28. Além

[16] O modelo LDA inicial pode ser consultado no endereço <http://lascam.facom.ufu.br/elder/ldamodel.html>

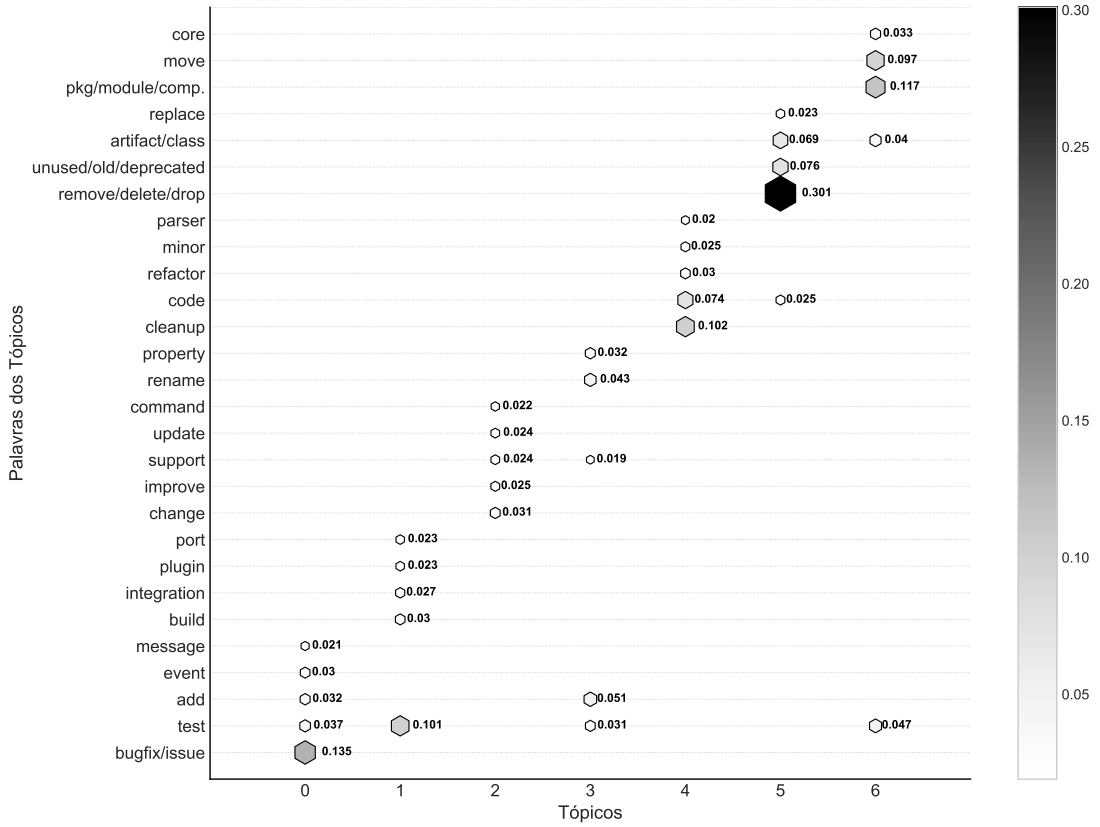


Figura 28 – Resultado do modelo LDA — 7 tópicos.

Tabela 19 – Semântica dos 7 tópicos.

Tópico	**Semântica dos Tópicos**	**N° *Commits***
0	Correções do código fonte (ex. erros) estão associadas à exclusão e/ou inclusão de arquivos no sistema.	1715 (16.00%)
1	O processo de construção de *builds* do sistema e/ou integração, envolvendo testes do sistema, estão associadas à remoção de classes do sistema.	1117 (10.42%)
2	Alterações de melhoria do código fonte, também contribuem para remover classes do sistema.	1249 (11.65%)
3	Testar, renomear e adicionar recursos são tarefas que suportam o desenvolvimento e auxiliam na remoção de classes do sistema.	1412 (13.18%)
4	Tarefas de refatoração e limpeza do código fonte também estão relacionadas ao movimento de classes do sistema.	1095 (10.22%)
5	Entidades do sistema deixam de existir por estarem obsoletas.	1914 (17.86%)
6	Entidades do sistema são movidas para outros locais.	1882 (17.56%)
—	*Commits* não categorizados pelos tópicos acima.	333 (3.11%)
	Total:	**10717 (100%)**

disso, nessa tabela, temos a quantidade de mensagens de *commits* que compõem cada um dos sete tópicos. Observe que cada tópico representa entre 10% e 17% das mensagens analisadas. Além disso, apenas 3% das mensagens não foram categorizadas por algum dos sete tópicos. Isso se deve ao fato de termos reduzido o número

de tópicos, de 15 para 7 tópicos. Na prática, 96% das mensagens de *commits* estão categorizadas em um dos sete tópicos.

Analisando a Tabela 19, observamos que os tópicos: cinco (17,86%), seis (17,56%) e zero (16%) são os mais proeminentes. Pela Figura 28, a palavra com maior peso (0,30%) está no quinto tópico, sendo ela formada pelos termos: *"remove", "delete", "drop"*. Pelas palavras associadas aos tópicos mais proeminentes, podemos inferir que classes são removidas por serem e/ou estarem obsoletas, bem como durante as tarefas de correção de erros e/ou no teste das classes e/ou na inclusão de outras classes. Ao analisar as mensagens de *commits* do tópico zero (ex. termo *"bug*"*) constatamos múltiplos casos em que alguma coisa é removida, seja o arquivo ou parte dele, e o código é corrigido (ex. *"bugfix & removed unused code", "remove unused class and fix the bug"*). O contexto envolvendo *"test*"*, também faz todo sentido, pois quando alguma coisa é removida do código, provavelmente as classes/pacotes de teste também são alteradas para manter a estabilidade do software e isso foi confirmado na análise das mensagens de *commits*: *"remove unused code & add test case", "remove unused NodeCache and TripleCache remove associated tests"*.

Ainda no contexto dos termos usados nos tópicos mais proeminentes, podemos observar que o termo *"test"* ocorre nos tópicos seis e zero. O tópico seis representa 17,56% das mensagens de *commits*, entretanto, há outros termos com maior peso do que o próprio termo *"test"*, são eles: *"pkg/module/component"* (0,117) e *"move"* (0,097). No contexto dos dados coletados, esses termos não deveriam apresentar tamanha afinidade, pois classes de teste não fazem parte da coleta de dados e as classes de produção que foram movidas e/ou renomeadas para outros pacotes do mesmo sistema foram rastreadas com a ferramenta *RefactoringMiner* e estão no grupo das classes *"Não Removidas"* (Subseção 5.1.1.2). Portanto, seguindo esse raciocínio, o contexto de mover/renomear uma dada classe do sistema deveria estar associado às classes *"Não Removidas"*.

Assim, para compreender o significado semântico dos termos *"pkg/module/component"* e *"move"*, analisamos as mensagens e o conteúdo das alterações de código dos próprios *commits* relacionados a esses termos. Como resultado, descobrimos diversos casos em que um conjunto de classes pertencentes a um dado repositório são movidas para outro repositório, exemplo de mensagens desses *commits*: *["move webapp to separate repository camunda-webapp"], ["moving some classes to their own separete module"], ["move api impl, security, identity and connectors to new common module"], ["move all related classes to a separate package"], ["remove workitems from jbpm that have been moved to the jbpm-work-items module"]*. Isso acontece porque muitos dos sistemas analisados têm inter-relação e/ou integração com outros sistemas que possuem o seu próprio ciclo de desenvolvimento, exemplo: o

sistema ActiveMQ[17] da Apache Software Foundation possui sub-sistemas (ex. Artemis[18], Apollo[19]) que têm o seu próprio repositório e portanto seu próprio ciclo de desenvolvimento. Entretanto, encontramos *commits* movendo classes do sistema ActiveMQ para outros repositórios (ex. *"The openwire generation scripts have moved to the activemq-openwire-generator module"*). Esse tipo de *commit*, ocorre com mais frequência no início dos sistemas e, geralmente, o movimento das classes entre sistemas ocorre em rajadas (Figura 29).

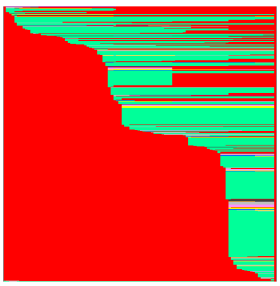


Figura 29 – Fragmento do comportamento de quando as classes migram e/ou são removidas.

A Figura 29 denota parte do fenômeno "migratório" das classes, note que a quantidade de classes "removidas/movidas" mais próximas do início do sistema (lado direto) é maior do que aquelas mais próximas da versão atual do sistema (lado esquerdo), ainda nesse lado se observa que a quantidade de classes realmente deletadas (que não migram de sistema) é maior, porém, isso raramente ocorre em massa (múltiplas classes simultaneamente). Suspeitamos que o comportamento relatado tem relação com a maturidade do sistema, em especial nos aspectos relacionados à organização e/ou ciclo de desenvolvimento. Considerando e analisando os dados do sistema ActiveMQ, a última grande migração (335 arquivos) ocorreu no dia 14 de Outubro de 2017 (*commit ID*: 6037e8f95), $\approx$ um ano antes da última versão do código fonte, e ela culminou na criação de um repositório específico para o sistema *activiti-cloud-service-common*[20]. Assim, constatamos que à medida em que o sistema se desenvolve, subsistemas tomam forma e pode surgir a necessidade de extrair esses subsistemas para seus próprios repositórios que possuem seu próprio ciclo de desenvolvimento. Dessa forma, esse fenômeno ajuda a explicar os gráficos de linha da Figura 26, que têm o comportamento polinomial em que a concavidade é direcionada para baixo e o ponto mínimo converge para a janela de *commit* que representa a última versão do código fonte.

Para o grupo de classes que são removidas e/ou passam pelo fenômeno migratório, observamos que o grau de amadurecimento dos sistemas tem impacto no volume

[17] <https://github.com/apache/activemq>
[18] <https://github.com/apache/activemq-artemis>
[19] <https://github.com/apache/activemq-apollo>
[20] <https://github.com/Activiti/activiti-cloud-service-common>

de classes, bem como em *"quando"* esse movimento ocorre. Isso pode ser observado pela Figura 26, independente do estado analisado, observa-se que no início do sistema o volume de classes removidas e/ou migratórias sempre é maior do que aquele observado nas versões mais recentes. Do ponto de vista prático, temos duas possíveis implicações: i) com base no tempo de amadurecimento do software as ferramentas de detecção de *smells* podem calcular a probabilidade de uma dada classe com um dado *smell* pertencer ou não pertencer ao grupo de classes removidas/migratórias, e baseado no resultado dessa informação, reportar aos desenvolvedores se aquela classe necessita passar por algum processo de manutenção prioritário; ii) as ferramentas que fornecem suporte aos desenvolvedores, seja detectando *smells* (DECOR) ou mesmo para outras finalidades (ex. *Feature Location*, *Refactoring Detection* — `RefactoringMiner`), não estão preparadas para considerar dados históricos de repositórios distintos/independentes. Logo, o suporte fornecido é limitado, especialmente para os sistemas que estão na fase inicial de amadurecimento.

RQ1.1 — Resumo

Ao observar o comportamento apenas das classes afetadas por algum *smell*, descobrimos uma relação linear que cresce à medida que nos aproximamos da janela de *commits* que representa a última versão do código fonte. Esse padrão ocorre em todos os estados de *smells* analisados. Além disso, ao longo do ciclo de vida dos sistemas, observamos que muitas classes deixam de existir nos sistemas. Em geral, isso ocorre por duas razões: i) classes são removidas/excluídas; ii) classes são movidas para outros repositórios. Em particular, o movimento de classes entre repositórios ocorre com mais frequência no início do sistema. Além disso e conforme discutido, isso tem impacto direto na gerência do código fonte, especialmente no aspecto relacionado às ferramentas que auxiliam os desenvolvedores e que não consideram esse fenômeno.

RQ1.2 Quando os estados transitam para outros estados?

Na RQ1.1 contabilizamos os estados que ocorrem em cada uma das k-ésimas janelas, entretanto na contabilização não considera-se o estado das janelas adjacentes. Considere duas classes que existem no sistema por 9 janelas de *commits* ($K_{0..8}$), em que, para cada janela, as classes se apresentam em determinado estado (destacados por cores), veja: ■■■■■■■■■ (Classe A) e ■■■■■■■■■ (Classe B). Nesse cenário e na questão anterior, a contabilização dos estados considera apenas existência deles em cada janela de *commit*, como mostrado na Tabela 20, exemplo: o estado ■ ocorre nas janelas K_0, K_1, K_2, K_7, K_8. A ideia é fornecer uma visão geral de quando (janelas de *commits*) os estados ocorrem. O resultado dessa perspectiva é aquele apresentado na Figura 26.

Tabela 20 – Contabilização dos estados por Janela de *Commit*.

	Janelas de *Commits*								
	k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8
■	1	2	2	0	0	0	0	2	2
■	0	0	0	1	1	1	1	0	0
■	1	0	0	1	1	1	1	0	0

Por outro lado, na atual questão, a contabilização de um dado estado, de determinada janela, ocorre se somente se houver uma transição entre a janela atual e a próxima janela. A ideia é fornecer uma visão geral de quando, isto é, em quais janelas de commits os estados transitam para outros estados. Nesse caso, uma transição marca o "início" de um estado. Considerando o exemplo anterior das Classes A e B, observamos que na janela k_0 temos uma transição entre o estado verde e o roxo. Depois, temos duas transições na janela k_2 uma do roxo para o rosa, e outra do roxo para o verde. A Tabela 21, resume essa contabilização para o exemplo das Classes A e B. Observe que nessa perspectiva, um dado estado pode ter seu estado de "inicialização" contabilizado em múltiplos pontos, como é o caso do estado verde da Classe B que tem transições para esse estado nas janelas k_0 e k_6.

Tabela 21 – Contabilização de transições de estados por Janela de *Commit*.

	Janelas de *Commits*								
	k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8
■	0	0	2	0	0	0	0	0	2
■	0	0	0	0	0	0	1	0	0
■	1	0	0	0	0	0	1	0	0

A Figura 30, demonstra o comportamento da transição de estados ao longo das k-ésimas janelas de *commits*. Observe que temos oito gráficos, um para cada estado analisado, que registram o número médio de transições ocorridas em cada janela de *commits*. Nos gráficos de linha representados na Figura 30, para cada estado, temos duas linhas: uma descreve o comportamento de um dado estado para as classes que ainda existem na versão atual do código fonte, e outra para detalhar a conduta desse mesmo estado nas classes que foram removidas do sistema. Na Figura 30, além dos gráficos de linhas também temos alguns gráficos de barra que demonstram a estabilidade média de cada estado conforme a ocorrência de transição de estados em uma dada janela de *commit* (ver Algoritmo 2).

Para complementar nossa análise, também demonstramos os estados que desencadearam a transição para um dado estado na k-ésima janela de *commit* (ver Figura 31), exemplo: pela Tabela 21, observamos duas transições na janela k_2, uma do estado roxo para o rosa e outra do estado roxo para o verde. Assim, temos dois estados (rosa e verde) que desencadeiam o estado roxo na janela k_2. Portanto, a Figura 31 complementa

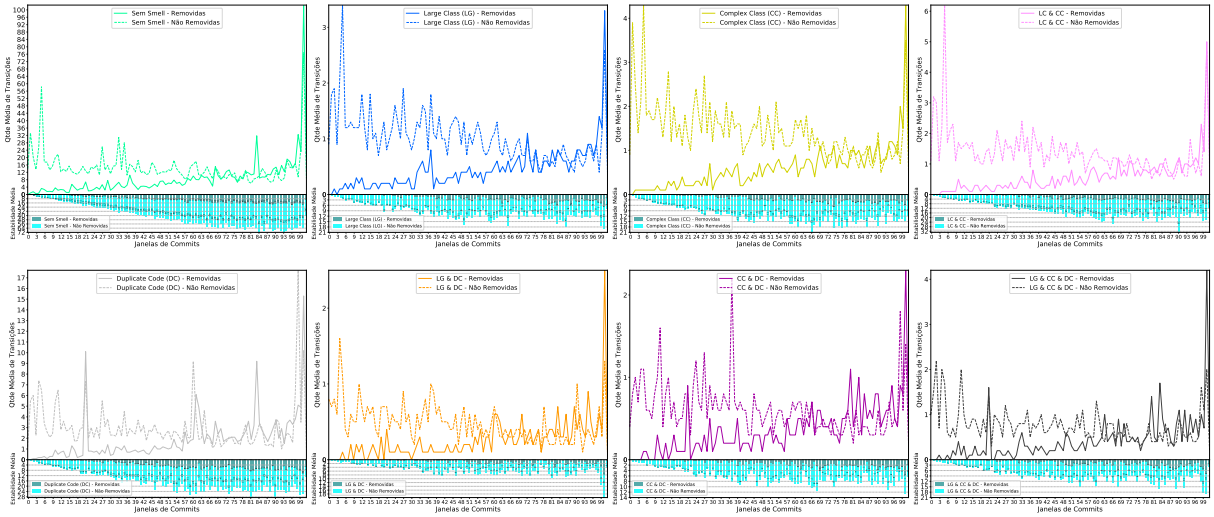


Figura 30 – Evolução histórica de cada transição de estado e estabilidade média dos estados, desmembrado conforme classes removidas e não removidas.

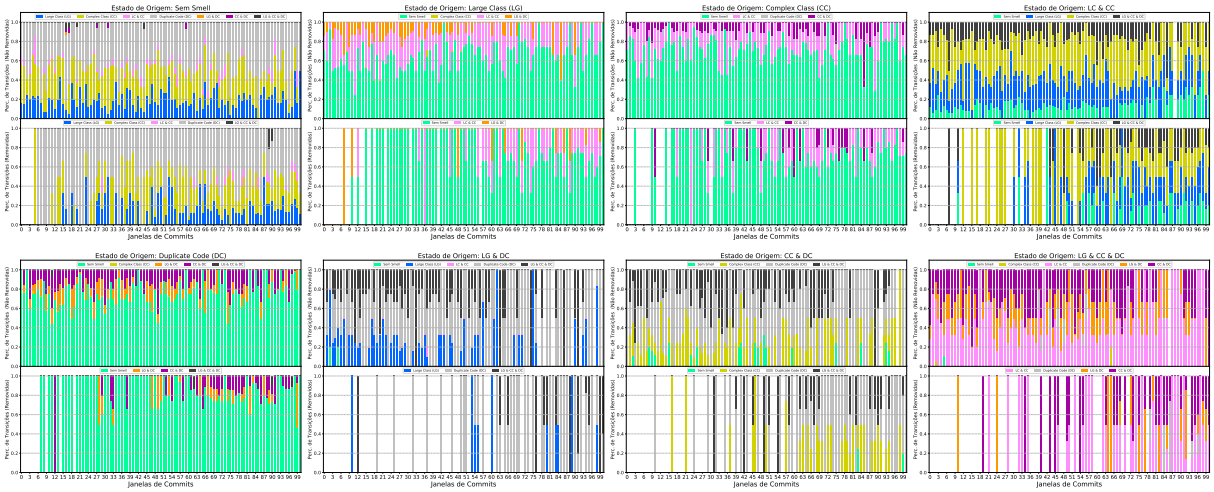


Figura 31 – Detalhamento das transições ocorridas em cada estado ao longo das K janelas de *commits*, desmembrado conforme classes removidas ou não removidas.

as informações da Figura 30, mostrando percentualmente para quais estados ocorrem as transições. Na Figura 30, considerando o grupo de classes não removidas, observamos que na primeira versão do código fonte (k_{100}) temos aproximadamente 74 classes efetuando uma transição para o estado *"sem smell"*. Consultando a Figura 31, observamos que 50% das transições para estado *"sem smell"*, ocorridas na janela k_{100}, foram desencadeadas pelo estado precursor Duplicate Code e o restante pelo estado precursor Large Class. Para simplificar a compreensão, essa figura não considera transições que envolvem o estado *"Classe não encontrada"* que geralmente é sinalizada pela cor vermelha. Outro fator que motivou essa decisão foi o fato de que esse estado ocorre em todas as classes (ex. na sua concepção) e, portanto, seu volume é demasiadamente alto. Entretanto, adiante criamos a Tabela 24 que demonstra, em termos percentuais, quanto cada tipo de transição ocorre nos sistemas. Nessa tabela, transições envolvendo esse estado foram contabiliza-

das. Note ainda que a Figura 31, apresenta um gráfico de barra para cada estado e estes estão subdivididos conforme as classes removidas e não removidas.

- ❑ `Análise e discussão dos gráficos representados na Figura 30 e 31.`

  Pela Figura 30, para o grupo de classes não removidas, observamos que depois de um dado ponto de deflexão, o número médio de transições aumenta à medida que a janela de *commit* se aproxima da última versão do código fonte. Por outro lado, para o grupo de classes removidas e após um dado ponto de deflexão, o número médio de transições diminui à medida que a janela de *commit* caminha em direção a última versão do código fonte. Interessante notar que nas primeiras versões do código fonte, o comportamento do número médio de transições não se altera significativamente entre os dois grupos (classes removidas e não removidas). Contudo, existe um ponto de deflexão em que ambas as linhas de tendência começam a divergir e esse comportamento prossegue aumentando até atingir a janela de *commit* que representa a última versão do código fonte. Independente do estado de transição analisado, todos apresentam linhas de tendência que têm o comportamento descrito anteriormente. Entretanto, alguns estados de transição apresentam pontos de deflexão em locais diferentes de outros estados de transição, exemplo: os estados *"SS", "LC", "CC", "LC&CC"* da Figura 30, têm pontos de deflexão entre as janelas k_{66} e k_{69}, já os estados *"DC", "LC&DC"*, apresentam pontos de deflexão entre as janelas k_{53} e k_{57}. Isso implica que alguns estados desempenham um papel diferenciado na relação entre a taxa e o momento em que as transições ocorrem. Podemos dizer que alguns estados são mais propensos à ocorrência transições, em especial, alguns estados já apresentam uma acentuação nas transições mais no início do sistema.

  Inicialmente, hipotetizamos que essa diferença no ponto de deflexão entre alguns estados teria uma relação com a quantidade de classes em cada estado. Entretanto, analisando os estados que apresentam o ponto de deflexão mais próximo da versão inicial do código (janelas k_{66} e k_{69}) observamos que o volume de classes não é um fator decisivo. Isso porque nesse grupo temos estados com alto número de classes (ex. *"SS", "LC&CC"* – ver equações de regressão linear da Figura 26) e estados que têm um número significativamente menor de classes (ex. *"LC"* – ver equações de regressão linear da Figura 26). Além disso, ao analisar os estados que têm pontos de deflexão entre as janelas k_{53} e k_{57}, também observamos estados que possuem uma alta taxa de classes (ex. *"DC"*) e estados com um número pequenos de classes (ex. *LC&DC*). Portanto, o fator de quantidade de classes em cada estado não é um fator decisivo para explicar a diferença no ponto de deflexão entre alguns estados, ou seja, o interesse "precoce" dos desenvolvedores por alguns estados não é completamente explicado pelo número de classes dos estados.

Continuando nossa investigação sobre a diferença no ponto de deflexão de alguns estados, hipotetizamos que as classes nos estados *"SS", "LC", "CC", "LC&CC"* têm uma importância maior para a evolução do sistema do que as classes nos estados *"DC", "LC&DC"* e por isso, no início do sistema, esses estados têm pontos de deflexão diferente. Para investigar essa hipótese, vamos verificar os estados que precederam as transições para os estados *"SS", "LC", "CC", "LC&CC"*, em especial, nas janelas de *commits* $k_{66} - k_{69}$ (ponto de deflexão da Figura 30). Observa-se que a maior parte dos estados predecessores são os próprios estados investigados, ou seja, *"SS", "LC", "CC", "LC&CC"*, exemplo: os principais estados predecessores ao estado *"LC"* são os estados *"SS","LC&CC"* (ver Tabela 22). Assim, nesses casos, as transições ficam restritas aos próprios estados analisados. Ao considerar que uma transição de estado ocorre quando o desenvolvedor altera o código fonte da classe de forma mais acentuada/significativa e que os estados com ponto de deflexão precoce têm a maior parte de suas transições restritas aos próprios estados, podemos concluir que: os estados *"SS", "LC", "CC", "LC&CC"* têm uma importância maior na evolução inicial do sistema do que outros estados (ex. *"DC", "LC&DC"*), confirmando assim nossa hipótese. Isso quer dizer que, as classes nos estados *"SS", "LC", "CC", "LC&CC"* sofrem múltiplas alterações de código que são suficientemente grandes para causar diversas transições de estado que estão localizadas mais no início do sistema. Entretanto, no início do sistema, outros estados não sofrem tantas alterações de código que causam transições de estados (no tempo de vida do sistema, isso acontece mais adiante). Demonstra-se assim, que certos estados são mais importantes para a evolução inicial do sistema. Além disso, o comportamento descrito pode indicar que alguns estados têm características cíclicas, ou seja, uma dada classe que apresente o estado X transita por outros estados, mas em algum momento ela retorna ao estado inicial X (isso será investigado adiante, RQ3).

Tabela 22 – Principais estados que precederam as transições para os estados *"SS", "LC", "CC", "LC&CC"* (janelas $k_{66} - k_{69}$).

Estado		Estado		Estado		Estado	
Destino	Origem	Destino	Origem	Destino	Origem	Destino	Origem
SS	DC	LC	SS	CC	SS	LC&CC	CC
	LC		LC&CC		LC&CC		SS
	CC						LC

Com base no que foi discutido no parágrafo anterior, os estados *"SS", "LC", "CC", "LC&CC"* são alvo de modificações/alterações de código tão significativas que observamos transições de estados, em particular, destaca-se que as transições desses estados acentuam-se antes do acentuamento das transições de outros estados (ex. *"DC"*). Sugerimos que esse comportamento pode ser usado para efetuar uma "priorização inter-*smell*", ou seja, priorizar determinados tipos de *smells* em detrimento

de outros, exemplo: no início do sistema (ex. primeiras janelas de *commits*), quando classes com determinados estados forem alteradas (ex. *"CC", "DC"*), priorizar a revisão de código daquelas classes que tinham os estados: *"SS", "LC", "CC", "LC&CC"*. Direcionando assim, as atividades de monitoramento. Isso quer dizer que, no início do sistema, a inspeção da qualidade do código pode ser feita de forma mais seletiva, ou seja, não seria necessário examinar/acompanhar todas as classes alteradas.

Ainda pela Figura 30, se observa que independente do estado analisado, todos têm classes removidas e não removidas. Para as classes removidas, o número de classes que efetuam transições se torna menor à medida que a janela de *commits* se aproxima da última versão do código fonte. Em termos de *smells*, a existência concomitante desses dois tipos de classes prejudicam as ferramentas que fornecem suporte aos desenvolvedores, bem como a tomada de certas decisões de desenvolvimento e/ou monitoramento do sistema. Para compreender, considere:

> i) a equipe de desenvolvimento de um dado sistema deseja identificar as classes que apresentam determinados *smells* para efetuar algum tipo de manutenção. Então uma ferramenta de detecção de *smell* é executada (ex. DECOR), produzindo uma lista de classes e seus respectivos *smells*. Na sequência, essas classes são averiguadas por desenvolvedores. Entretanto, o conjunto de classes reportada pela ferramenta de detecção de *smells* contém classes que serão removidas nas próximas versões. Desse modo, em algum momento, desenvolvedores se ocuparam desnecessariamente, pois certas classes alvo do processo deixarão de participar do sistema.
>
> ii) um desenvolvedor que esteja executando ou vá executar alguma tarefa de desenvolvimento em uma classe que tem chances de ser removida do sistema, poderia usar essa oportunidade para efetuar uma reestruturação que resulte na remoção da classe. Evitando assim o retrabalho, ou seja, efetuar uma alteração na classe que no futuro sofrerá uma reestruturação que causará sua remoção.

Assim, os cenários sugerem a necessidade de mecanismos e/ou técnicas para realizar a priorização das entidades conforme o objetivo do desenvolvimento. O primeiro cenário é o caso de uma priorização intra-*smell*. O outro é similar, mas é uma priorização da reestruturação baseado na necessidade de alterar uma dada classe por algum motivo.

Na Figura 30, temos gráficos de barra que demonstram a estabilidade média (ver Algoritmo 2) de um dado estado variando conforme a janela de *commit* analisada, exemplo: classes no estado Large Class da janela de *commit* k_{99} têm estabilidade média de 4 janelas de *commits* para as classes do grupo que foram removidas e estabilidade média de 12 janelas de *commits* para as classes do grupo que **não** foram removidas. Considerando esse mesmo exemplo, mas em outra janela de *commit*

(k_{98}), a estabilidade média de ambos os grupos é de 4 e 7 janelas de *commits*, respectivamente. Ao analisar de forma mais ampla, os gráficos de barra da Figura 30 demonstram, que a estabilidade média das classes que foram removidas é, em grande parte, menor do que a estabilidade média das classes **não** removidas. Em termos numéricos, a razão entre estabilidade média das classes **não** removidas e aquelas removidas é maior ou igual a dois em 93,4% dos casos, ou seja, em apenas 6,6% dos casos essa métrica não consegue identificar corretamente se a classe será ou não removida. Isso quer dizer que essa métrica apresenta a capacidade de classificação, em particular é possível identificar se uma classe tem chance de ser removida no futuro. Portanto, a métrica de estabilidade média pode ser usada na priorização das entidades conforme o objetivo do desenvolvimento, ou seja, realizar a priorização intra-*smells* e/ou priorização de reestruturação descrita acima.

Cabe ressaltar que em nosso estudo, não elaboramos um classificador que usa a métrica de estabilidade média. Ao invés disso, apresentamos indícios de sua viabilidade em operações de classificação. A construção de um classificador requer um estudo amplo, no qual essa métrica é analisada sob diversas abordagens a fim de identificar a melhor forma de aplicar, bem como suas restrições, exemplos: i) essa métrica deve ter um limiar absoluto ou estatístico (ver Subseção 2.2.2); ii) a quantidade de *commits* existente dentro de uma dada janela de *commits* tem alguma influência na precisão e/ou acurácia do classificador. Assim, a construção e validação do classificador será realizada em um trabalho futuro.

RQ1.2 — Resumo

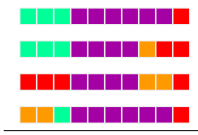
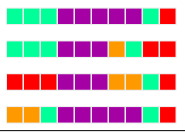
Na RQ1.1 observamos que, ao longo do ciclo de vida dos sistemas, diversas classes deixam de existir e isso é um inconveniente no processo de desenvolvimento dos sistemas. Nesse caso, ferramentas acabam dizendo aos desenvolvedores para refatorar certas classes que deixarão de existir no sistema. Entretanto, classes que irão deixar de existir nos sistemas não deveriam ser sugeridas à refatoração ou, talvez, em determinadas tarefas, possam ser sugeridas com uma prioridade específica. Assim, nessa questão de pesquisa, apresentamos indícios de que a métrica *"estabilidade média das transições"* pode ser usada para separar (clusterizar) as classes em dois grupos: *Não Removidas* ou *Removidas*. Permitindo, por exemplo, que as ferramentas de detecção de *smells* façam a priorização da indicação de classes que devem passar por refatoração. Também mostramos que certos estados de *smells*, ocorrendo no início do ciclo de vida dos sistemas, são alvo de alterações que causam transições de estados e têm pontos de deflexão deslocados em relação a outros estados. Conforme discutido, esse comportamento pode ser usado para efetuar "priorização inter-*smell*".

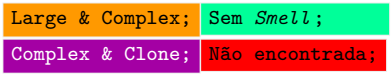
RQ1.3 Quando os estados de *smells* surgem pela primeira vez?

Nessa questão, investigamos se os *smells* são introduzidos como consequência das atividades contínuas de manutenção e evolução (15). Estudamos quando os *smells* são introduzidos, investigando se eles são introduzidos assim que uma entidade é criada ou se são repentinamente introduzidos no contexto de atividades específicas. Como mencionado anteriormente, esse tipo de questão de pesquisa fora estudada no passado (55, 146). Contudo não consideraram os *smells*: Duplicate Code, Large Class e Complex Class, bem como a co-ocorrência deles.

Para investigar essa perspectiva vamos examinar qual é a representatividade das classes que têm *smells* e estes foram introduzidos no momento de criação da classe. Para isso, o primeiro passo é filtrar apenas as classes de interesse, ou seja, selecionar todas as classes que apresentem pelo menos um dos estados que denotam algum *smell*. Em outras palavras, vamos descartar todas as classes que têm o estado *"Sem Smell"* e não realizaram transições para outros estados. Na sequência, agrupamos as classes conforme o momento em que ocorreu o primeiro estado que representa algum *smell*. Nesse caso, temos dois grupos: i) classes que nasceram com algum *smell* e ii) classes que nasceram sem *smell* mas em algum momento futuro apresentaram ou apresentam algum *smell*. O primeiro grupo, referenciado como "congênito", é formado pelas classes que têm a primeira janela de *commit* marcada com o estado de algum *smell*, ou seja, estado inicial $\neq$ *"Sem Smell"*. Esse grupo é representado na primeira coluna da Tabela 23, observe que na janela de *commit* K_9 as classes não existem no sistema e na próxima janela (k_8) diversas classes são introduzidas no sistema, sendo que cada uma delas nasce com algum *smell*. Observe ainda que nesse grupo temos classes que nasceram com algum *smell*, mas que no futuro foram removidas do sistema ou mesmo que se tornaram saudáveis (*Sem smell*). A Tabela 23, também ilustra o outro grupo ("a posteriori"), que são classes introduzidas no sistema sem a existência de *smells*, mas apresentam algum *smell* ao longo do ciclo de vida do sistema.

Tabela 23 – Classificação das classes conforme o momento em que os *smell* se manifestam.

Grupo I Congênito	Grupo II A posteriori
	



Seguindo, para cada sistema analisado, calculamos a proporção de classes que nascem com algum *smell* e a proporção de classes que se tornam doentes (*smells* surgem a posteriori). O resultado dessa proporção é mostrado na Figura 32. Observe que o resultado é apresentado na forma de três grandes grupos: i) Todas – considera todas as classes independente se elas foram removidas ou se continuam até a versão atual do sistema;

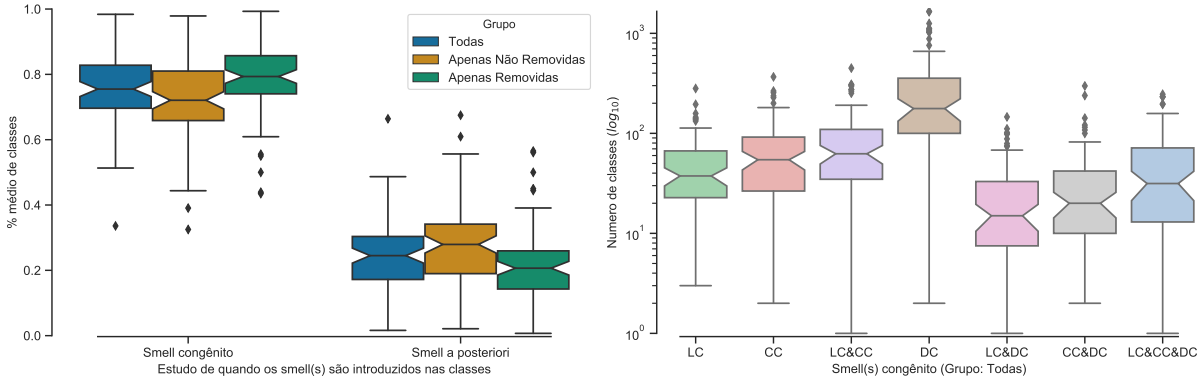


Figura 32 – Relação entre o nascimento de *smells* e a inclusão das classes.

ii) Apenas Não Removidas – considera apenas as classes que não foram removidas do sistema; iii) Apenas Removidas – é justamente o oposto do grupo anterior.

❑ `Análise e discussão dos gráficos representados na Figura 32.`

Analisando os dados do gráfico a esquerda na Figura 32 observamos que, independente do grupo, a maior parte das classes com *smell(s)* (incluindo aquelas que tiveram *smell(s)* no passado) foram concebidas com eles. Portanto, a proporção das classes que nascem com *smell(s)* é maior do que a proporção de classes que passam a apresentar *smell(s)* após sua criação. Em termos numéricos, $\approx 78\%$ das classes analisadas nascem com algum *smell*. Considerando os diferentes grupos das classes que nascem com *smell(s)*, observa(m)-se pouca(s) diferença(s) entre ele(s), especialmente na forma dos boxplots, bem como na mediana deles. Por isso, no gráfico à direita da Figura 32, analisamos apenas os dados das classes que nascem com *smell(s)* e pertencem ao grupo *Todas*. Esse gráfico, apresenta sete boxplots, um para cada estado associado aos *smells* estudados neste capítulo. Por essa figura e conforme a posição/amplitude interquartílica que representa a metade (50%) dos dados, podemos agrupar os dados em dois grupos: O primeiro é formado pelos estados *"LC", "CC", "LC&CC", "DC"* e o segundo por *"LC&DC","CC&DC","LC&CC&DC"*. O estado *"DC"* do primeiro grupo tem um grande destaque, pois além de seus dados apresentarem a maior amplitude, a maioria dos sistemas têm de 100 a 200 classes com clones que surgiram na concepção das classes. Cabe ressaltar que a variabilidade é alta, pois alguns sistemas têm valores tão baixos quanto 2 classes e tão elevadas quanto 1000 classes. Resumindo, a estratégia de clonagem é bastante usual entre os desenvolvedores, contudo, em certos sistemas, seu uso é limitado/restrito. Ainda sobre o primeiro grupo, observa-se que alguns sistemas não têm classes que nascem com a co-ocorrência dos *smells* Large Class e Complex Class. Por outro lado, todos os sistemas analisados têm classes que nascem com algum dos estados: *"LC","CC"*. Quanto ao segundo grupo, o destaque é a co-ocorrência de *smells* descrita pelo estado *"LC","CC","DC"*, pois 50% dos sistemas analisados têm

de 10 a 39 classes que nasceram com esse estado, mas houve casos em que esse estado não ocorreu no nascimento das classes. Esse grupo pode ser visto como aquele formado pela co-ocorrência de *smells* que têm predominantemente baixa frequência de ocorrência no nascimento das classes.

Nesse mesmo gráfico surgem outras questões relacionadas ao que acontece com os *smell(s)* congênito(s). Nesse caso, vamos investigar se as classes com *smell* congênito permanecem ou não com algum *smell* na última janela de *commit* observável. Esse tipo de investigação permite estabelecer se os desenvolvedores têm interesse em remover *smell(s)* e, após a refatoração, manter essas classes "saudáveis" enquanto elas existirem no sistema. Nessa investigação, vamos usar os mesmos dados do gráfico a direita da Figura 32, mas eles serão categorizados em dois tipos: i) classes que terminaram sem *smell* — que são classes que têm a última janela de *commits* em que elas aparecem marcadas com o estado *"Sem smell"*, mostradas na Figura 33 à esquerda; ii) classes que terminaram com algum *smell* — que são classes que têm a última janela de *commits* em que elas aparecem marcadas com estado $\neq$ de *"Sem smell"*, mostradas na Figura 33 à direita. Observa-se, que não existe um resultado significativo na remoção de *smells* e manutenção das classes em um estado saudável.

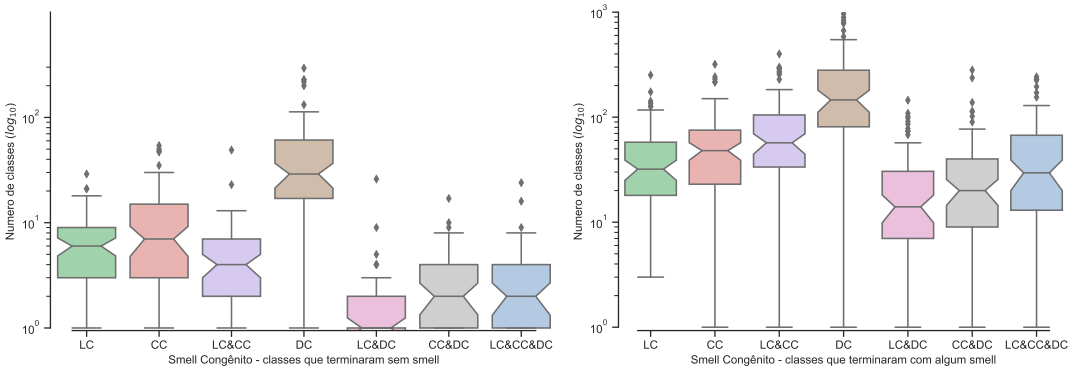


Figura 33 – Relação entre *smell(s)* congênito e o último estado observado.

Ainda pela Figura 33, o melhor caso de remover *smell(s)* e manter as classes "saudáveis" ocorre com o estado DC. Pelo intervalo interquartil do boxplot, observa-se que 50% dos dados analisados mantêm "saudáveis" um número restrito de classes (12 a 40) que nasceram com o *smell* Duplicate Code. Em termos numéricos e considerando a mediana da Figura 32 do lado direito e da Figura 33 do lado esquerdo, 12% das classes em que o *smell* Duplicate Code é congênito, passam por refatoração e são mantidas sem esse *smell*. Por outro lado, não há nenhum resultado no sentido de manter "saudáveis" as classes que nasceram com a co-ocorrência de *smells*, especialmente aquelas sinalizadas como: LC&DC, CC&DC, LC&CC&DC. O pior caso ocorre com o estado LC&DC, pelo boxplot na Figura 33 do lado esquerdo, observa-se que 75% do dados analisados mantêm "saudáveis" no máximo 2 classes que nasceram com a co-ocorrência dos *smells* Large Class & Complex Class.

Note ainda que, 50% desses dados não têm registro algum de manter esse tipo de classe "saudável". Ao analisar o estado LC&CC&DC e considerando a mediana, apenas 7% das classes que têm essa co-ocorrência congênita passaram por refatoração e são mantidas sem esse *smell*. Talvez tal comportamento possa ser explicado, ao menos em parte, pelo fato de que a quantidade de classes nesses estados é relativamente pequena (ver Figura 32b), ou porque, a complexidade para manter esse tipo de classe sem *smell* é relativamente mais alta. Além desses possíveis motivos, também argumentamos que o desconhecimento (ao menos formal — emitido por alguma ferramenta que fornece suporte ao desenvolvimento) desse tipo de co-ocorrência de *smell* pode induzir esse comportamento. Uma hipótese seria que se os desenvolvedores não possuem a real magnitude desse tipo de situação, os mesmos não podem se esforçar para manter a situação controlada. Essa hipótese é reforçada pelo fato de que não se tem conhecimento de ferramentas de detecção de *smell* que reportem a co-ocorrência de *smells*, permitindo o acompanhamento dessas classes.

Comparando a Figura 33 do lado esquerdo e direito, fica claro que a maior parte das classes com *smell(s)* congênitos continuam com *smell* na última janela de *commit* em que elas ocorrem. Isso implica em dois possíveis casos: i) $Estado_{Inicial} \equiv Estado_{Final}$ — há classes onde o estado da última janela de *commit* em que elas aparecem, coincide com o estado apresentado no seu nascimento (a classe começa e "termina" com o mesmo estado) e ii) $Estado_{Inicial} \neq Estado_{Final}$ — há classes em que o estado da última janela de *commit* observável **não** coincide com o estado apresentado no seu nascimento. Considerando os dados da Figura 33 do lado direito, investigamos se algum desses casos ocorre mais do que o outro. Para isso, criamos a Figura 34, em que o lado esquerdo representa o primeiro caso e o lado direito o outro.

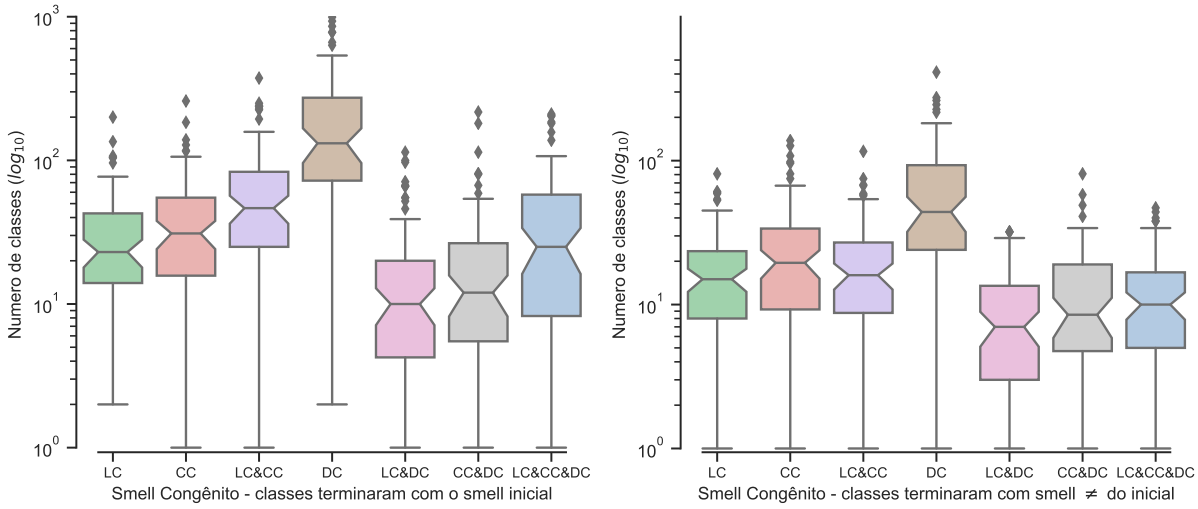


Figura 34 – Transição entre o estado inicial congênito e o último estado observável.

Ao considerar as medianas apresentadas na Figura 34, em geral, podemos dizer que é menos frequente a ocorrência de uma transição de estado entre a janela de *commit* da

concepção da classe e a última janela de *commit* analisada dessa mesma classe. Ou seja, o caso $Estado_{Inicial} \equiv Estado_{Final}$ é mais comum, por exemplo: considerando o estado DC, o teste Mann-Whitney demonstra que o número de classes com *smell* congênito que se mantiveram no mesmo estado observado no ato da criação da classe é significativamente maior do que aquelas em que o estado final é diferente daquele observado na criação da classe ($U = 7982,5;\ P < 0,05;\ r = 0,476$).

Analisando os dados das classes que começam e "terminam" com o mesmo estado ($Estado_{Inicial} \equiv Estado_{Final}$), observamos casos com transições intermediárias. Isso implica que, entre as janelas das extremidades, existem classes com estados diferentes daqueles mostrados nas janelas de *commits* das extremidades, ou seja, há indícios de classes com *smell(s)* cíclicos. Em termos numéricos e considerando os dados da Figura 34a, classes contendo o estado CC nas janelas da extremidade tem o maior percentual (13,8%) de classes com estado(s) intermediário(s). A segunda e terceira posição são ocupadas, respectivamente, pelos estados LC (13,5%) e CC&DC (11.7%). Por outro lado, os menores percentuais ocorrem nos estados LC&CC&DC (8,1%) e DC (4,6%). Esse resultado reforça a necessidade de analisar o possível comportamento cíclico dos *smells*, assunto este relatado adiante.

Em geral e com base nas análises anteriores, observamos que *smells* já nascem com as classes e ao analisar o estado final dessas, observa-se que elas mantém o mesmo *smell* observado no nascimento da classe. Assim, podemos concluir que, a maior parte dos *smells* são congênitos e permanecem "intocados" ao longo do tempo, tanto no sentido de não serem removidos definitivamente da classe (por vezes têm características cíclicas), quanto no sentido de **não** "evoluírem" para outro estado de *smell* (ao menos quando analisamos o estado da primeira e da última janela de *commits* — as janelas intermediárias serão examinadas na RQ3).

RQ1.3 — Resumo

Confirmamos que a maior parte das classes com algum *smell* já nascem nessa condição (78%). Também observamos que muitas dessas classes mantêm na última versão do código fonte o mesmo *smell* observado no seu nascimento. Ao analisar o ciclo de vida dessas classes, algumas vezes observamos transições intermediárias. Contudo, no fim (última versão do código fonte), o *smell* observado na concepção da classe retorna à classe. Indicando a existência de comportamento cíclico de *smells* (Investigado na RQ3). Além disso, também observamos que as classes com certas co-ocorrências de *smells* congênitos (ex. *LC&DC*), raramente terminam sem *smell*, ou seja, não apresentam *smell(s)* na última versão do código fonte.

RQ2 — Entidades com a simples ocorrência de *smell* podem se tornar entidades com a co-ocorrência deles?

Segundo Lehman et al. (149), um sistema se tornará progressivamente mais complexo, a menos que alguma coisa seja feita para reduzir explicitamente sua complexidade. Então, se nada é feito, é natural pensar que uma simples ocorrência de *smell* pode se tornar ao longo do tempo uma co-ocorrência de *smells*. Assim, essa questão investiga essa hipótese e para responder essa questão de pesquisa levantamos o percentual correspondente à participação de cada tipo de transição, pois examinando a taxa de transições entre estados definidos como a simples ocorrência de *smells* para estados de co-ocorrência, podemos verificar se existe a combinação gradual e/ou acumulativa de *smells*. Conforme apresentado anteriormente, uma transição ocorre quando temos dois estados distintos entre duas janelas de *commits* consecutivas (k_i e k_{i+1}). Neste estudo, definimos nove estados distintos (ver Subseção 5.1.1.3) e com base no conceito de transição, o espaço combinatório resulta no total de 72 tipos distintos de transições. Transição não possui a propriedade comutativa[21], por exemplo: a transição CC → DC tem um significado diferente de DC → CC. Assim, para calcular o percentual de contribuição de cada transição, para cada sistema analisado, calculamos a quantidade total de transições por classe e realizamos a totalização individual de cada tipo de transição. Com esses valores, para cada tipo de transição, obtemos o percentual médio por sistema. Esse procedimento está detalhado e representado no Algoritmo 3.

Algoritmo 3: Percentual médio de contribuição de cada tipo de transição.

```
1  AvgTransactionsContribution (DataSet)
       input : DataSet is the dataset collected (Projects, Classes and Status).
       output: Contribution, in percentage, of each transition.
2      EndResult ← HashMap<Status,HashMap<Status,int>>();
3      foreach project ∈ DataSet do
4          SetResultP ← HashMap<Status,HashMap<Status,int>>();
5          foreach class ∈ project do
6              SetResult ← HashMap<Status,HashMap<Status,int>>();
7              k ← 0; NTransitions ← 0;
8              while k < GetNumberOfWindows(class) − 1 do
9                  StatusK ← GetStatus(class, k);
10                 StatusNext ← GetStatus(class, k + 1);
11                 k ← k + 1;
12                 if StatusK ≠ StatusNext then
13                     CounterIncrement(SetResult, StatusK, StatusNext);
14                     NTransitions ← NTransitions + 1;
15             SetResultP ← SetResultP + (SetResult ÷ NTransitions);
16         EndResult ← EndResult + (SetResultP ÷ GetNumberOfClasses(project));
17     return EndResult ÷ GetNumberOfProjects(DataSet);
```

[21] Operação é comutativa se somente se uma alteração na ordem dos elementos não altera o resultado.

Diferentemente do que foi feito na questão de pesquisa anterior, o Algoritmo 3 não faz distinção entre classes removidas e/ou que ainda existam no sistema. Essa distinção é relevante quando queremos investigar a relação temporal de fatores que estão possivelmente relacionados. Entretanto, essa questão de pesquisa não investiga a relação temporal, e sim a relação transicional de estados. Por isso, essa questão de pesquisa não faz distinção entre classes removidas e/ou não removidas.

A Tabela 24, apresenta o resultado do Algoritmo 3. Nela, com exceção das células pertencentes à diagonal principal, cada célula representa um tipo de transição. A existência de uma transição implica na ocorrência de um estado predecessor e um sucessor, que são distintos entre si, exemplo: $Estado_{Predecessor} \rightarrow Estado_{Sucessor}$. Na Tabela 24, os estados predecessores são representados pelas colunas e as linhas refletem os estados sucessores. Desse modo, a interseção entre a primeira coluna e a segunda linha denota a transição NE → SS, ou seja, essa transição representa a inclusão/criação de classe(s) com o estado *"Sem Smell"*. Observe que nessa tabela, além dos estados que denotam a existência de *smell(s)* (ex. LC), também temos dois estados complementares: *"Classe não encontrada (NE)"* e *"Classe sem smell (SS)"*. Eles foram incluídos para proporcionar uma visão ampla do estudo de *smells*.

No contexto deste estudo, uma transição não apresenta a propriedade comutativa. Portanto, a posição em que um dado estado ocorre nas transições exerce influência em sua semântica, exemplo: NE → SS e SS → NE. Essas duas transições são antagônicas, no exemplo, a transição NE → SS denota a inclusão de classes sem *smell* e a transição SS → NE indica a remoção de classes sem *smell*.

Tabela 24 – Percentual geral de transições ocorridas nos sistemas.

Predecessor / Sucessor	NE	SS	LC	CC	DC	LC&CC	LC&DC	CC&DC	LC&CC&DC
NE		11.75	0.26	0.33	1.90	0.44	0.15	0.20	0.34
SS	60.50		0.30	0.48	1.41	0.08	0.01	0.02	0.03
LC	1.15	0.51		0.01	0.01	0.18	0.08	0.00	0.01
CC	1.47	0.86	0.00		0.01	0.19	0.00	0.10	0.01
DC	8.18	1.68	0.00	0.01		0.00	0.10	0.13	0.01
LC&CC	2.08	0.15	0.26	0.34	0.01		0.01	0.01	0.20
LC&DC	0.67	0.02	0.09	0.00	0.17	0.00		0.00	0.07
CC&DC	0.64	0.05	0.00	0.12	0.23	0.00	0.00		0.08
LC&CC&DC	1.29	0.02	0.01	0.01	0.04	0.26	0.10	0.13	

* NE - Classe não encontrada; SS - Classe sem *smell*; LC - Large Class; CC - Complex Class; DC - Duplicate Code.
* Os dados acima e abaixo da diagonal principal são percentuais que representam a proporção média das transições.

Analisando a Tabela 24, observamos que algumas transições ocorridas nos sistemas são mais predominantes do que outras. As transições mais representativas são: i) inclusão/criação de classes nos sistemas sem a existência de *smell* (NE → SS), essa transição representa 60,5% do total geral de transições; ii) 11,75% das transições são de classes sem

smell que foram removidas/migradas[22] dos sistemas (SS → NE); iii) por fim, a transição NE → DC representa 8,18% do total de transições; juntas elas representam aproximadamente 80% de todas as transições. Considerando as principais transições, podemos dizer que a maior parte das classes nascem sem *smell* e muitas classes (11,75%) sem *smell* são removidas. Entretanto, a maior parte das transições envolvendo *smell*, ocorre no nascimento das classes. Isso reforça o resultado anterior de que as classes com *smell*, já nascem com algum *smell* (78). Assim, considerando os *smells* estudados tanto nesse capítulo como no artigo (78), os *smells* Blob Class, Class Data Should be Private, Complex Class, Functional Decomposition, Spaghetti Code, Large Class, Complex Class, Duplicate Code têm muitas instâncias congênitas.

Para compreender melhor as transições de *smells*, vamos examinar os principais estados de *smell(s)* conforme a posição em que eles surgem nas transições, bem como a co-ocorrência deles. Nesse caso, examinamos os estados quando ele se encontra na posição *predecessora* e *sucessora*, exemplo: $Estado_{Predecessor} \rightarrow Estado_{Sucessor}$. Quando o estado analisado se encontra na posição *predecessora*, exploramos o que acontece com esse estado, por exemplo: se a classe sofre mudança do código que remove o *smell*. Por outro lado, na posição *sucessora*, investigamos de onde advêm esse estado, exemplo: *smell(s)* advêm das transições que criam/incluem classes nos sistemas.

❑ **Large Class (LC)**

Predecessor: Em termos numéricos, observamos que a remoção desse *smell* por mudança do código[23] é a transição mais frequente (LC → SS), contudo, representa apenas 0,30% do total geral de transições. Além disso, a próxima transição mais frequente também remove esse *smell*, entretanto, não o remove da classe, mas sim a própria classe (LC → NE). Por outro lado, encontramos algumas transições em que esse *smell* se torna uma co-ocorrência (LC → LC&CC, LC → LC&DC), indicando uma acentuação do fenômeno de *smells*. Entretanto, a transição desse *smell* para outro estado de *smell* em que ele não está envolvido (LC → CC, LC → DC), praticamente não acontece. Portanto, podemos dizer que o *smell* Large Class é mais propenso a desaparecer (removido por mudança do código e/ou migração).

Sucessor: A transição NE → LC é a que mais contribui (1,15% do total de transições) para a ocorrência desse *smell*, ou seja, muitas classes já nascem Large Class (considerando as classes Large Class da última versão do código, em média, 80,7% delas já nasceram com esse *smell*). Entretanto, a próxima transição que mais contribui para a ocorrência desse *smell* é aquela originada a partir de classes sem *smell* (SS → LC). Também notamos que as transições envolvendo a co-ocorrência de *smells* têm um papel marginal no surgimento desse *smell*, especialmente as transições: CC&DC → LC e LC&CC&DC → LC.

[22] Conforme apresentado, uma dada classe pode migrar de sistema.
[23] Nesse caso, a classe continua a existir no sistema mas sem o *smell* em questão

❑ **Complex Class (CC)**

Predecessor: Aqui observamos um comportamento semelhante àquele descrito no estado LC. Sendo a transição de remoção desse *smell* por mudança do código[23] (CC → SS — 0.48%) ou exclusão/migração de classes (CC → NE — 0.33%) são as que mais acontecem. Em termos de co-ocorrência de *smell*, o estado CC é mais propenso de vir a coexistir com o estado LC (CC → CC&LC). Por outro lado, quase não observamos a transformação do estado LC em outros completamente independentes (CC → LC, CC → DC, CC → LC&DC).

Sucessor: Novamente, as observações realizadas no estado LC também são válidas para este ponto. A diferença é que nesse ponto o estudo é focado no estado CC e, além disso, o percentual das transições é relativamente maior do que aqueles apresentado quando o estado alvo era LC (a transição $NE \rightarrow CC$ ocorre 21% mais do que $NE \rightarrow LC$; e $SS \rightarrow CC$ ocorre 40% mais do que a transição $SS \rightarrow LC$).

Em geral, podemos concluir que existe interesse por parte dos desenvolvedores em remover esse *smell* (CC), mesmo que algumas vezes a remoção desse *smell* implique na transformação dele em outro(s) completamente distinto(s) (LC, DC, LC&DC). Pela Tabela 24, se consideramos a coluna CC e apenas as linhas em que esse *smell* **não** aparece, ou seja, quando ele é removido, observamos que 0,83% das transições removem esse *smell*. Por outro lado, se fizer essa mesma análise da perspectivava de *sucessor* (linha CC por N colunas onde esse *smell* não ocorre), observamos que esse *smell* é introduzido por 2,34% das transições. Assim, a taxa de ocorrência desse *smell* é maior do a taxa de remoção, em particular, a maior parte das transições em que esse *smell* aparece ocorre no nascimento da classe ($NE \rightarrow CC$ — 1,47%). Portanto, é necessário melhorar o monitoramento/gerenciamento do surgimento desse *smell*, em particular, na criação de classes e/ou na transição SS → CC.

❑ **Duplicate Code (DC)**

Predecessor: Diferente do que observamos anteriormente e analisando o estado de destino das transições iniciadas no estado DC, as classes com o *smell* Duplicate Code são mais propensas de serem eliminadas do sistema (1,90%) do que sofrerem mudança do código[23] (1,41%). Além disso, em geral, o destino final de uma transição com o *predecessor* DC é a remoção do *smell* Duplicate Code. Esse comportamento é análogo ao observado nos estados *predecessores* LC e CC. Em termos de propensão, se o estado *predecessor* DC não for removido, então é provável que esse clone se junte/complemente com o estado CC ou LC. Formando assim, uma co-ocorrência dupla: CC&DC ou LC&DC.

Sucessor: A manifestação do estado DC é relativamente alta na criação das classes (8,18%) e, também, na transição SS → DC (1,68%). Além disso, notamos

que raramente os estados *predecessores* LC, CC, LC&CC, LC&CC%DC resultam em uma transição onde o *sucessor* é o estado DC.

Dentre os *smells* estudados, esse certamente é o mais introduzido[24] (9,87%), bem como aquele que mais se remove[25] (3.32%). Como descrito no estado LC, sugere-se o acompanhamento das classes em que o estado DC é introduzido na criação da classe, bem como nas situações em que uma classe sem *smell* se transforma em DC.

❑ **Co-ocorrência de Smell**

Predecessor: Considerando o estado *predecessor* LC&CC (co-ocorrência *predecessora* com os maiores percentuais), a maior parte das transições causam a remoção de classe (LC&CC → NE). Entretanto, se a remoção não acontecer, é provável que esse estado se combine com o estado DC, o que resulta no estado LC&CC&DC.

Sucessor: Considerando o estado *sucessor* LC&CC, 2,08% das transições resultam nesse estado no momento da criação das classes do sistema. A segunda grande fonte do estado *sucessor* LC&CC, são transições com o estado *predecessor* CC.

No geral, o estado *predecessor* LC&CC tem uma boa chance de resultar no estado *sucessor* LC&CC&DC. Entretanto, os estados *predecessores* LC&DC e CC&DC possuem chances equivalentes tanto para sofrer um *"downgrade"* (LC&DC → DC, CC&DC → DC) quanto para um *"upgrade"* (LC&DC → LC&CC&DC, CC&DC → LC&CC&DC). Por outro lado, o estado *predecessor* LC&CC&DC comumente realiza um *"downgrade"*, retornando para o estado LC&CC.

RQ2 — Resumo

Novamente constatamos que muitas classes com *smell* já nascem assim. Isso indica que a criação/inclusão das classes nos sistemas é um momento crítico, especialmente naqueles sistemas em que os engenheiros de software utilizam os *smells* como *proxy* da qualidade do código. Assim, nesses casos, sugerimos a necessidade de implementar políticas de controle de *smell(s)* no momento em que as classes são criadas, por exemplo: quando um arquivo de código fonte com algum *smell* é inserido no sistema, sinalizar aos engenheiros de software para examiná-lo. Também observamos que qualquer estado de *smell*, ocupando a posição *predecessora*, pode realizar tanto um *"downgrade"* como um *"upgrade"*. Entretanto, esse comportamento é menos frequente do que a mudança do código[23] e/ou exclusão/migração de classes. Portanto, a combinação gradual e acumulativa de *smells* pode ocorrer, mas esse evento não é a principal "fonte/causa" de classes com *smell*. Do ponto de vista da mudança do código[23] e/ou exclusão/migração de classes dos sistemas, alguns estados *predecessores*

[24] Transições que introduz DC: $\neg(DC \in estado) \rightarrow DC$
[25] Transições que remove DC: $(DC \in estado) \rightarrow \neg(DC \in estado) \vee (NE)$

simples (LC, CC) são mais suscetíveis à mudança do código[23] do que para a exclusão/migração de classes. Por outro lado, os estados *predecessores* LC&CC, LC&DC, CC&DC, LC&CC&DC (co-ocorrência) são mais suscetíveis à exclusão/migração de classes do que a mudança do código[23]. Isso pode ser um indício da necessidade de ferramentas que forneçam suporte à refatoração (mudança do código[23]) de classes que têm a co-ocorrência de *smells*.

RQ3 — Existe algum *smell* com comportamento cíclico, sejam eles isolados ou em co-ocorrência? Se sim, quais padrões se manifestam?

Essa RQ investiga possíveis movimentos cíclicos de *smell*, ocorridos entre uma faixa de janelas de *commit* (k_0 — k_n). Então o primeiro passo é definir objetivamente a ocorrência de um ciclo no contexto do nosso conjunto de dados. A ideia de ciclo se refere a recorrência de padrões ao longo do tempo, onde sempre se iniciam e terminam da mesma forma. No contexto desse estudo, procuramos por estados intermediários que se iniciam e terminam com o mesmo estado inicial. Adicionalmente, os estados intermediários devem ser formados por estados diferentes daqueles apresentados no inicio e/ou no final do ciclo, ou seja, eles não podem ser apenas uma cadeia de repetição do estado inicial e/ou final.

Para compreender a ideia dos ciclos, considere a análise de um sistema em um espaço de 10 janelas de *commits* ($k_{0...9}$). Nesse cenário, a classe `toy` foi analisada em cada janela de *commit* e se extraiu os estados dessa classe. No final, encontramos a seguinte configuração: . Observa-se que essa classe foi introduzida logo no início do sistema (k_8) e foi removida recentemente (k_1), dessa forma essa classe não existe nas janelas de *commits* que representam as versões mais recentes desse sistema (k_0 e k_1). Ao analisar apenas as janelas de *commits* em que essa classe existe ($k_{2...8}$), o estado se repete. Observe que entre uma repetição e outra, há estados intermediários que se diferem do estado inicial e/ou final. Portanto, o estado desencadeia dois ciclos de repetição, cada um com a seguinte cadeia de estados intermediários: . Conclui-se que para a ocorrência de um ciclo, é necessário haver duas transições de estados: i) *ahead* — o estado *predecessor* migra para outro estado qualquer (→); ii) *reverse* — mais adiante, um estado qualquer deve retornar ao estado *predecessor* (→). Salientamos que a cadeia de estados intermediários de um ciclo pode conter transições para outros estados. Entretanto, essas transições não envolvem o estado apresentado no início/final do ciclo, exemplo: .

O Algoritmo 4, descreve detalhadamente o procedimento de identificação dos ciclos de *smells*. Nele, para cada classe de um dado sistema, descartamos as janelas de *commits* em que a classe não existe (linhas 6 a 10). Na sequência, para cada estado analisado, identificamos a ocorrência da transição *ahead* (linha 16). Depois, identificamos o ponto que ocorre a transição *reverse* (linha 17 e 18). Seguindo, extraímos a cadeia de estados que compõem o estado intermediário do ciclo (linha 19) e relatamos a existência desse

Algoritmo 4: Estratégia de recuperação dos padrões cíclicos de *smells*.

```
1  GetCyclicPatterns (DataSet)
       input  : DataSet is the dataset collected (Projects, Classes and Status).
       output: Returns the cyclic patterns founded on each status.
2      Result ← HashMap<Status,HashMap<Pattern,HashMap<ClassType,Integer>>>();
3      foreach project ∈ DataSet do
4          foreach class ∈ project do
5              k ← 0; classtype ← ClassWasRemoved(class);      // return Removed or NotRemoved
6              for ; k < GetNumberOfWindows(class) ∧ GetStatus(class, k) ≡ NE; k++ do
7                  ⌊
8              FirstW ← k;
9              for ; k < GetNumberOfWindows(class) ∧ GetStatus(class, k) ≠ NE; k++ do
10                 ⌊
11             LastW ← k − 1;
12             class ← CutUnnecessaryWindow(class, FirstW, LastW);
13             foreach status ∈ {SS, LC, CC, DC, LC&CC, LC&DC, CC&DC, LC&CC&DC} do
14                 k ← 0;
15                 while k < GetNumberOfWindows(class) − 1 do
16                     StatusK ← GetStatus(class, k);
17                     StatusNext ← GetStatus(class, k + 1);
18                     if StatusK ≡ status ∧ StatusNext ≠ status then
19                         ProxK ← NextOccurrenceOfStatus(class, k + 1, status);
20                         if ProxK > k + 1 ∧ status ≡ GetStatus(class, ProxK) then
21                             pattern ← ExtractMiddlePatterns(class, k + 1, ProxK − 1);
22                             PutAndCountPatternEquals(Result, status, pattern, classtype);
23                             k ← ProxK;
24                             continue;
25                     k ← k + 1;
26     return Result;
```

ciclo de *smell* (linha 20). Por fim, para cada estado, esse algoritmo retorna uma lista contendo os ciclos de *smells*, as cadeias de estados intermediárias e a quantidade de vezes que os ciclos aconteceram por entre os sistemas analisados (linha 24).

A Tabela 25 demonstra os ciclos de *smells* mais recorrentes retornado pelo Algoritmo 4. A coluna `Estado` representa o estado inicial e final de um dado ciclo de *smell*, a coluna `Intermediários` são as cadeias de estados que ocorrem entre o estado inicial e final, as próximas colunas denotam a quantidade de vezes que um dado ciclo de *smell* ocorreu. Em geral, encontramos 1749 tipos distintos de ciclos de *smells*. Observamos que 26,9% (471) dos ciclos ocorrem tanto em classes que foram removidas do sistema, quanto em classes que permanecem no sistema até a última versão. Por outro lado, 73% (1278) dos ciclos ocorrem **apenas** nas classes que já foram removidas dos sistemas. Essa discrepância motivou a organização dessa tabela em dois grupos: i) *Both* — os ciclos que ocorrem em qualquer tipo de classe; ii) *OnlyOnRemoved* — os ciclos que ocorrem **apenas** nas classes que já foram removidas do sistema. Por limitação de espaço, para cada um desses grupos, essa tabela mostra apenas os 30 ciclos de *smells* mais recorrentes.

Tabela 25 – Representação dos *smells* cíclicos e seus estados intermediários.

Both: Ciclos ocorridos em qualquer tipo de classe						*OnlyOnRemoved*: Ciclos ocorridos apenas nas classes removidas					
ID	Estado[1]	Intermediários[2]	♠	♢	△	ID	Estado[1]	Intermediários[2]	♠	♢	△
01	CC		473	68	541	01	LC&CC&DC		0	10	10
02	LC		282	59	341	02	LC&CC&DC		0	8	8
03	CC		242	61	303	03	LC&CC&DC		0	8	8
04	LC&CC		189	56	245	04	CC		0	8	8
05	CC		191	52	243	05	LC&DC		0	7	7
06	DC		203	37	240	06	LC		0	7	7
07	LC&CC		171	66	237	07	LC		0	7	7
08	DC		183	49	232	08	CC		0	7	7
09	DC		167	39	206	09	LC&DC		0	6	6
10	CC&DC		144	55	199	10	LC&DC		0	6	6
11	LC		129	57	186	11	LC&DC		0	6	6
12	DC		133	51	184	12	LC&CC&DC		0	6	6
13	LC		136	40	176	13	LC&CC&DC		0	6	6
14	LC		126	41	167	14	LC&CC		0	6	6
15	LC&CC		121	41	162	15	LC		0	6	6
16	LC&DC		118	41	159	16	LC		0	6	6
17	CC		110	40	150	17	DC		0	6	6
18	CC&DC		113	36	149	18	DC		0	6	6
19	LC&CC		94	39	133	19	CC&DC		0	6	6
20	CC&DC		95	33	128	20	CC		0	6	6
21	LC&CC		83	43	126	21	CC		0	6	6
22	LC&CC&DC		76	44	120	22	LC&CC&DC		0	5	5
23	CC		74	39	113	23	LC&CC&DC		0	5	5
24	LC&DC		78	34	112	24	LC&CC&DC		0	5	5
25	LC		69	42	111	25	LC&CC&DC		0	5	5
26	CC		72	38	110	26	LC&CC&DC		0	5	5
27	DC		61	37	98	27	LC&CC&DC		0	5	5
28	CC		57	38	95	28	LC&CC		0	5	5
29	CC		62	31	93	29	LC&CC		0	5	5
30	LC&CC&DC		48	44	92	30	LC&CC		0	5	5

[1]Representa o estado inicial e final de um ciclo; [2]São as cadeias de estados intermediárias ocorridas e necessárias p/ completar um ciclo.

Qtde de ciclos ocorridos em: ♠ classes não removidas; ♢ classes removidas; △ todas as classes. Cada quadrado indica um estado de uma dada janela de *commit*:

CC - Complex Class | LC - Large Class | DC - Duplicate Code | SS - Sem Smell | CC&DC | LC&DC | LC&CC | LC&CC&DC

Conforme a Tabela 25, e considerando os dados do grupo *Both*, observa-se que o início/fim de diversos ciclos de *smell* são marcados pela ocorrência isolada de *smell*, sendo que eles ocorrem predominantemente nas classes não removidas dos sistemas. O ciclo mais frequente ocorreu 541 vezes, e caracterizado por apenas um estado intermediário (SS — *Sem Smell*) que se inicia/termina no estado que descreve o *smell* Complex Class. Esse ciclo ocorre majoritariamente (87,4%) nas classes não removidas do sistema. O segundo ciclo mais recorrente envolve o *smell* Large Class. Ele se repete 341 vezes, dessas, 82,6% acontecem em classes não removidas. O estado intermediário desse ciclo também é sinalizado pelo estado intermediário SS.

Em geral, os principais ciclos do grupo *Both* são caracterizados por: i) pequena cadeia de estados intermediário, sendo ela formada majoritariamente por um ou dois estados; ii) os ciclos ocorrem principalmente nas classes que não foram removidas dos sistemas; iii) boa parte dos ciclos passam pela refatoração/eliminação de *smell(s)*, exemplo: o ciclo mais recorrente, refatora o *smell* Complex Class, ou seja, determinadas classes passam um tempo (n janelas de *commits* — refletido pelo comprimento da cadeia de estados intermediários) sem um dado *smell*. Nesse contexto, podemos inferir que os desenvolvedores enfrentam dificuldades em manter o estado das classes refatoradas, ou seja, sem o *smell* que foi removido. Além disso, o pequeno comprimento da cadeia de estados intermediários pode indicar que os desenvolvedores têm uma percepção humana da existência desses *smells* e, geralmente, tentam refatorá-los.

Analisando o grupo *OnlyOnRemoved*, em comparação com os ciclos do grupo anterior, o número de ocorrência de cada ciclo é significativamente menor (o mais representativo se repetiu apenas 10 vezes). Por outro lado, esse grupo apresenta a maior diversidade de tipos distintos de ciclos (1278). Como esses ciclos ocorrem apenas em classes que foram removidas do sistemas, possivelmente eles podem ser usados para identificar se uma dada classe tem maior probabilidade de ser removida do sistema. Sugerimos que esses padrões podem auxiliar na priorização de *smells*, exemplo: uma ferramenta de detecção de *smell* que detecte algum desses ciclos em uma dada classe que apresente determinado *smell*, pode recomendar ao desenvolvedor que postergue a refatoração dessa classe, pois ela tem chances de ser removida. Portanto, o padrão desses ciclos pode auxiliar na priorização de *smells* e complementar a abordagem sugerida/discutida nos resultados da RQ1.

Nesse grupo, a principal particularidade da cadeia de estados intermediários se encontra no seu comprimento. Aqui, a cadeia é significativamente maior do que aquelas do grupo *Both*, em média formada por 8 janelas de *commits*. Além disso, observamos que a proporção entre os ciclos que envolve a refatoração ($LC\&CC\&DC \rightarrow CC\&DC \rightarrow LC\&CC\&DC$) e aquelas que abrangem a "inclusão" de *smell(s)* ($CC \rightarrow LC\&CC \rightarrow CC$) têm proporção numérica semelhante. Possivelmente, esse comportamento indica que as classes removidas são aquelas em que o controle de *smell(s)* é deficitário, gerando assim, alguns padrões que "incluem" *smell(s)* e outros que "removem".

Independente do grupo analisado (*Both*/*OnlyOnRemoved*), não encontramos cadeias de estados intermediários que, em algum momento, efetuem transições. Para visualizar, considere o seguinte exemplo hipotético da cadeia de estados intermediários:  , observe que existe uma transição do estado CC&DC para o estado DC e outra do estado DC para CC&DC. Na prática, isso significa que não há ciclos com estados intermediários muito variáveis, exemplo: $\overbrace{\texttt{LC\&CC\&DC}}^{\texttt{início}} \rightarrow \overbrace{\texttt{CC\&DC, CC\&DC, DC, CC\&DC}}^{\texttt{cadeia de estados intermediários}} \rightarrow \overbrace{\texttt{LC\&CC\&DC}}^{\texttt{fim}}$. Entretanto, a Tabela 24 da RQ2 demonstra que transições bruscas ($LC\&CC\&DC \rightarrow DC$) podem ocorrer. Assim, concluí-se que elas não ocorrem dentro de estados cíclicos.

RQ3 — Resumo

Demonstramos que existem *smells* com comportamento cíclico. Além disso, observamos que os *smells* cíclicos das classes que foram removidas dos sistemas apresentam características diferentes daquelas observadas nas classes que não foram removidas, exemplo: o comprimento da cadeia de estados intermediários é maior nos *smells* cíclicos das classes removidas. Argumentamos que isso pode ser usado pelas ferramentas de detecção de *smells* para priorizar a refatoração/monitoramento das classes, exemplo: uma ferramenta de detecção de *smell* que detecte algum dos ciclos de *smells* ocorridos nas classes removidas, pode recomendar ao desenvolvedor que postergue a refatoração dessa classe pois ela tem chances de ser removida.

5.3 Limitações e Ameaças à Validade

As principais ameaças estão relacionadas à possíveis divergências entre a observação teórica e prática, isso se deve especialmente à imprecisões/erros implícitos no processo de detectar uma dada instância de *smell*. Portanto, estamos cientes de que nossos resultados podem ser afetados por: i) *thresholds* internos usados pelas ferramentas de detecção de *smells* e ii) a presença de falsos positivos e falsos negativos. No caso da ferramenta PMD, na subseção 4.1.3, definimos o *threshold* igual a 75, pois empiricamente ele se mostrou mais realista. Por outro lado, a implementação da ferramenta DECOR apresenta até 88% de *Precision* e 100% de *Recall* (4). Além disso, a implementação usada para rastrear as classes por entre os diversos *commits* (`RefactoringMiner`), também possui limitações semelhantes. Entretanto, segundo Tsantalis et al. (151), os índices de *Precision* (98%) e *Recall* (87%) são consideravelmente elevados e significativamente melhores do que outras ferramentas (`RefDiff`) consideradas estado da arte.

Alguns estudos também fazem uma validação manual das instâncias de *smells*, que foram automaticamente detectadas pelas ferramentas computacionais (78, 146). Eles, geralmente adotam uma abordagem amostral, ou seja, um subconjunto das instâncias de *smells* que passam pela avaliação humana que aponta se elas realmente são instâncias de *smells*. Entretanto essa estratégia não elimina o componente relacionado à subjetividade

humana e, por isso, o processo também é impreciso. Além disso, em nosso estudo, devido ao volume de dados (investigação de nove estados distintos por entre 318,887 mil classes distribuídas em 89 sistemas e organizadas em 100 janelas de *commits*), essa amostragem deveria ser relativamente grande. Assim, devido à limitação de tempo para manualmente avaliar todas as instâncias de *smells* que estudamos e, ainda, devido à subjetividade humana, confiamos nos resultados das ferramentas usadas nesse estudo empírico.

Uma limitação desse estudo está relacionada à generalização das observações, em particular, referente ao grau de generalização dos achados. No experimento relatado, essas ameaças são mitigadas pelo fato de termos utilizado sistemas com diferentes tamanhos, propósitos e domínios. Sendo que esses foram implementados usando diferentes estilos de *design*. Além disso, os sistemas alvo foram desenvolvidos por equipes de diferentes tamanhos.

O planejamento experimental apresentado neste estudo também apresenta certas limitações. Algumas estão relacionadas ao conjunto de dados analisado e outras são referentes ao processo de análise dos dados. As limitações do primeiro caso, são decorrentes das regras usadas para selecionar os repositórios, exemplo: sistema deve conter no mínimo 500 *commits* (ver Subseção 5.1.1.1). Entretanto, como o principal foco deste estudo é a investigação da cronologia de *smells*, as regras são necessárias para minimizar possíveis inconsistências, exemplo: sistemas pequenos e/ou com poucos *commits* podem induzir o pensamento de que *smells* não são refatorados, talvez porque os desenvolvedores ainda não tiveram tempo para removê-los. Por outro lado, o processo de análise se limita aos *commits* da ramificação principal do sistema de controle de versão. Além disso, a análise é realizada usando o conceito de janela de *commit* (ver Subseção 5.1.1.2), ou seja, apenas um subconjunto de *commits* da ramificação principal passa pelo processo de análise. Essas são estratégias comumente empregadas (158), exemplo: Robles et al. (159) coletam dados apenas dos sistemas que têm ramificação principal denominada como *master*. Em nosso estudo, essas estratégias foram necessárias para: i) mitigar a coleta de dados irrelevantes, como por exemplo, *commits* que não fazem parte do sistema principal (ex. *fork*); ii) a janela de *commits*, mitiga a diferença de longevidade entre os sistemas ($\geq$ 5 anos), pois sua representação cronológica é feita distribuindo-se os *commits* em faixas de percentuais (ver Subseção 5.1.1.3).

5.4 Considerações Finais

Nesse capítulo, apresentamos um estudo empírico em larga escala da cronologia de *smells*. O estudo foi realizado usando o histórico de *commits* de 89 sistemas de código fonte aberto. A análise da cronologia de *smell*, que é caracterizada pelo rastreamento da ancestralidade de uma dada entidade que apresenta algum *smell*, visa compreender: i) quando os *smells* ocorrem; ii) se a simples ocorrência de *smell* se torna uma co-ocorrência

de *smell*; iii) apontar o comportamento cíclico dos *smells* (ocorre quando uma instância de *smell* é removida, mas por algum motivo, ela retorna à classe). Os resultados apontam várias descobertas valiosas para a comunidade de pesquisa:

- ❏ Em geral, observamos que em relação ao total de classes, a proporção de classes com algum *smell* é maior apenas no início do sistema. Em outras palavras, à medida que o sistema cresce (número de classes), a proporção de classes afetadas por algum *smell* também cresce, mas em uma taxa menor. Isso implica que, de alguma forma, à medida que o sistema cresce, os desenvolvedores tendem a controlar a proliferação de *smells*.

- ❏ Ao considerar apenas classes com algum *smell*, todos os *smells* analisados, bem como a co-ocorrência deles, têm uma relação linear que cresce à medida que o tempo passa. Entretanto, cada estado de *smell* tem seu próprio ritmo de crescimento. Também observamos que muitas classes deixam de existir nos sistemas, isso ocorre por duas razões: i) classes são removidas/excluídas; ii) classes são movidas para outros repositórios. Isso tem impacto direto na gerência do código fonte, especialmente no aspecto relacionado às ferramentas que auxiliam os desenvolvedores e que não consideram esse fenômeno.

- ❏ Apresentamos indícios de que a métrica *"estabilidade média das transições"* pode ser usada para separar (clusterizar) as classes em dois grupos: *Não Removidas* ou *Removidas*. Permitindo, por exemplo, a produção de ferramentas que indiquem quais classes podem passar por operações de reestruturação que culminem na sua remoção, reduzindo assim o retrabalho. Além disso, apresentamos que, no início do sistema, certos estados são mais propensos a sofrer alterações que causam transição de estados. Possibilitando, por exemplo, priorizar a revisão de código das classes que têm esses estados.

- ❏ Confirmamos que a maior parte das classes com algum *smell* já nascem nessa condição (78%). Além disso, observamos que muitas dessas classes mantêm na última versão do código fonte o mesmo *smell* observado no seu nascimento. Também observamos que as classes com certas co-ocorrências de *smells* congênitos, raramente terminam sem *smell*. Indicando, por exemplo, o desconhecimento (ao menos formal — emitido por alguma ferramenta que fornece suporte ao desenvolvimento) desse tipo de co-ocorrência de *smell*. Deste modo, como os desenvolvedores não possuem a real magnitude desse tipo de situação, os mesmos não podem se esforçar para manter a situação controlada. Isso é reforçado pelo fato de que não se tem conhecimento de ferramentas de detecção de *smell* que reportem a co-ocorrência de *smells*.

- ❏ Observamos que qualquer estado de *smell*, ocupando a posição *predecessora*, pode realizar tanto um *"downgrade"* como um *"upgrade"*. Entretanto, esse comportamento

é menos frequente do que a mudança do código e/ou exclusão/migração de classes. Portanto, a combinação gradual e acumulativa de *smells* pode ocorrer, mas esse evento não é a principal "fonte/causa" de classes com *smell*. Além disso, observamos que alguns estados *predecessores* simples (LC, CC) são mais suscetíveis à mudança do código do que para a exclusão/migração de classes. Por outro lado, certos estados *predecessores* de co-ocorrência de *smell* são mais suscetíveis à exclusão/migração de classes do que a mudança do código. Isso reforça a necessidade de ferramentas que forneçam suporte à refatoração de classes que têm a co-ocorrência de *smells*.

- Demonstramos que existem *smells* com comportamento cíclico. Além disso, observamos que os *smells* cíclicos das classes que foram removidas dos sistemas apresentam características diferentes daquelas observadas nas classes que não foram removidas, exemplo: o comprimento da cadeia de estados intermediários é maior nos *smells* cíclicos das classes removidas. Argumentamos que isso pode ser usado pelas ferramentas de detecção de *smells* para priorizar a refatoração/monitoramento das classes, exemplo: uma ferramenta de detecção de *smell* que detecte algum dos ciclos de *smells* ocorridos nas classes removidas, pode recomendar ao desenvolvedor que postergue a refatoração dessa classe, pois ela tem chances de ser removida.

As descobertas apresentadas permeiam nossa agenda de futuras pesquisas no tema *bad smell*, focando principalmente em projetar, desenvolver e testar uma nova geração de verificadores da qualidade do código. A exemplo, desejamos construir um classificador que use a métrica de estabilidade média. Além disso, desejamos complementar as ferramentas de detecção de *smell*, integrando nosso classificador e aplicando os padrões acima para priorizar as operações de reestruturação e/ou refatoração.

Capítulo 6

Conclusão

Os sistemas de software precisam evoluir continuamente para lidar com as solicitações de novos requisitos e constantes mudanças de ambiente, bem como corrigir possíveis *bugs*. Nesse contexto, códigos de alta qualidade desempenham um importante papel, pois são fáceis de entender, analisar, modificar, manter e reutilizar (160). Contudo, os desenvolvedores eventualmente produzem códigos subótimos, possivelmente introduzindo problemas de projeto, exemplo: produzem estruturas de código que desrespeitam princípios fundamentais da engenharia de software, como alta coesão e baixo acoplamento. Nesse caso, podem introduzir *bad smells*[1] (15) que tendem a impactar negativamente no reuso e na decomposição modular do sistema (161), acarretando o declínio da qualidade do software e eventualmente aumentam a complexidade do código de forma desnecessária.

Nesse contexto, estudos anteriores relatam que até 80% do custo total do software é destinado ao financiamento das atividades de manutenção e evolução (10). Além disso, aproximadamente 60% do tempo total gasto para realizar a manutenção do software é utilizado no entendimento do código fonte (10). Isso é um dos fatores que estimula a utilização dos *bad smells* como indicadores da necessidade de refatorar[2] o código da aplicação (15) e, assim, manter/melhorar a qualidade do código. Por isso, neste trabalho de doutorado realizamos um estudo empírico exploratório de alguns aspectos relacionados ao tema *bad smell*.

O tema *bad smell* é extenso e complexo, pois, uma vez decidido usá-los para melhorar a qualidade do código, em geral, se tem a necessidade de: (i) identificar onde os diversos tipos de *smells* ocorrem no código; (ii) decidir quais desses *smells* são relevantes e devem ser refatorados; (iii) determinar como a tarefa de refatorar cada um deve ser executada; (iv) estabelecer quando a tarefa de refatoração deve ocorrer; (v) atribuir a missão de refatoração à equipe de desenvolvedores mais adequada.

Assim, neste trabalho de doutorado, para identificar quais *smells* e quais aspectos

[1] Referenciadas na literatura como: *Bad Smells, Code Smells, design defects ou anti-patterns.*

[2] Processo de melhorar a estrutura interna do código sem alterar o comportamento externo da aplicação (15).

desse tema necessita de estudo empírico, inicialmente realizamos uma extensa revisão sistemática da literatura. Nossa revisão sistemática considerou artigos publicados em 14 conferências e 6 revistas. Os veículos de publicação que consideramos são conferências e revistas de grande importância na área de engenharia de software (ex. ICSE). Desses veículos, coletamos e analisamos diversos artigos publicados sobre o tema *bad smell*. A revisão considerou publicações dos últimos 27 anos (1990 — 2017), resultando em um total de 351 artigos sobre o tema. Esse levantamento possibilitou identificar diversos horizontes para futuras pesquisas (ex. necessidade de *benchmark*, uso de métricas evolucionárias na detecção de *smell*, ...) (39). Com base nos resultados da revisão, o horizonte de pesquisa que decidimos investigar foi a possível interação de múltiplos *smells*. Nesse caso, a literatura apresenta alguns estudos (27), mas estes são limitados a determinados *smells* e geralmente não investigam detalhadamente cada uma das interações (30). Além disso, também observamos que possíveis interações de *smells* apontados informalmente na literatura tradicional (Fowler e Beck (15)) não haviam sido formalmente investigadas em estudos empíricos, neste caso, a possível relação semântica entre Duplicate Code, Large Class e Complex Class. Intuitivamente conjectura-se que entidades complexas, com muitas estruturas de desvio de fluxo, têm maior chance de apresentar códigos clonados; por outro lado, estruturas grandes são intuitivamente mais complexas. Considerando que estes *smells* (DC, LC, CC) são recorrentes no código fonte dos projetos e, ainda, que estes *smells* foram amplamente estudados, mas não pela perspectiva da possível interação entre eles, decidimos realizar neste trabalho um estudo empírico exploratório sobre a interação desses *smells*.

A investigação empírica considerou a interação entre o *smells*: (i) a ocorrência de clones nas classes acometidas apenas pelo *smell* Large Class; (ii) de forma análoga, também consideramos os clones das classes onde o *smell* Complex Class ocorre de forma isolada; (iii) além disso, examinamos a associação da prevalência de clones nas classes onde os *smell* Large Class e Complex Class co-existem simultaneamente. Também investigamos como a intensidade dos *smells* LC e/ou CC influencia a prevalência de clones.

Os dados do estudo revelaram que, em alguns sistemas, a complexidade da classe é um fator muito mais associado à prevalência de clones do que ao tamanho da classe propriamente dita. Também mostramos que apenas o tamanho da classe não é significativamente associado à prevalência de clones. Contudo, quando uma classe apresenta a co-ocorrência dos *smells* Large & Complex Class, as chances de prevalência dos clones aumentam significativamente. Adicionalmente, mostramos que a intensidade dos *smells* LC e CC, bem como a forma com que eles ocorrem nas entidades, influenciam no tipo de clone (Intra-, Inter-, Mix-classe) que mais prevalece nas classes. Mostramos que os fatores analisados explicam no máximo 11% da prevalência de clones, indicando que há outros fatores associados aos clones que devem ser estudados em outros trabalhos.

Por fim, também apresentamos algumas possíveis implicações práticas dos resultados do estudo empírico.

O estudo empírico que considera a prevalência de clones e a intensidade dos *smells*: Duplicate Code (DC), Large Class (LC) e Complex Class (CC), levanta questões relacionadas à temporalidade dos *smells*. Isso porque o estudo do Capítulo 4, não investiga como e quando certos padrões da inter-relação dos *smells* foram introduzidos no código fonte. Assim, no Capítulo 5, estudamos a cronologia desses *smells*. A cronologia de *smells* é caracterizada pelo rastreamento da ancestralidade de uma dada entidade que apresenta algum *smell*. Os resultados indicam que, após um certo período inicial de amadurecimento dos sistemas, a proporção de classes sem *smell* é maior do que a quantidade com algum *smell*. Além disso, ao longo do ciclo de vida dos sistemas, diversas classes são excluídas e/ou migram de repositórios. Como apresentado, isso tem impacto nas ferramentas que auxiliam os desenvolvedores, pois elas não consideram esse fenômeno. Também demonstramos a viabilidade de usar a métrica *"estabilidade média das transições"* para separar (clusterizar) as classes que *"Serão Removidas"* daquelas que *"Não Serão Removidas"*, permitindo, por exemplo, a produção de ferramentas que indiquem quais classes podem passar por operações de reestruturação. Por fim, apresentamos algumas situações de *smells* com comportamento cíclico e que, raramente, a simples ocorrência de um dado *smell* se torna uma co-ocorrência.

Os resultados apresentados nesta tese revelaram achados que complementam o estado da arte da literatura. Portanto, destacamos que os achados listados neste estudo têm impacto na área de *bad smells*, em particular no desenvolvimento de uma nova geração de ferramentas verificadoras da qualidade do código fonte. Além disso, conforme o enunciado de tese, verificamos que a presença dos *smells* descreve certos padrões de codificação, e esses contém informação relevante que poderia ser usada pelos engenheiros de software para controlar a qualidade do código.

6.1 Próximos Passos da Pesquisa

As atividades descritas nos capítulos anteriores foram elaboradas e executadas com o intuito de evidenciar os primeiros resultados desta tese de doutorado. Em especial, para esta banca, o material apresentado tem como objetivo: (i) demonstrar a capacidade de aplicação da metodologia adequada à pesquisa; (ii) manifestar a habilidade de formulação de hipóteses e de desenvolvimento do raciocínio lógico; (iii) evidenciar os resultados empíricos preliminares.

A continuação desta pesquisa de doutorado, passa pela investigação dos aspectos descritos abaixo:

- ❑ **Granularidade dos *Smells*.** Atualmente este estudo analisa os *smells* relacionados à complexidade (CC) e/ou tamanho (LC) na granularidade de classe, mas

os clones são detectados na granularidade de *tokens* (medida semelhante a LOC). Possivelmente refazer a mesma análise detalhada neste capítulo no nível de método e/ou blocos de linhas de código, pode produzir resultados diferentes. Isso porque é mais comum refatorar/alterar um bloco de linhas de código e/ou um método do que efetivamente toda a classe. Em termos de contribuição, essa abordagem pode revelar padrões escondidos que são úteis para priorizar a refatoração.

- ❑ **Implementação do Classificador.** Construir um classificador que use a métrica de *"estabilidade média das transições"*. Além disso, desejamos complementar as ferramentas de detecção de *smell*, integrando nosso classificador e aplicando os padrões apontados nos Capítulo 4 e 5 para priorizar as operações de reestruturação e/ou refatoração. Cabe ressaltar que a construção de um classificador requer um estudo amplo, o qual a métrica usada é analisada sob diversas abordagens a fim de identificar a melhor forma de aplicar, bem como apontar suas restrições. Por exemplo, identificando como o cálculo da métrica deve ser ajustado para diferentes tipos de sistemas, exemplo hipotético: usar uma média móvel simples para sistemas com janelas de *commits* que tenham poucos *commits* ou, ainda, usar uma média móvel exponencial para sistemas com janelas de *commits* que tenham muitos *commits*.

6.2 Produção Bibliográfica

No presente estado, este trabalho de pesquisa já demonstra algumas contribuições bibliográficas, a saber: artigo submetido e aceito em periódico (*TSE*) e conferência (*ICSE*). Além disso, os resultados dos Capítulos 4 e 5 estão em fase de submissão.

- ❑ PAULO SOBRINHO, E.; DE LUCIA, A.; MAIA, M.; A systematic literature review on bad smells — 5 W's: which, when, what, who, where — Extended Abstract. **accepted for presentation at ACM/IEEE International Conference on Software Engineering**. January 2019.

- ❑ PAULO SOBRINHO, E.; DE LUCIA, A.; MAIA, M.; A systematic literature review on bad smells — 5 W's: which, when, what, who, where. **accepted for publication in IEEE Transactions on Software Engineering**. November 2018.

Referências

1 MARINESCU, R. **Assessing and Improving Object-Oriented Design**. Tese (Doutorado) — Politehnica University of Timisoara, 2012.

2 MARINESCU, R. Detection strategies: metrics-based rules for detecting design flaws. In: **20th IEEE International Conference on Software Maintenance, 2004. Proceedings.** [S.l.]: IEEE, 2004. p. 350–359. <https://doi.org/10.1109/ICSM.2004.1357820>.

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

4 MOHA, N. et al. DECOR: A Method for the Specification and Detection of Code and Design Smells. **IEEE Transactions on Software Engineering**, v. 36, n. 1, p. 20–36, jan 2010. <https://doi.org/10.1109/TSE.2009.50>.

5 KHOMH, F. et al. A Bayesian Approach for the Detection of Code and Design Smells. In: **Quality Software, 2009. QSIC '09. 9th International Conference on**. [S.l.: s.n.], 2009. p. 305–314. <https://doi.org/10.1109/QSIC.2009.47>.

6 LILIENTHAL, C. From pair programming to mob programming to mob architecting. In: DIETMAR et al. (Ed.). **Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies: 9th International Conference, SWQD 2017, Vienna, Austria, January 17-20, 2017, Proceedings**. Cham: Springer International Publishing, 2017. p. 3–12. <https://doi.org/10.1007/978-3-319-49421-0_1>.

7 SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. **Refactoring for Software Design Smells: Managing Technical Debt**. [S.l.]: Elsevier Science, 2014.

8 SEAMAN, C.; GUO, Y. Chapter 2 - measuring and monitoring technical debt. In: ZELKOWITZ, M. V. (Ed.). [S.l.]: Elsevier, 2011, (Advances in Computers, v. 82). p. 25 – 46.

9 BAVOTA, G.; RUSSO, B. A large-scale empirical study on self-admitted technical debt. In: **Proceedings of the 13th International Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2016. (MSR '16), p. 315–326. <https://doi.org/10.1145/2901739.2901742>.

10 APRIL, A.; ABRAN, A. A software maintenance maturity model (s3m): Measurement practices at maturity levels 3 and 4. **Electronic Notes in Theoretical Computer Science**, v. 233, n. 0, p. 73 – 87, 2009.

11 ALVES, N. S. R. et al. Towards an ontology of terms on technical debt. In: **2014 Sixth International Workshop on Managing Technical Debt**. [S.l.: s.n.], 2014. p. 1–7. <https://doi.org/10.1109/MTD.2014.9>.

12 OLIVETO, R. et al. Identifying method friendships to remove the feature envy bad smell. In: **Proceeding of the 33rd international conference on Software engineering - ICSE '11**. New York, New York, USA: ACM Press, 2011. p. 820. <https://doi.org/10.1145/1985793.1985913>.

13 OIZUMI, W. et al. When Code-Anomaly Agglomerations Represent Architectural Problems? An Exploratory Study. In: **Software Engineering (SBES), 2014 Brazilian Symposium on**. [S.l.: s.n.], 2014. p. 91–100. <https://doi.org/10.1109/SBES.2014.18>.

14 TABA, S. E. S. et al. Predicting Bugs Using Antipatterns. In: **2013 IEEE International Conference on Software Maintenance**. [S.l.]: IEEE, 2013. p. 270–279. <https://doi.org/10.1109/ICSM.2013.38>.

15 FOWLER, M.; BECK, K. **Refactoring: Improving the Design of Existing Code**. [S.l.]: Addison-Wesley, 1999. (Object Technology Series).

16 BROWN, W. J. et al. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. New York, NY, USA: John Wiley & Sons, Inc., 1998.

17 AIKO, Y.; LEON, M. To What Extent Can Maintenance Problems Be Predicted by Code Smell Detection? - An Empirical Study. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 55, n. 12, p. 2223–2242, 2013. <https://doi.org/10.1016/j.infsof.2013.08.002>.

18 LI, W.; SHATNAWI, R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. **Journal of Systems and Software**, v. 80, n. 7, p. 1120–1128, jul 2007. <https://doi.org/10.1016/j.jss.2006.10.018>.

19 MACIA, I. et al. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In: **2012 16th European Conference on Software Maintenance and Reengineering**. [S.l.]: IEEE, 2012. p. 277–286. <https://doi.org/10.1109/CSMR.2012.35>.

20 GURP, J. V.; BOSCH, J. Design erosion: problems and causes. **Journal of Systems and Software**, v. 61, n. 2, p. 105 – 119, 2002.

21 FERME, V.; MARINO, A.; FONTANA, F. A. Is it a Real Code Smell to be Removed or not? In: **International Workshop on Refactoring & Testing (RefTest), co-located event with XP 2013 Conference**. [S.l.: s.n.], 2013.

22 RAHMAN, F.; BIRD, C.; DEVANBU, P. Clones: what is that smell? **Empirical Software Engineering**, Springer US, v. 17, n. 4-5, p. 503–530, 2012. <https://doi.org/10.1007/s10664-011-9195-3>.

23 VAUCHER, S. et al. Tracking Design Smells: Lessons from a Study of God Classes. In: **2009 16th Working Conference on Reverse Engineering**. [S.l.]: IEEE, 2009. p. 145–154. <https://doi.org/10.1109/WCRE.2009.23>.

24 LI, Z. et al. CP-Miner: finding copy-paste and related bugs in large-scale software code. **IEEE Transactions on Software Engineering**, v. 32, n. 3, p. 176–192, mar 2006. <https://doi.org/10.1109/TSE.2006.28>.

25 HUMMEL, B. et al. Index-based code clone detection: incremental, distributed, scalable. In: **2010 IEEE International Conference on Software Maintenance**. [S.l.]: IEEE, 2010. p. 1–9. <https://doi.org/10.1109/ICSM.2010.5609665>.

26 MACIA, I. et al. Supporting the identification of architecturally-relevant code anomalies. In: **Software Maintenance (ICSM), 2012 28th IEEE International Conference on**. [S.l.: s.n.], 2012. p. 662–665. <https://doi.org/10.1109/ICSM.2012.6405348>.

27 OIZUMI, W. N. **Synthesis of Code Anomalies: Revealing Design Problems in the Source Code**. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015.

28 MOHA, N. et al. A Domain Analysis to Specify Design Defects and Generate Detection Algorithms. In: **Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering**. Berlin, Heidelberg: Springer-Verlag, 2008. (FASE'08/ETAPS'08), p. 276–291.

29 CAI, D.; KIM, M. An Empirical Study of Long-Lived Code Clones. In: GIANNAKOPOULOU, D.; OREJAS, F. (Ed.). **Fundamental Approaches to Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6603). p. 432–446. <https://doi.org/10.1007/978-3-642-19811-3_30>.

30 LIU, H. et al. Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort. **IEEE Transactions on Software Engineering**, v. 38, n. 1, p. 220–235, jan 2012. <https://doi.org/10.1109/TSE.2011.9>.

31 LEHMAN, M. M. On understanding laws, evolution, and conservation in the large-program life cycle. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 1, p. 213–221, set. 1984. <https://doi.org/10.1016/0164-1212(79)90022-0>.

32 PALOMBA, F. et al. Anti-pattern detection: Methods, challenges, and open issues. **Advances in Computers**, v. 95, p. 201–238, 2015.

33 KHOMH, F. et al. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness. **Empirical Softw. Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 17, n. 3, p. 243–275, 2012. <https://doi.org/10.1007/s10664-011-9171-y>.

34 ABBES, M. et al. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In: **2011 15th European Conference on Software Maintenance and Reengineering**. [S.l.]: IEEE, 2011. p. 181–190. <https://doi.org/10.1109/CSMR.2011.24>.

35 YAMASHITA, A.; MOONEN, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: **2013 35th International Conference on Software Engineering (ICSE)**. [S.l.]: IEEE, 2013. p. 682–691. <https://doi.org/10.1109/ICSE.2013.6606614>.

36 ARCOVERDE, R.; GARCIA, A.; FIGUEIREDO, E. Understanding the longevity of code smells: Preliminary results of an explanatory survey. In: **Proceedings of the 4th Workshop on Refactoring Tools**. New York, NY, USA: ACM, 2011. (WRT '11), p. 33–36. <https://doi.org/10.1145/1984732.1984740>.

37 CHATZIGEORGIOU, A.; MANAKOS, A. Investigating the Evolution of Bad Smells in Object-Oriented Code. In: **Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the**. [S.l.: s.n.], 2010. p. 106–115. <https://doi.org/10.1109/QUATIC.2010.16>.

38 PALOMBA, F. et al. Detecting bad smells in source code using change history information. In: **2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.]: IEEE, 2013. p. 268–278. <https://doi.org/10.1109/ASE.2013.6693086>.

39 SOBRINHO, E. V. d. P.; LUCIA, A. D.; MAIA, M. d. A. A systematic literature review on bad smells — 5 w's: which, when, what, who, where. **IEEE Transactions on Software Engineering**, p. 1–1, 2018. ISSN 0098-5589. <https://doi.org/10.1109/TSE.2018.2880977>.

40 BIAN, Y. et al. SPAPE: A semantic-preserving amorphous procedure extraction method for near-miss clones. **Journal of Systems and Software**, v. 86, n. 8, p. 2077–2093, aug 2013. <https://doi.org/10.1016/j.jss.2013.03.061>.

41 SAHA, R. K. et al. Understanding the evolution of Type-3 clones: An exploratory study. In: **2013 10th Working Conference on Mining Software Repositories (MSR)**. [S.l.]: IEEE, 2013. p. 139–148. <https://doi.org/10.1109/MSR.2013.6624021>.

42 GÖDE, N.; KOSCHKE, R. Incremental Clone Detection. In: **2009 13th European Conference on Software Maintenance and Reengineering**. [S.l.]: IEEE, 2009. p. 219–228. <https://doi.org/10.1109/CSMR.2009.20>.

43 GERMAN, D. M. et al. Code siblings: Technical and legal implications of copying code between applications. In: **2009 6th IEEE International Working Conference on Mining Software Repositories**. [S.l.]: IEEE, 2009. p. 81–90. <https://doi.org/10.1109/MSR.2009.5069483>.

44 BRIXTEL, R. et al. Language-Independent Clone Detection Applied to Plagiarism Detection. In: **2010 10th IEEE Working Conference on Source Code Analysis and Manipulation**. [S.l.]: IEEE, 2010. p. 77–86. <https://doi.org/10.1109/SCAM.2010.19>.

45 MÄNTYLÄ, M. V.; LASSENIUS, C. Subjective evaluation of software evolvability using code smells: An empirical study. **Empirical Software Engineering**, v. 11, n. 3, p. 395–431, may 2006. <https://doi.org/10.1007/s10664-006-9002-8>.

46 MANTYLA, M.; VANHANEN, J.; LASSENIUS, C. Bad smells -humans as code critics. In: **20th IEEE International Conference on Software Maintenance, 2004. Proceedings.** [S.l.]: IEEE, 2004. p. 399–408. <https://doi.org/10.1109/ICSM.2004.1357825>.

47 KHOMH, F.; Di Penta, M.; GUEHENEUC, Y.-G. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In: **2009 16th Working Conference on Reverse Engineering**. [S.l.]: IEEE, 2009. p. 75–84. <https://doi.org/10.1109/WCRE.2009.28>.

48 PALOMBA, F. et al. 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.]: IEEE, 2014. p. 101–110. <https://doi.org/10.1109/ICSME.2014.32>.

49 JAAFAR, F. et al. Mining the relationship between anti-patterns dependencies and fault-proneness. In: **2013 20th Working Conference on Reverse Engineering (WCRE)**. [S.l.]: IEEE, 2013. p. 351–360. <https://doi.org/10.1109/WCRE.2013.6671310>.

50 MCCABE, T. A complexity measure. **Software Engineering, IEEE Transactions on**, SE-2, n. 4, p. 308–320, Dec 1976. <https://doi.org/10.1109/TSE.1976.233837>.

51 OLBRICH, S. M.; CRUZES, D. S.; SJOBERG, D. I. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In: **2010 IEEE International Conference on Software Maintenance**. [S.l.]: IEEE, 2010. p. 1–10. <https://doi.org/10.1109/ICSM.2010.5609564>.

52 FONTANA, F. A. et al. Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains. In: **2013 IEEE International Conference on Software Maintenance**. [S.l.]: IEEE, 2013. p. 260–269. <https://doi.org/10.1109/ICSM.2013.37>.

53 MARINESCU, R.; MARINESCU, C. Are the Clients of Flawed Classes (Also) Defect Prone? In: **2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation**. [S.l.]: IEEE, 2011. p. 65–74. <https://doi.org/10.1109/SCAM.2011.9>.

54 CARNEIRO, G. d. F. et al. Identifying code smells with multiple concern views. In: **2010 Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2010. p. 128–137. <https://doi.org/10.1109/SBES.2010.21>.

55 CHATZIGEORGIOU, A.; MANAKOS, A. Investigating the evolution of code smells in object-oriented systems. **Innovations in Systems and Software Engineering**, v. 10, n. 1, p. 3–18, 2014.

56 HECHT, G. et al. Tracking the software quality of android applications along their evolution (t). In: **Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. Washington, DC, USA: IEEE Computer Society, 2015. (ASE '15), p. 236–247. <https://doi.org/10.1109/ASE.2015.46>.

57 JAAFAR, F. et al. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. **Empirical Software Engineering**, v. 21, n. 3, p. 896–931, 2016. <https://doi.org/10.1007/s10664-015-9361-0>.

58 FONTANA, F. A. et al. Comparing and experimenting machine learning techniques for code smell detection. **Empirical Software Engineering**, v. 21, n. 3, p. 1143–1191, 2016. <https://doi.org/10.1007/s10664-015-9378-4>.

59 MEYERS, T. M.; BINKLEY, D. An empirical study of slice-based cohesion and coupling metrics. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 17, n. 1, p. 2:1–2:27, dez. 2007. <https://doi.org/10.1145/1314493.1314495>.

60 MARTIN, R. C. **Agile Software Development: Principles, Patterns, and Practices**. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

61 BAVOTA, G. et al. An experimental investigation on the innate relationship between quality and refactoring. **Journal of Systems and Software**, v. 107, p. 1 – 14, 2015. <https://doi.org/10.1016/j.jss.2015.05.024>.

62 CHACON, S. **Pro Git**. 1st. ed. Berkely, CA, USA: Apress, 2009.

63 ROONEY, G. **Practical Subversion**. [S.l.]: Apress, 2008.

64 ALALI, A.; KAGDI, H.; MALETIC, J. I. What's a typical commit? a characterization of open source software repositories. In: **Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on**. [S.l.: s.n.], 2008. p. 182–191. <https://doi.org/10.1109/ICPC.2008.24>.

65 RATTAN, D.; BHATIA, R.; SINGH, M. Software clone detection: A systematic review. **Information and Software Technology**, v. 55, n. 7, p. 1165–1199, 2013.

66 WETTEL, R.; MARINESCU, R. Archeology of code duplication: recovering duplication chains from small duplication fragments. In: **Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)**. [S.l.: s.n.], 2005. p. 8 pp.–. <https://doi.org/10.1109/SYNASC.2005.20>.

67 ROY, C.; CORDY, J. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In: **2008 16th IEEE International Conference on Program Comprehension**. [S.l.]: IEEE, 2008. p. 172–181. <https://doi.org/10.1109/ICPC.2008.41>.

68 KAMIYA, T.; KUSUMOTO, S.; INOUE, K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. **IEEE Transactions on Software Engineering**, v. 28, n. 7, p. 654–670, jul 2002. <https://doi.org/10.1109/TSE.2002.1019480>.

69 LI, Z. et al. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In: **Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6**. Berkeley, CA, USA: USENIX Association, 2004. (OSDI'04), p. 20.

70 JIANG, L. et al. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In: **29th International Conference on Software Engineering (ICSE'07)**. [S.l.]: IEEE, 2007. p. 96–105. <https://doi.org/10.1109/ICSE.2007.30>.

71 KRINKE, J. Identifying similar code with program dependence graphs. In: **Proceedings Eighth Working Conference on Reverse Engineering**. [S.l.]: IEEE Comput. Soc, 2001. p. 301–309. <https://doi.org/10.1109/WCRE.2001.957835>.

72 LORENZ, M.; KIDD, J. **Object-oriented Software Metrics: A Practical Guide**. [S.l.]: PTR Prentice Hall, 1994. (Prentice Hall object-oriented series).

73 FONTANA, F. A.; BRAIONE, P.; ZANONI, M. Automatic detection of bad smells in code: An experimental assessment. **Journal of Object Technology**, v. 11, n. 2, p. 1–5, 2012.

74 PELDSZUS, S. et al. Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching. In: **2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2016. p. 578–589. <https://doi.org/10.1145/2970276.2970338>.

75 TSANTALIS, N.; CHATZIGEORGIOU, A. Identification of Move Method Refactoring Opportunities. **IEEE Transactions on Software Engineering**, v. 35, n. 3, p. 347–367, may 2009. <https://doi.org/10.1109/TSE.2009.1>.

76 PALOMBA, F. **Code Smells: Relevance of the Problem and Novel Detection Techniques**. Tese (Doutorado) — University Of Salerno, Salerno, Italy, 2017.

77 MOHA, N. et al. From a domain analysis to the specification and detection of code and design smells. **Formal Aspects of Computing**, Springer-Verlag, v. 22, n. 3-4, p. 345–361, 2010. <https://doi.org/10.1007/s00165-009-0115-x>.

78 TUFANO, M. et al. When and why your code starts to smell bad. In: **Proceedings of the 37th International Conference on Software Engineering - Volume 1**. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 403–414. <https://doi.org/10.1109/ICSE.2015.59>.

79 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**. New York, NY, USA: ACM, 2008. (SoftVis '08), p. 77–86. <https://doi.org/10.1145/1409720.1409733>.

80 KESSENTINI, W. et al. A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection. **IEEE Transactions on Software Engineering**, v. 40, n. 9, p. 841–861, sep 2014. <https://doi.org/10.1109/TSE.2014.2331057>.

81 PALOMBA, F. et al. A textual-based technique for smell detection. In: **2016 IEEE 24th International Conference on Program Comprehension (ICPC)**. [S.l.: s.n.], 2016. p. 1–10. <https://doi.org/10.1109/ICPC.2016.7503704>.

82 CHATTERJI, D.; CARVER, J. C.; KRAFT, N. A. Code clones and developer behavior: results of two surveys of the clone research community. **Empirical Software Engineering**, v. 21, n. 4, p. 1476–1508, 2016. <https://doi.org/10.1007/s10664-015-9394-4>.

83 VINH, P. et al. **Context-Aware Systems and Applications: Second International Conference, ICCASA 2013, Phu Quoc Island, Vietnam, November 25-26, 2013, Revised Selected Papers**. [S.l.]: Springer International Publishing, 2014. (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering).

84 COOPER, K.; TORCZON, L. **Engineering a Compiler**. [S.l.]: Elsevier Science, 2011.

85 MUNRO, M. J. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In: **Software Metrics, 2005. 11th IEEE International Symposium**. [S.l.: s.n.], 2005. p. 15. <https://doi.org/10.1109/METRICS.2005.38>.

86 CHAMBERS, J. **Graphical methods for data analysis**. [S.l.]: Wadsworth International Group, 1983. (Chapman & Hall statistics series).

87 MCCABE, T. J. A complexity measure. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 2, n. 4, p. 308–320, jul. 1976. <https://doi.org/10.1109/TSE.1976.233837>.

88 BANDI, A.; WILLIAMS, B.; ALLEN, E. Empirical evidence of code decay: A systematic mapping study. In: **Reverse Engineering (WCRE), 2013 20th Working Conference on**. [S.l.: s.n.], 2013. p. 341–350. <https://doi.org/10.1109/WCRE.2013.6671309>.

89 PATE, J. R.; TAIRAS, R.; KRAFT, N. A. Clone evolution: a systematic review. **Journal of Software: Evolution and Process**, John Wiley & Sons, Ltd, v. 25, n. 3, p. 261–283, 2013. <https://doi.org/10.1002/smr.579>.

90 ZHANG, M.; HALL, T.; BADDOO, N. Code bad smells: a review of current knowledge. **Journal of Software Maintenance and Evolution: Research and Practice**, John Wiley & Sons, Ltd., v. 23, n. 3, p. 179–202, 2011. <https://doi.org/10.1002/smr.521>.

91 AIKO, Y.; LEON, M. Do developers care about code smells? An exploratory survey. In: **2013 20th Working Conference on Reverse Engineering (WCRE)**. [S.l.]: IEEE, 2013. p. 242–251. <https://doi.org/10.1109/WCRE.2013.6671299>.

92 KITCHENHAM, B. A.; BUDGEN, D.; BRERETON, O. P. Using mapping studies as the basis for further research - a participant-observer case study. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 53, n. 6, p. 638–651, jun. 2011. <https://doi.org/10.1016/j.infsof.2010.12.011>.

93 CHEN, T.-H.; THOMAS, S. W.; HASSAN, A. E. A survey on the use of topic models when mining software repositories. **Empirical Software Engineering**, p. 1–77, 2015. <https://doi.org/10.1007/s10664-015-9402-8>.

94 OPDYKE, W. F.; JOHNSON, R. E. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In: **Proceedings of SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications**. New York, New York, USA: ACM Press, 1990.

95 OPDYKE, W. F. **Refactoring Object-oriented Frameworks**. Tese (Doutorado) — University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

96 COHEN, J. A coefficient of agreement for nominal scales. **Educational and Psychological Measurement**, v. 20, n. 1, p. 37–46, 1960.

97 LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. **Biometrics**, [Wiley, International Biometric Society], v. 33, n. 1, p. 159–174, 1977.

98 KESSENTINI, M. et al. Design Defects Detection and Correction by Example. In: **2011 IEEE 19th International Conference on Program Comprehension**. [S.l.]: IEEE, 2011. p. 81–90. <https://doi.org/10.1109/ICPC.2011.22>.

99 ROMANO, D. et al. Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes. In: **2012 19th Working Conference on Reverse Engineering**. [S.l.]: IEEE, 2012. p. 437–446. <https://doi.org/10.1109/WCRE.2012.53>.

100 DUALA-EKOKO, E.; ROBILLARD, M. P. Clone Region Descriptors: Representing and Tracking Duplication in Source Code. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 20, n. 1, p. 3:1—-3:31, 2010. <https://doi.org/10.1145/1767751.1767754>.

101 ZHANG, G. et al. Towards contextual and on-demand code clone management by continuous monitoring. In: **2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.]: IEEE, 2013. p. 497–507. <https://doi.org/10.1109/ASE.2013.6693107>.

102 BASIT, H. A.; JARZABEK, S. Detecting Higher-level Similarity Patterns in Programs. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 30, n. 5, p. 156–165, 2005. <https://doi.org/10.1145/1095430.1081733>.

103 BURROWS, S.; TAHAGHOGHI, S. M. M.; ZOBEL, J. Efficient plagiarism detection for large code repositories. **Software: Practice and Experience**, John Wiley & Sons, Ltd., v. 37, n. 2, p. 151–175, 2007. <https://doi.org/10.1002/spe.750>.

104 KAWAMITSU, N. et al. Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity. In: **2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation**. [S.l.]: IEEE, 2014. p. 305–314. <https://doi.org/10.1109/SCAM.2014.17>.

105 AL-OMARI, F. et al. Detecting Clones Across Microsoft .NET Programming Languages. In: **2012 19th Working Conference on Reverse Engineering**. [S.l.]: IEEE, 2012. p. 405–414. <https://doi.org/10.1109/WCRE.2012.50>.

106 LI, H.; THOMPSON, S. Incremental Clone Detection and Elimination for Erlang Programs. In: GIANNAKOPOULOU, D.; OREJAS, F. (Ed.). **Fundamental Approaches to Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6603). p. 356–370. <https://doi.org/10.1007/978-3-642-19811-3{_}25>.

107 BROWN, C.; THOMPSON, S. Clone Detection and Elimination for Haskell. In: **Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation**. New York, NY, USA: ACM, 2010. (PEPM '10), p. 111–120. <https://doi.org/10.1145/1706356.1706378>.

108 WANG, W.; GODFREY, M. W. A Study of Cloning in the Linux SCSI Drivers. In: **2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation**. [S.l.]: IEEE, 2011. p. 95–104. <https://doi.org/10.1109/SCAM.2011.17>.

109 ANTONIOL, G. et al. Analyzing cloning evolution in the Linux kernel. **Information and Software Technology**, v. 44, n. 13, p. 755–765, 2002.

110 DEAN, T. R.; CHEN, J.; ALALFI, M. H. Clone Detection in Matlab Stateflow Models. **ECEASST**, v. 63, 2014.

111 CALEFATO, F.; LANUBILE, F.; MALLARDO, T. Function Clone Detection in Web Applications: A Semiautomated Approach. **J. Web Eng.**, Rinton Press, Incorporated, Paramus, NJ, v. 3, n. 1, p. 3–21, 2004.

112 KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. An empirical study of refactoring challenges and benefits at microsoft. **IEEE Transactions on Software Engineering**, IEEE – Institute of Electrical and Electronics Engineers, v. 40, n. 7, July 2014.

113 YAMASHITA, A. Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data. **Empirical Softw. Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 19, n. 4, p. 1111–1143, 2014. <https://doi.org/10.1007/s10664-013-9250-3>.

114 FARD, A. M.; MESBAH, A. JSNOSE: Detecting JavaScript Code Smells. In: **2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [S.l.]: IEEE, 2013. p. 116–125. <https://doi.org/10.1109/SCAM.2013.6648192>.

115 FOURATI, R.; BOUASSIDA, N.; ABDALLAH, H. A Metric-Based Approach for Anti-pattern Detection in UML Designs. In: LEE, R. (Ed.). **Computer and Information Science 2011**. [S.l.]: Springer Berlin Heidelberg, 2011, (Studies in Computational Intelligence, v. 364). p. 17–33. <https://doi.org/10.1007/978-3-642-21378-6_2>.

116 PIETRZAK, B.; WALTER, B. Leveraging Code Smell Detection with Inter-smell Relations. In: **Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering**. Berlin, Heidelberg: Springer-Verlag, 2006. (XP'06), p. 75–84. <https://doi.org/10.1007/11774129{_}8>.

117 SJOBERG, D. I. et al. Quantifying the Effect of Code Smells on Maintenance Effort. **IEEE Transactions on Software Engineering**, v. 39, n. 8, p. 1144–1156, aug 2013. <https://doi.org/10.1109/TSE.2012.89>.

118 YAMASHITA, A. et al. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In: **2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2015. p. 121–130. <https://doi.org/10.1109/ICSM.2015.7332458>.

119 MONTEIRO, M. P.; FERNANDES, J. a. M. Towards a catalog of aspect-oriented refactorings. In: **Proceedings of the 4th International Conference on Aspect-oriented Software Development**. New York, NY, USA: ACM, 2005. (AOSD '05), p. 111–122. <https://doi.org/10.1145/1052898.1052908>.

120 KHOMH, F. et al. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. **Journal of Systems and Software**, v. 84, n. 4, p. 559–572, apr 2011. <https://doi.org/10.1016/j.jss.2010.11.921>.

121 LIGU, E. et al. Identification of refused bequest code smells. In: **Software Maintenance (ICSM), 2013 29th IEEE International Conference on**. [S.l.: s.n.], 2013. p. 392–395. <https://doi.org/10.1109/ICSM.2013.55>.

122 KARASNEH, B. et al. Studying the relation between anti-patterns in design models and in source code. In: **2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.: s.n.], 2016. v. 1, p. 36–45. <https://doi.org/10.1109/SANER.2016.104>.

123 ANICHE, M. et al. A validated set of smells in model-view-controller architectures. In: **2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2016. p. 233–243. <https://doi.org/10.1109/ICSME.2016.12>.

124 HECHT, G.; MOHA, N.; ROUVOY, R. An empirical study of the performance impacts of android code smells. In: **Proceedings of the International Conference on Mobile Software Engineering and Systems**. New York, NY, USA: ACM, 2016. (MOBILESoft '16), p. 59–69.

125 PUNT, L.; VISSCHER, S.; ZAYTSEV, V. The a?b*a pattern: Undoing style in css and refactoring opportunities it presents. In: **2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2016. p. 67–77. <https://doi.org/10.1109/ICSME.2016.73>.

126 SABOURY, A. et al. An empirical study of code smells in javascript projects. In: **2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2017. p. 294–305. <https://doi.org/10.1109/SANER.2017.7884630>.

127 SHARMA, T.; FRAGKOULIS, M.; SPINELLIS, D. Does your configuration code smell? In: **Proceedings of the 13th International Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2016. (MSR '16), p. 189–200. <https://doi.org/10.1145/2901739.2901761>.

128 FONTANA, F. A. et al. Towards a prioritization of code debt: A code smell intensity index. In: **2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)**. [S.l.: s.n.], 2015. p. 16–24. <https://doi.org/10.1109/MTD.2015.7332620>.

129 STEIDL, D.; EDER, S. Prioritizing Maintainability Defects Based on Refactoring Recommendations. In: **Proceedings of the 22Nd International Conference on Program Comprehension**. New York, NY, USA: ACM, 2014. (ICPC 2014), p. 168–176. <https://doi.org/10.1145/2597008.2597805>.

130 CHRISTENSEN, R. **Log-Linear Models and Logistic Regression**. [S.l.]: Springer New York, 2006. (Springer Texts in Statistics).

131 FIELD, A.; MILES, J.; FIELD, Z. **Discovering Statistics Using R**. [S.l.]: SAGE Publications, 2012.

132 OSBORNE, J. **Best Practices in Logistic Regression**. [S.l.]: SAGE Publications, 2014.

133 SCHUMACKER, R. **Learning Statistics Using R**. [S.l.]: SAGE Publications, 2014.

134 DUNN, O.; CLARK, V. **Basic Statistics: A Primer for the Biomedical Sciences**. [S.l.]: Wiley, 2009.

135 HAHS-VAUGHN, D. **Applied Multivariate Statistical Concepts**. [S.l.]: Taylor & Francis, 2016.

136 HOSMER, D.; LEMESHOW, S.; STURDIVANT, R. **Applied Logistic Regression**. [S.l.]: Wiley, 2013. (Wiley Series in Probability and Statistics).

137 MCCULLAGH, P.; NELDER, J. **Generalized Linear Models, Second Edition**. [S.l.]: Taylor & Francis, 1989. (Chapman & Hall/CRC Monographs on Statistics & Applied Probability).

138 BAVOTA, G. et al. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: **2012 28th IEEE International Conference on Software Maintenance (ICSM)**. [S.l.]: IEEE, 2012. p. 56–65. <https://doi.org/10.1109/ICSM.2012.6405253>.

139 ROMPAEY, B. V. et al. On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. **IEEE Transactions on Software Engineering**, v. 33, n. 12, p. 800–817, dec 2007. <https://doi.org/10.1109/TSE.2007.70745>.

140 PALOMBA, F. et al. Landfill: An open dataset of code smells with public evaluation. In: **2015 IEEE/ACM 12th Working Conference on Mining Software Repositories**. [S.l.: s.n.], 2015. p. 482–485. <https://doi.org/10.1109/MSR.2015.69>.

141 MARINESCU, R. Assessing technical debt by identifying design flaws in software systems. **IBM Journal of Research and Development**, v. 56, n. 5, p. 9:1–9:13, Sept 2012. <https://doi.org/10.1147/JRD.2012.2204512>.

142 SAHA, R. K.; ROY, C. K.; SCHNEIDER, K. A. An automatic framework for extracting and classifying near-miss clone genealogies. In: **2011 27th IEEE International Conference on Software Maintenance (ICSM)**. [S.l.]: IEEE, 2011. p. 293–302. <https://doi.org/10.1109/ICSM.2011.6080796>.

143 SAHA, R. K. et al. Evaluating Code Clone Genealogies at Release Level: An Empirical Study. In: **2010 10th IEEE Working Conference on Source Code Analysis and Manipulation**. [S.l.]: IEEE, 2010. p. 87–96. <https://doi.org/10.1109/SCAM.2010.32>.

144 XIE, S. et al. An empirical study on the fault-proneness of clone migration in clone genealogies. In: **Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on**. [S.l.: s.n.], 2014. p. 94–103. <https://doi.org/10.1109/CSMR-WCRE.2014.6747229>.

145 KIM, M. et al. An Empirical Study of Code Clone Genealogies. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 30, n. 5, p. 187–196, 2005. <https://doi.org/10.1145/1095430.1081737>.

146 TUFANO, M. et al. When and why your code starts to smell bad (and whether the smells go away). **IEEE Transactions on Software Engineering**, PP, n. to appear, p. 1–1, 2017. <https://doi.org/10.1109/TSE.2017.2653105>.

147 OLBRICH, S. et al. The Evolution and Impact of Code Smells: A Case Study of Two Open Source Systems. In: **Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement**. Washington, DC, USA: IEEE Computer Society, 2009. (ESEM '09), p. 390–400. <https://doi.org/10.1109/ESEM.2009.5314231>.

148 GODFREY, M. W.; GERMAN, D. M. The past, present, and future of software evolution. In: **2008 Frontiers of Software Maintenance**. [S.l.: s.n.], 2008. p. 129–138. <https://doi.org/10.1109/FOSM.2008.4659256>.

149 LEHMAN, M. M. et al. Metrics and laws of software evolution - the nineties view. In: **Proceedings of the 4th International Symposium on Software Metrics**. Washington, DC, USA: IEEE Computer Society, 1997. (METRICS '97), p. 20–. ISBN 0-8186-8093-8. <https://doi.org/10.1109/METRIC.1997.637156>.

150 DYER, R. et al. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: **35th International Conference on Software Engineering**. [S.l.: s.n.], 2013. (ICSE'13), p. 422–431. <https://doi.org/10.1109/ICSE.2013.6606588>.

151 TSANTALIS, N. et al. Accurate and efficient refactoring detection in commit history. In: **Proceedings of the 40th International Conference on Software Engineering**. New York, NY, USA: ACM, 2018. (ICSE '18), p. 483–494. ISBN 978-1-4503-5638-1. <https://doi.org/10.1145/3180155.3180206>.

152 YAMASHITA, A.; MOONEN, L. Do code smells reflect important maintainability aspects? In: **2012 28th IEEE International Conference on Software Maintenance (ICSM)**. [S.l.]: IEEE, 2012. p. 306–315. <https://doi.org/10.1109/ICSM.2012.6405287>.

153 CRUZES, D. S.; DYBA, T. Recommended steps for thematic synthesis in software engineering. In: **2011 International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2011. p. 275–284. ISSN 1949-3789. <https://doi.org/10.1109/ESEM.2011.36>.

154 PALOMBA, F.; ZAIDMAN, A.; LUCIA, A. D. Automatic test smell detection using information retrieval techniques. In: **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2018. p. 311–322. ISSN 2576-3148. <https://doi.org/10.1109/ICSME.2018.00040>.

155 BAVOTA, G. et al. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. **IEEE Transactions on Software Engineering**, v. 40, n. 7, p. 671–694, jul 2014. <https://doi.org/10.1109/TSE.2013.60>.

156 BLEI, D. M.; NG, A. Y.; JORDAN, M. I. Latent dirichlet allocation. **J. Mach. Learn. Res.**, JMLR.org, v. 3, p. 993–1022, mar. 2003. ISSN 1532-4435.

157 THOMAS, S. W. et al. Studying software evolution using topic models. **Science of Computer Programming**, v. 80, p. 457 – 479, 2014. ISSN 0167-6423.

158 KAGDI, H.; COLLARD, M. L.; MALETIC, J. I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. **Journal of Software Maintenance and Evolution: Research and Practice**, v. 19, n. 2, p. 77–131, 2007. <https://doi.org/10.1002/smr.344>.

159 ROBLES, G. et al. In: . [S.l.: s.n.].

160 JIANG PEIJUN MA, X. S. D.; WANG, T. Distance metric based divergent change bad smell detection and refactoring scheme analysis. **International Journal of Innovative Computing, Information and Control**, v. 10, n. 4, p. 1519–1531, 2014.

161 BUCKLEY, J. et al. Towards a taxonomy of software change: Research articles. **J. Softw. Maint. Evol.**, John Wiley & Sons, Inc., New York, NY, USA, v. 17, n. 5, p. 309–332, set. 2005. <https://doi.org/10.1002/smr.v17:5>.

162 LINARES-VÁSQUEZ, M. et al. Domain Matters: Bringing Further Evidence of the Relationships Among Anti-patterns, Application Domains, and Quality-related Metrics in Java Mobile Apps. In: **Proceedings of the 22Nd International Conference on Program Comprehension**. New York, NY, USA: ACM, 2014. (ICPC 2014), p. 232–243. <https://doi.org/10.1145/2597008.2597144>.

163 MACIA, I. et al. Are Automatically-detected Code Anomalies Relevant to Architectural Modularity?: An Exploratory Analysis of Evolving Systems. In: **Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development**. New York, NY, USA: ACM, 2012. (AOSD '12), p. 167–178. <https://doi.org/10.1145/2162049.2162069>.

164 MACIA, I. et al. On the Impact of Aspect-Oriented Code Smells on Architecture Modularity: An Exploratory Study. In: **Software Components, Architectures and Reuse (SBCARS), 2011 Fifth Brazilian Symposium on**. [S.l.: s.n.], 2011. p. 41–50. <https://doi.org/10.1109/SBCARS.2011.18>.

165 AHMED, I. et al. An empirical study of design degradation: How software projects get worse over time. In: **2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.: s.n.], 2015. p. 1–10. <https://doi.org/10.1109/ESEM.2015.7321186>.

166 YAMASHITA, A.; COUNSELL, S. Code smells as system-level indicators of maintainability: An empirical study. **Journal of Systems and Software**, v. 86, n. 10, p. 2639–2653, oct 2013. <https://doi.org/10.1016/j.jss.2013.05.007>.

167 SOH, Z. et al. Do code smells impact the effort of different maintenance programming activities? In: **2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.: s.n.], 2016. v. 1, p. 393–402. <https://doi.org/10.1109/SANER.2016.103>.

168 Macia Bertran, I.; GARCIA, A.; STAA, A. von. An Exploratory Study of Code Smells in Evolving Aspect-oriented Systems. In: **Proceedings of the Tenth International Conference on Aspect-oriented Software Development**. New York, NY, USA: ACM, 2011. (AOSD '11), p. 203–214. <https://doi.org/10.1145/1960275.1960300>.

169 JAAFAR, F. et al. **Analysing Anti-patterns Static Relationships with Design Patterns**. 2013. Electronic Communications of the EASST (ECEASST).

170 ZAZWORKA, N. et al. Comparing four approaches for technical debt identification. **Software Quality Journal**, v. 22, n. 3, p. 403–426, 2014. <https://doi.org/10.1007/s11219-013-9200-8>.

171 VIDAL, S. A.; MARCOS, C.; DÍAZ-PACE, J. A. An approach to prioritize code smells for refactoring. **Journal Automated Software Engineering**, v. 23, n. 3, p. 501–532, 2016. <https://doi.org/10.1007/s10515-014-0175-x>.

172 MURPHY-HILL, E.; BLACK, A. P. An interactive ambient visualization for code smells. In: **Proceedings of the 5th International Symposium on Software Visualization**. New York, NY, USA: ACM, 2010. (SOFTVIS '10), p. 5–14. <https://doi.org/10.1145/1879211.1879216>.

173 LIU, H.; GUO, X.; SHAO, W. Monitor-Based Instant Software Refactoring. **IEEE Transactions on Software Engineering**, v. 39, n. 8, p. 1112–1126, aug 2013. <https://doi.org/10.1109/TSE.2013.4>.

174 MORALES, R.; MCINTOSH, S.; KHOMH, F. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In: **2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.: s.n.], 2015. p. 171–180. <https://doi.org/10.1109/SANER.2015.7081827>.

175 MACIA, I. et al. Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies. In: **2013 17th European Conference on Software Maintenance and Reengineering**. [S.l.]: IEEE, 2013. p. 177–186. <https://doi.org/10.1109/CSMR.2013.27>.

176 OIZUMI, W. et al. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: **Proceedings of the 38th International Conference on Software Engineering**. New York, NY, USA: ACM, 2016. (ICSE '16), p. 440–451. <https://doi.org/10.1145/2884781.2884868>.

177 SAHIN, D. et al. Code-smell detection as a bilevel problem. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 24, n. 1, p. 6:1–6:44, out. 2014. <https://doi.org/10.1145/2675067>.

178 OIZUMI, W. N. et al. On the relationship of code-anomaly agglomerations and architectural problems. **Journal of Software Engineering Research and Development**, v. 3, n. 1, p. 11, 2015. <https://doi.org/10.1186/s40411-015-0025-y>.

179 TUFANO, M. et al. An empirical investigation into the nature of test smells. In: **Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: ACM, 2016. (ASE 2016), p. 4–15. <https://doi.org/10.1145/2970276.2970340>.

180 PALOMBA, F. et al. Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. In: **2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2016. p. 244–255. <https://doi.org/10.1109/ICSME.2016.27>.

181 HALL, T. et al. Some Code Smells Have a Significant but Small Effect on Faults. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 23, n. 4, p. 33:1—-33:39, 2014. <https://doi.org/10.1145/2629648>.

182 PALOMBA, F. et al. Mining version histories for detecting code smells. **IEEE Transactions on Software Engineering**, v. 41, n. 5, p. 462–489, May 2015. <https://doi.org/10.1109/TSE.2014.2372760>.

183 SALEHIE, M.; LI, S.; TAHVILDARI, L. A metric-based heuristic framework to detect object-oriented design flaws. In: **14th IEEE International Conference on Program Comprehension (ICPC'06)**. [S.l.: s.n.], 2006. p. 159–168. <https://doi.org/10.1109/ICPC.2006.6>.

184 D'AMBROS, M.; BACCHELLI, A.; LANZA, M. On the impact of design flaws on software defects. In: **2010 10th International Conference on Quality Software**. [S.l.: s.n.], 2010. p. 23–31. <https://doi.org/10.1109/QSIC.2010.58>.

185 FONTANA, F. A. et al. Automatic metric thresholds derivation for code smell detection. In: **Proceedings of the Sixth International Workshop on Emerging Trends in Software Metrics**. Piscataway, NJ, USA: IEEE Press, 2015. (WETSoM '15), p. 44–53. <https://doi.org/10.1109/WETSoM.2015.14>.

186 MORALES, R. et al. Finding the best compromise between design quality and testing effort during refactoring. In: **2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.: s.n.], 2016. v. 1, p. 24–35. <https://doi.org/10.1109/SANER.2016.23>.

187 MAIGA, A. et al. SMURF: A SVM-based Incremental Anti-pattern Detection Approach. In: **2012 19th Working Conference on Reverse Engineering**. [S.l.]: IEEE, 2012. p. 466–475. <https://doi.org/10.1109/WCRE.2012.56>.

188 SAE-LIM, N.; HAYASHI, S.; SAEKI, M. Context-based code smells prioritization for prefactoring. In: **2016 IEEE 24th International Conference on Program Comprehension (ICPC)**. [S.l.: s.n.], 2016. p. 1–10. <https://doi.org/10.1109/ICPC.2016.7503705>.

189 LIU, H. et al. Dynamic and automatic feedback-based threshold adaptation for code smell detection. **IEEE Transactions on Software Engineering**, v. 42, n. 6, p. 544–558, June 2016. <https://doi.org/10.1109/TSE.2015.2503740>.

190 TSANTALIS, N.; CHATZIGEORGIOU, A. Ranking Refactoring Suggestions Based on Historical Volatility. In: **2011 15th European Conference on Software Maintenance and Reengineering**. [S.l.]: IEEE, 2011. p. 25–34. <https://doi.org/10.1109/CSMR.2011.7>.

191 FU, S.; SHEN, B. Code bad smell detection through evolutionary data mining. In: **2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.: s.n.], 2015. p. 1–9. <https://doi.org/10.1109/ESEM.2015.7321194>.

192 BOUSSAA, M. et al. Competitive coevolutionary code-smells detection. In: **Proceedings of the 5th International Symposium on Search Based Software Engineering - Volume 8084**. New York, NY, USA: Springer-Verlag New York, Inc., 2013. (SSBSE 2013), p. 50–65. <https://doi.org/10.1007/978-3-642-39742-4_6>.

193 OUNI, A. et al. Maintainability defects detection and correction: a multi-objective approach. **Journal Automated Software Engineering**, v. 20, n. 1, p. 47–79, 2013. <https://doi.org/10.1007/s10515-011-0098-8>.

194 EMDEN, E. van; MOONEN, L. Java quality assurance by detecting code smells. In: **Ninth Working Conference on Reverse Engineering, 2002. Proceedings.** [S.l.]: IEEE Comput. Soc, 2002. p. 97–106. <https://doi.org/10.1109/WCRE.2002.1173068>.

195 HERMANS, F.; AIVALOGLOU, E. Do code smells hamper novice programming? a controlled experiment on scratch programs. In: **2016 IEEE 24th International Conference on Program Comprehension (ICPC)**. [S.l.: s.n.], 2016. p. 1–10. <https://doi.org/10.1109/ICPC.2016.7503706>.

196 RAPU, D. et al. Using history information to improve design flaws detection. In: **Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.** [S.l.: s.n.], 2004. p. 223–232. <https://doi.org/10.1109/CSMR.2004.1281423>.

197 MIHANCEA, P. F.; MARINESCU, R. Towards the optimization of automatic detection of design flaws in object-oriented software systems. In: **Ninth European Conference on Software Maintenance and Reengineering**. [S.l.: s.n.], 2005. p. 92–101. <https://doi.org/10.1109/CSMR.2005.53>.

198 TOURWE, T.; MENS, T. Identifying refactoring opportunities using logic meta programming. In: **Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.** [S.l.: s.n.], 2003. p. 91–100. <https://doi.org/10.1109/CSMR.2003.1192416>.

199 KREIMER, J. Adaptive detection of design flaws. **Electronic Notes in Theoretical Computer Science**, v. 141, n. 4, p. 117 – 136, 2005.

200 JIAU, H. C.; CHEN, J. C. OBEY: Optimal Batched Refactoring Plan Execution for Class Responsibility Redistribution. **IEEE Transactions on Software Engineering**, v. 39, n. 9, p. 1245–1263, sep 2013. <https://doi.org/10.1109/TSE.2013.19>.

201 De Lucia, A.; OLIVETO, R.; VORRARO, L. Using structural and semantic metrics to improve class cohesion. In: **2008 IEEE International Conference on Software**

Maintenance. [S.l.]: IEEE, 2008. p. 27–36. <https://doi.org/10.1109/ICSM.2008.4658051>.

202 FOKAEFS, M. et al. Identification and application of Extract Class refactorings in object-oriented systems. **Journal of Systems and Software**, v. 85, n. 10, p. 2241–2260, oct 2012. <https://doi.org/10.1016/j.jss.2012.04.013>.

203 SALES, V. et al. Recommending Move Method refactorings using dependency sets. In: **2013 20th Working Conference on Reverse Engineering (WCRE)**. [S.l.]: IEEE, 2013. p. 232–241. <https://doi.org/10.1109/WCRE.2013.6671298>.

204 ABEBE, S. L. et al. Can Lexicon Bad Smells Improve Fault Prediction? In: **2012 19th Working Conference on Reverse Engineering**. [S.l.]: IEEE, 2012. p. 235–244. <https://doi.org/10.1109/WCRE.2012.33>.

205 BAVOTA, G.; De Lucia, A.; OLIVETO, R. Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. **Journal of Systems and Software**, v. 84, n. 3, p. 397–414, mar 2011. <https://doi.org/10.1016/j.jss.2010.11.918>.

206 ARNAOUDOVA, V. et al. A New Family of Software Anti-patterns: Linguistic Anti-patterns. In: **2013 17th European Conference on Software Maintenance and Reengineering**. [S.l.]: IEEE, 2013. p. 187–196. <https://doi.org/10.1109/CSMR.2013.28>.

207 ABEBE, S. L. et al. The Effect of Lexicon Bad Smells on Concept Location in Source Code. In: **2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation**. [S.l.]: IEEE, 2011. p. 125–134. <https://doi.org/10.1109/SCAM.2011.18>.

208 SCHUMACHER, J. et al. Building Empirical Support for Automated Code Smell Detection. In: **Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement**. New York, NY, USA: ACM, 2010. (ESEM '10), p. 8:1—-8:10. <https://doi.org/10.1145/1852786.1852797>.

209 ARNAOUDOVA, V.; PENTA, M. D.; ANTONIOL, G. Linguistic antipatterns: what they are and how developers perceive them. **Empirical Software Engineering**, v. 21, n. 1, p. 104–158, 2016. <https://doi.org/10.1007/s10664-014-9350-8>.

210 BAVOTA, G. et al. Are test smells really harmful? an empirical study. **Empirical Software Engineering**, v. 20, n. 4, p. 1052–1094, 2015. <https://doi.org/10.1007/s10664-014-9313-0>.

211 KIRK, D.; ROPER, M.; WALKINSHAW, N. Using Attribute Slicing to Refactor Large Classes. In: BINKLEY, D. W.; HARMAN, M.; KRINKE, J. (Ed.). **Beyond Program Slicing**. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum fur Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. (Dagstuhl Seminar Proceedings, 05451).

212 Du Bois, B. et al. Does god class decomposition affect comprehensibility? In: **IASTED Conf. on Software Engineering**. [S.l.: s.n.], 2006. p. 346–355.

213 GREILER, M.; Van Deursen, A.; STOREY, M.-A. Automated Detection of Test Fixture Strategies and Smells. In: **Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on**. [S.l.: s.n.], 2013. p. 322–331. <https://doi.org/10.1109/ICST.2013.45>.

214 NEUKIRCHEN, H.; BISANZ, M. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In: **Proceedings of the 19th IFIP TC6/WG6.1 International Conference, and 7th International Conference on Testing of Software and Communicating Systems**. Berlin, Heidelberg: Springer-Verlag, 2007. (TestCom'07/FATES'07), p. 228–243.

215 GREILER, M. et al. Strategies for avoiding text fixture smells during software evolution. In: **2013 10th Working Conference on Mining Software Repositories (MSR)**. [S.l.]: IEEE, 2013. p. 387–396. <https://doi.org/10.1109/MSR.2013.6624053>.

216 CARETTE, A. et al. Investigating the energy impact of android smells. In: **2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2017. p. 115–126. <https://doi.org/10.1109/SANER.2017.7884614>.

217 FENSKE, W. et al. When code smells twice as much: Metric-based detection of variability-aware code smells. In: **2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [S.l.: s.n.], 2015. p. 171–180. <https://doi.org/10.1109/SCAM.2015.7335413>.

218 STEIDL, D.; DEISSENBOECK, F. How do java methods grow? In: **2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [S.l.: s.n.], 2015. p. 151–160. <https://doi.org/10.1109/SCAM.2015.7335411>.

219 RASOOL, G.; ARSHAD, Z. A review of code smell mining techniques. **Journal of Software: Evolution and Process**, v. 27, n. 11, p. 867–895, 2015. <https://doi.org/10.1002/smr.1737>.

Apêndices

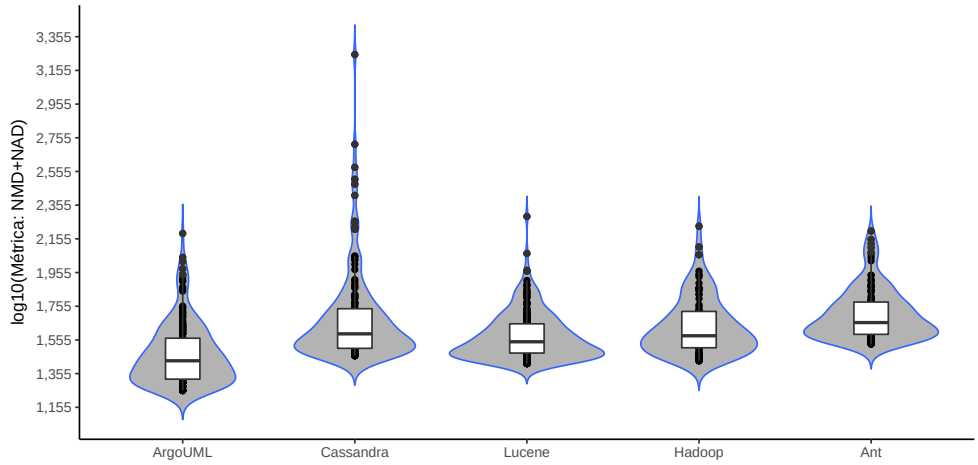


Figura 35 – Boxplot da métrica $NMD+NAD$ nas classes com o *smell* LC. Considera classes com apenas o *smell* LC e classes com a co-ocorrência de *smells* (LC/CC).

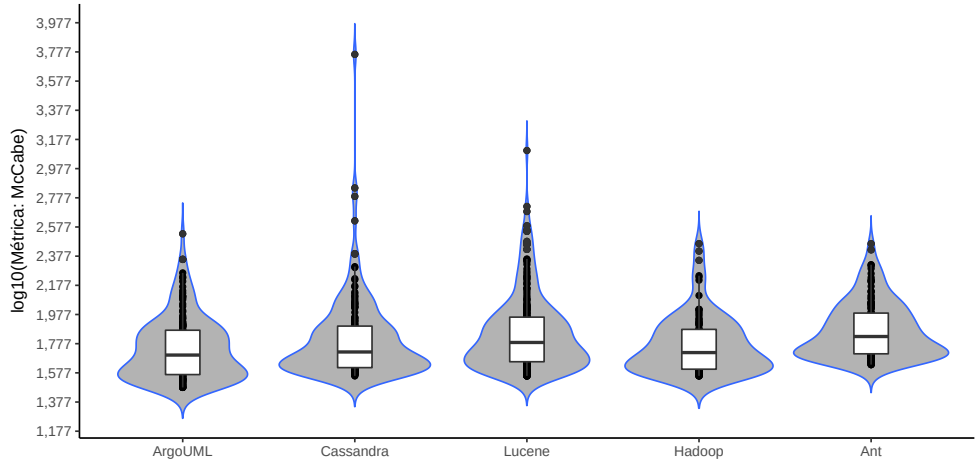


Figura 36 – Boxplot da métrica *McCabe* nas classes com o *smell* CC. Considera classes com apenas o *smell* CC e classes com a co-ocorrência de *smells* (LC/CC).

Tabela 26 – *Bad smells* para cada artigo no OBSG.

Paper ID	Large Class (Blob class, God Class)	Feature Envy	Long Method (God Method)	Data Class	Shotgun Surgery	Refused Bequest	Long Parameter List	Spaghetti Code	Duplicated Code	Message Chains Class	Few Methods (Lazy Class, Small Class)	Abstract Class (Speculative Generality)	Function Class (Func. Decomposition)	Data Clumps	Complex Class Only	Swiss Army Knife	Field Public (CDSBP)	Divergent Change	Misplaced Class	Temporary Variable, Several Purposes	Brain Method	Intensive Coupling	Dispersed (Extensive) Coupling	Interface Segregation Principle Violation	Switch Statements	Unit Test Smells
(47)°	X		X	X			X			X	X	X	X		X		X									
(45)°	X	X	X	X	X	X	X		X	X	X	X		X				X		X					X	
(152)°	X	X	X		X	X	X		X	X	X	X		X				X	X	X				X	X	
(162)°	X		X			X	X	X		X	X	X	X		X	X	X									
(163)•	X	X	X	X	X		X				X							X	X							
(52)°	X	X	X	X	X	X	X		X			X									X	X	X			
(164)•																										
(14)°	X		X			X	X	X		X	X	X			X	X	X									
(19)°	X	X	X		X		X											X	X							
(33)•	X		X			X	X	X		X	X	X			X	X	X									
(13)•	X	X	X	X	X													X								
(165)°	X	X		X	X	X			X	X				X							X	X				
(166)°	X	X	X	X	X	X								X					X	X				X		
(117)°	X	X	X	X	X	X								X					X	X				X		
(113)•	X	X	X	X	X	X								X					X	X				X		
(99)°	X		X			X	X	X		X	X	X			X	X	X									
(48)°	X	X	X			X	X	X			X	X			X		X									
(35)•	X	X	X	X	X	X								X					X	X				X		
(17)•	X	X	X	X	X	X								X					X	X				X		
(118)•	X	X		X	X	X			X	X				X								X				
(167)°	X	X	X	X	X	X								X					X	X				X		
(49)°	X		X			X	X	X		X		X			X	X	X									

Paper ID	AntiSingleton	Tradition Breaker	Large Class Only	Duplicated Code in Conditional Branches	Use Interface Instead of Implementation	Schizophrenic Class	Brain Class	Middle Man	Parallel Inheritance Hierarchies	Inappropriate Intimacy	Ambiguous Interface	Primitive Obsession	Lexicon Bad Smells	Type Check (State Check)	Duplicate Pointcut	God Pointcut	Redundant Pointcut	Connector Envy	Component Concern Overload	Scattered Parasitic Functionality	Smells in Android (Specific)	Controller Class	Low Cohesion Only	Classes with Different Interfaces	Wide Subsystem Interface	Instanceof
(47)°			X																			X	X			
(45)°								X	X	X		X												X		
(152)°																								X	X	
(162)°	X	X																								
(163)•			X												X	X	X									
(52)°		X				X	X																			
(164)•											X				X	X	X	X	X	X						
(14)°	X		X																							
(19)°										X	X							X	X	X						
(33)•	X		X																							
(13)•											X							X	X	X						
(165)°		X				X																				
(166)°				X	X																					
(117)°				X	X																					
(113)•				X	X																					
(99)°	X																									
(48)°								X		X																
(35)•				X	X																					
(17)•				X	X																					
(118)•		X				X																				
(167)°				X	X																					
(49)°	X																									

Paper ID	Typecasts	God Aspect	Composition Bloat	Forced Join Point	Lazy Aspect	Linguistic Antipatterns	Anonymous Pointcut	Lava Flow (Dead Code)	Extraneous Connector	Cyclic Dependency	Idle Pointcut	Child Class	Class Global Variable	Class One Method	Field Private	Has Children	Many Attributes	Method No Parameter	Multiple Interface	No Inheritance	No Polymorphism	Not Abstract	Not Complex	One Child Class	Parent Class Provides Protected	Rare Overriding
(47)°												X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
(45)°								X																		
(152)°																										
(162)°																										
(163)•		X	X	X																						
(52)°																										
(164)•		X	X	X	X		X				X															
(14)°																										
(19)°									X																	
(33)•																										
(13)•									X	X																
(165)°										X																
(166)°																										
(117)°																										
(113)•																										
(99)°																										
(48)°																										
(35)•																										
(17)•																										
(118)•																										
(167)°																										
(49)°																										

Paper ID	Two Inheritance	Incomplete Library Class	Comments	Simulation of Multiple Inheritance	Useless Field	Useless Method	Useless Class	Empty Catch Blocks	BaseClassKnowsDerivedClass	BaseClassShouldBeAbstract	ManyFieldAttributesButNotComplex	Junk Material	Borrowed Pointcut	Various Concerns	Abstract Method Introduction	God Package	Extraneous Adjacent Connector	Unused Interface	Promiscuous Package	Distorted Hierarchy	Smells in MVC Arq. (Specific)	Smells in CSS (Specific - ¹DSL)	Smells in JavaScript (Specific - ¹DSL)	Obsolete Parameter	Annotation Bundle	Smells in Puppet (Specific - ¹DSL)	#
(47)°	X																										29
(45)°		X	X																								23
(152)°				X																							19
(162)°									X	X	X																17
(163)•																											16
(52)°																											15
(164)•																	X										14
(14)°																											13
(19)°																											13
(33)•																											13
(13)•																		X									13
(165)°																											13
(166)°																											12
(117)°																											12
(113)•																											12
(99)°																											12
(48)°																											12
(35)•																											12
(17)•																											12
(118)•																				X							12
(167)°																											12
(49)°																											11

continued on next page

continued from previous page

Paper ID	Large Class (Blob class, God Class)	Feature Envy	Long Method (God Method)	Data Class	Shotgun Surgery	Refused Bequest	Long Parameter List	Spaghetti Code	Duplicated Code	Message Chains Class	Few Methods (Lazy Class, Small Class)	Abstract Class (Speculative Generality)	Function Class (Func. Decomposition)	Data Clumps	Complex Class Only	Swiss Army Knife	Field Public (CDSBP)	Divergent Change	Misplaced Class	Temporary Variable, Several Purposes	Brain Method	Intensive Coupling	Dispersed (Extensive) Coupling	Interface Segregation Principle Violation	Switch Statements	Unit Test Smells	AntiSingleton	Tradition Breaker	Large Class Only	Duplicated Code in Conditional Branches	Use Interface Instead of Implementation	Schizophrenic Class	Brain Class	Middle Man	Parallel Inheritance Hierarchies
(168)•		X																																	
(169)°	X		X			X	X	X		X		X			X	X	X										X								
(61)°	X	X	X			X	X	X		X	X	X			X		X																		
(32)°	X	X			X	X		X	X				X			X		X																	X
(57)°	X		X			X	X			X		X			X	X	X										X								
(77)•	X		X			X	X	X			X	X	X			X													X						
(79)°	X	X	X	X		X	X			X				X																				X	
(170)°	X	X		X	X	X															X	X	X					X					X		
(171)°	X	X		X	X	X															X	X	X					X					X		
(2)°	X	X	X	X	X	X													X																
(30)•	X	X	X				X		X																										
(80)°	X	X		X	X		X	X			X		X																						
(141)°	X			X		X			X					X							X	X										X			
(172)°	X	X	X							X				X											X										
(173)°	X		X	X			X		X								X								X										
(174)°	X	X		X					X					X														X				X			
(175)°	X	X	X		X														X			X	X												
(176)•	X	X	X	X	X		X											X																	
(177)°	X	X		X			X	X			X		X																						
(18)°	X	X	X	X	X	X																													
(178)•	X	X	X	X	X													X																	
(73)°	X	X	X				X		X																				X						
(179)•	X							X					X		X		X									X									

Paper ID	Inappropriate Intimacy	Ambiguous Interface	Primitive Obsession	Lexicon Bad Smells	Type Check (State Check)	Duplicate Pointcut	God Pointcut	Redundant Pointcut	Connector Envy	Component Concern Overload	Scattered Parasitic Functionality	Smells in Android (Specific)	Controller Class	Low Cohesion Only	Classes with Different Interfaces	Wide Subsystem Interface	Instanceof	Typecasts	God Aspect	Composition Bloat	Forced Join Point	Lazy Aspect	Linguistic Antipatterns	Anonymous Pointcut	Lava Flow (Dead Code)	Extraneous Connector	Cyclic Dependency	Idle Pointcut	Child Class	Class Global Variable	Class One Method	Field Private	Has Children	Many Attributes	Method No Parameter
(168)•						X	X	X														X		X				X							
(169)°																																			
(61)°																																			
(32)°					X																														
(57)°																																			
(77)•																																			
(79)°			X																																
(170)°																																			
(171)°																																			
(2)°																X																			
(30)•			X																																
(80)°																																			
(141)°																																			
(172)°																	X	X																	
(173)°																																			
(174)°																																			
(175)°																																			
(176)•																																			
(177)°																																			
(18)°																																			
(178)•																																			
(73)°																																			
(179)•																																			

Paper ID	Multiple Interface	No Inheritance	No Polymorphism	Not Abstract	Not Complex	One Child Class	Parent Class Provides Protected	Rare Overriding	Two Inheritance	Incomplete Library Class	Comments	Simulation of Multiple Inheritance	Useless Field	Useless Method	Useless Class	Empty Catch Blocks	BaseClassKnowsDerivedClass	BaseClassShouldBeAbstract	ManyFieldAttributesButNotComplex	Junk Material	Borrowed Pointcut	Various Concerns	Abstract Method Introduction	God Package	Extraneous Adjacent Connector	Unused Interface	Promiscuous Package	Distorted Hierarchy	Smells in MVC Arq. (Specific)	Smells in CSS (Specific - 1DSL)	Smells in JavaScript (Specific - 1DSL)	Obsolete Parameter	Annotation Bundle	Smells in Puppet (Specific - 1DSL)	#
(168)•																				X	X	X	X												11
(169)°																																			11
(61)°																																			11
(32)°																																			11
(57)°																																			10
(77)•																																			10
(79)°																																			10
(170)°																																			10
(171)°																																			10
(2)°																								X											9
(30)•													X	X	X																				9
(80)°																																			8
(141)°																																			8
(172)°																																			8
(173)°																																			7
(174)°																																			7
(175)°																																			7
(176)•																																			7
(177)°																																			7
(18)°																																			6
(178)•																																			6
(73)°																																			6
(179)•																																			6

continued on next page

continued from previous page

Paper ID	Large Class (Blob class, God Class)	Feature Envy	Long Method (God Method)	Data Class	Shotgun Surgery	Refused Bequest	Long Parameter List	Spaghetti Code	Duplicated Code	Message Chains Class	Few Methods (Lazy Class, Small Class)	Abstract Class (Speculative Generality)	Function Class (Func. Decomposition)	Data Clumps	Complex Class Only	Swiss Army Knife	Field Public (CDSBP)	Divergent Change	Misplaced Class	Temporary Variable, Several Purposes	Brain Method	Intensive Coupling	Dispersed (Extensive) Coupling	Interface Segregation Principle Violation	Switch Statements	Unit Test Smells
(180)°	X			X	X					X											X		X			
(128)°	X			X	X					X											X		X			
(38)°	X	X			X													X								
(114)°	X						X				X														X	
(115)°	X			X									X			X										
(181)•										X		X		X											X	
(74)°	X			X												X										
(81)°	X	X	X																X							
(78)°	X							X					X		X		X									
(182)°	X	X			X													X								
(183)°	X	X	X		X	X																				
(184)°		X			X																X	X	X			
(185)°	X			X	X																X		X			
(186)°	X						X	X			X	X														
(187)°	X							X					X			X										
(28)•	X							X					X			X										
(4)•	X							X					X			X										
(53)°	X	X		X																						
(58)°	X	X	X	X																						
(56)°	X		X												X											
(188)°	X			X		X																				
(189)°		X	X						X																	
(9)°		X	X						X																	

Paper ID	AntiSingleton	Tradition Breaker	Large Class Only	Duplicated Code in Conditional Branches	Use Interface Instead of Implementation	Schizophrenic Class	Brain Class	Middle Man	Parallel Inheritance Hierarchies	Inappropriate Intimacy	Ambiguous Interface	Primitive Obsession	Lexicon Bad Smells	Type Check (State Check)	Duplicate Pointcut	God Pointcut	Redundant Pointcut	Connector Envy	Component Concern Overload	Scattered Parasitic Functionality	Smells in Android (Specific)	Controller Class	Low Cohesion Only	Classes with Different Interfaces	Wide Subsystem Interface	Instanceof
(180)°																										
(128)°																										
(38)°									X																	
(114)°																										
(115)°																										
(181)•								X																		
(74)°																						X	X			
(81)°																										
(78)°																										
(182)°									X																	
(183)°																										
(184)°																										
(185)°																										
(186)°																										
(187)°																										
(28)•																										
(4)•																										
(53)°							X																			
(58)°																										
(56)°																					X					
(188)°						X																				
(189)°										X																
(9)°													X													

Paper ID	Typecasts	God Aspect	Composition Bloat	Forced Join Point	Lazy Aspect	Linguistic Antipatterns	Anonymous Pointcut	Lava Flow (Dead Code)	Extraneous Connector	Cyclic Dependency	Idle Pointcut	Child Class	Class Global Variable	Class One Method	Field Private	Has Children	Many Attributes	Method No Parameter	Multiple Interface	No Inheritance	No Polymorphism	Not Abstract	Not Complex	One Child Class	Parent Class Provides Protected	Rare Overriding
(180)°																										
(128)°																										
(38)°																										
(114)°																										
(115)°								X																		
(181)•																										
(74)°																										
(81)°																										
(78)°																										
(182)°																										
(183)°																										
(184)°																										
(185)°																										
(186)°																										
(187)°																										
(28)•																										
(4)•																										
(53)°																										
(58)°																										
(56)°																										
(188)°																										
(189)°																										
(9)°																										

Paper ID	Two Inheritance	Incomplete Library Class	Comments	Simulation of Multiple Inheritance	Useless Field	Useless Method	Useless Class	Empty Catch Blocks	BaseClassKnowsDerivedClass	BaseClassShouldBeAbstract	ManyFieldAttributesButNotComplex	Junk Material	Borrowed Pointcut	Various Concerns	Abstract Method Introduction	God Package	Extraneous Adjacent Connector	Unused Interface	Promiscuous Package	Distorted Hierarchy	Smells in MVC Arq. (Specific)	Smells in CSS (Specific - [1]DSL)	Smells in JavaScript (Specific - [1]DSL)	Obsolete Parameter	Annotation Bundle	Smells in Puppet (Specific - [1]DSL)	#
(180)°																											6
(128)°																											6
(38)°																											5
(114)°								X																			5
(115)°																											5
(181)•																											5
(74)°																											5
(81)°																			X								5
(78)°																											5
(182)°																											5
(183)°																											5
(184)°																											5
(185)°																											5
(186)°																											5
(187)°																											4
(28)•																											4
(4)•																											4
(53)°																											4
(58)°																											4
(56)°																											4
(188)°																											4
(189)°																											4
(9)°																											4

continued on next page

continued from previous page

Paper ID	Large Class (Blob class, God Class)	Feature Envy	Long Method (God Method)	Data Class	Shotgun Surgery	Refused Bequest	Long Parameter List	Spaghetti Code	Duplicated Code	Message Chains Class	Few Methods (Lazy Class, Small Class)	Abstract Class (Speculative Generality)	Function Class (Func. Decomposition)	Data Clumps	Complex Class Only	Swiss Army Knife	Field Public (CDSBP)	Divergent Change	Misplaced Class	Temporary Variable, Several Purposes	Brain Method	Intensive Coupling	Dispersed (Extensive) Coupling	Interface Segregation Principle Violation	Switch Statements	Unit Test Smells	AntiSingleton	Tradition Breaker	Large Class Only	Duplicated Code in Conditional Branches	Use Interface Instead of Implementation	Schizophrenic Class	Brain Class	Middle Man	Parallel Inheritance Hierarchies	Inappropriate Intimacy	Ambiguous Interface	Primitive Obsession	Lexicon Bad Smells	Type Check (State Check)	Duplicate Pointcut	God Pointcut	Redundant Pointcut	Connector Envy	Component Concern Overload	Scattered Parasitic Functionality	Smells in Android (Specific)	Controller Class	Low Cohesion Only	Classes with Different Interfaces	Wide Subsystem Interface	Instanceof	Typecasts	God Aspect	Composition Bloat	Forced Join Point	Lazy Aspect	Linguistic Antipatterns	Anonymous Pointcut	Lava Flow (Dead Code)	Extraneous Connector	Cyclic Dependency	Idle Pointcut	Child Class	Class Global Variable	Class One Method	Field Private	Has Children	Many Attributes	Method No Parameter	Multiple Interface	No Inheritance	No Polymorphism	Not Abstract	Not Complex	One Child Class	Parent Class Provides Protected	Rare Overriding	Two Inheritance	Incomplete Library Class	Comments	Simulation of Multiple Inheritance	Useless Field	Useless Method	Useless Class	Empty Catch Blocks	BaseClassKnowsDerivedClass	BaseClassShouldBeAbstract	ManyFieldAttributesButNotComplex	Junk Material	Borrowed Pointcut	Various Concerns	Abstract Method Introduction	God Package	Extraneous Adjacent Connector	Unused Interface	Promiscuous Package	Distorted Hierarchy	Smells in MVC Arq. (Specific)	Smells in CSS (Specific - [1]DSL)	Smells in JavaScript (Specific - [1]DSL)	Obsolete Parameter	Annotation Bundle	Smells in Puppet (Specific - [1]DSL)	#
(55)°	X	X	X																																					X																																																														4
(122)°	X		X								X				X																																																																																									4
(46)°	X						X		X																																																																																																	3
(91)°	X		X						X																																																																																																3
(120)°	X							X					X																																																																																											3
(98)°	X							X					X																																																																																										3
(190)°		X	X																																					X																																																																3
(116)•	X	X		X																																																																																																							3
(37)°		X	X																						X																																																																														3
(191)°					X				X									X																																																																																						3
(54)°	X	X																X																																																																																					3
(192)°	X							X					X																																																																																									3
(193)°	X							X					X																																																																																								3
(51)°	X																																X																																																																								2
(194)°																																																				X	X																																																								2
(147)°	X				X																																																																																																							2
(129)°			X						X																																																																																																2
(34)•	X							X																																																																																																		2
(21)°	X			X																																																																																																								2
(85)°											X									X																																																																																					2
(195)°			X						X																																																																																																2
(196)°	X			X																																																																																																								2
(197)°	X			X																																																																																																							2
(198)°																																					X																																																																		X			2
(199)°	X		X																																																																																																										2

continued on next page

continued from previous page

Paper ID	Large Class (Blob class, God Class)	Feature Envy	Long Method (God Method)	Data Class	Shotgun Surgery	Refused Bequest	Long Parameter List	Spaghetti Code	Duplicated Code	Message Chains Class	Few Methods (Lazy Class, Small Class)	Abstract Class (Speculative Generality)	Function Class (Func. Decomposition)	Data Clumps	Complex Class Only	Swiss Army Knife	Field Public (CDSBP)	Divergent Change	Misplaced Class	Temporary Variable, Several Purposes	Brain Method	Intensive Coupling	Dispersed (Extensive) Coupling	Interface Segregation Principle Violation	Switch Statements	Unit Test Smells
(23)	X																									
(200)																										
(201)	X																									
(202)	X																									
(155)		X																								
(203)		X																								
(75)		X																								
(138)																										X
(204)																										
(205)	X																									
(139)																										X
(206)																										
(207)																										
(208)	X																									
(209)																										
(210)																										X
(211)	X																									
(212)	X																									
(213)																										X
(214)																										X
(215)																										X
(5)	X																									
(123)																										

Paper ID	AntiSingleton	Tradition Breaker	Large Class Only	Duplicated Code in Conditional Branches	Use Interface Instead of Implementation	Schizophrenic Class	Brain Class	Middle Man	Parallel Inheritance Hierarchies	Inappropriate Intimacy	Ambiguous Interface	Primitive Obsession	Lexicon Bad Smells	Type Check (State Check)	Duplicate Pointcut	God Pointcut	Redundant Pointcut	Connector Envy	Component Concern Overload	Scattered Parasitic Functionality	Smells in Android (Specific)	Controller Class	Low Cohesion Only	Classes with Different Interfaces	Wide Subsystem Interface
(23)																									
(200)								X																	
(201)																									
(202)																									
(155)																									
(203)																									
(75)																									
(138)																									
(204)													X												
(205)																									
(139)																									
(206)																									
(207)													X												
(208)																									
(209)																									
(210)																									
(211)																									
(212)																									
(213)																									
(214)																									
(215)																									
(5)																									
(123)																									

Paper ID	Instanceof	Typecasts	God Aspect	Composition Bloat	Forced Join Point	Lazy Aspect	Linguistic Antipatterns	Anonymous Pointcut	Lava Flow (Dead Code)	Extraneous Connector	Cyclic Dependency	Idle Pointcut	Child Class	Class Global Variable	Class One Method	Field Private	Has Children	Many Attributes	Method No Parameter	Multiple Interface	No Inheritance	No Polymorphism	Not Abstract	Not Complex	One Child Class	Parent Class Provides Protected	Rare Overriding
(23)																											
(200)																											
(201)																											
(202)																											
(155)																											
(203)																											
(75)																											
(138)																											
(204)																											
(205)																											
(139)																											
(206)							X																				
(207)																											
(208)																											
(209)							X																				
(210)																											
(211)																											
(212)																											
(213)																											
(214)																											
(215)																											
(5)																											
(123)																											

Paper ID	Two Inheritance	Incomplete Library Class	Comments	Simulation of Multiple Inheritance	Useless Field	Useless Method	Useless Class	Empty Catch Blocks	BaseClassKnowsDerivedClass	BaseClassShouldBeAbstract	ManyFieldAttributesButNotComplex	Junk Material	Borrowed Pointcut	Various Concerns	Abstract Method Introduction	God Package	Extraneous Adjacent Connector	Unused Interface	Promiscuous Package	Distorted Hierarchy	Smells in MVC Arq. (Specific)	Smells in CSS (Specific - [1]DSL)	Smells in JavaScript (Specific - [1]DSL)	Obsolete Parameter	Annotation Bundle	Smells in Puppet (Specific - [1]DSL)	#
(23)																											1
(200)																											1
(201)																											1
(202)																											1
(155)																											1
(203)																											1
(75)																											1
(138)																											1
(204)																											1
(205)																											1
(139)																											1
(206)																											1
(207)																											1
(208)																											1
(209)																											1
(210)																											1
(211)																											1
(212)																											1
(213)																											1
(214)																											1
(215)																											1
(5)																											1
(123)																				X						1

continued on next page

continued from previous page

Paper ID	Large Class (Blob class, God Class)	Feature Envy	Long Method (God Method)	Data Class	Shotgun Surgery	Refused Bequest	Long Parameter List	Spaghetti Code	Duplicated Code	Message Chains Class	Few Methods (Lazy Class, Small Class)	Abstract Class (Speculative Generality)	Function Class (Func. Decomposition)	Data Clumps	Complex Class Only	Swiss Army Knife	Field Public (CDSBP)	Divergent Change	Misplaced Class	Temporary Variable, Several Purposes	Brain Method	Intensive Coupling	Dispersed (Extensive) Coupling	Interface Segregation Principle Violation	Switch Statements	Unit Test Smells
(125)																										
(216)																										
(126)																										
(217)																										
(218)			X																							
(127)																										
(124)																										
(219)																										
	87	49	48	37	32	30	26	23	18	18	17	16	16	15	14	14	13	12	12	9	9	8	8	7	7	7

Paper ID	AntiSingleton	Tradition Breaker	Large Class Only	Duplicated Code in Conditional Branches	Use Interface Instead of Implementation	Schizophrenic Class	Brain Class	Middle Man	Parallel Inheritance Hierarchies	Inappropriate Intimacy	Ambiguous Interface	Primitive Obsession	Lexicon Bad Smells	Type Check (State Check)	Duplicate Pointcut	God Pointcut	Redundant Pointcut	Connector Envy	Component Concern Overload	Scattered Parasitic Functionality	Smells in Android (Specific)	Controller Class	Low Cohesion Only	Classes with Different Interfaces	Wide Subsystem Interface	Instanceof
(125)																										
(216)																					X					
(126)																										
(217)																										
(218)																										
(127)																										
(124)																					X					
(219)																										
	7	7	6	6	6	6	5	5	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2

Paper ID	Typecasts	God Aspect	Composition Bloat	Forced Join Point	Lazy Aspect	Linguistic Antipatterns	Anonymous Pointcut	Lava Flow (Dead Code)	Extraneous Connector	Cyclic Dependency	Idle Pointcut	Child Class	Class Global Variable	Class One Method	Field Private	Has Children	Many Attributes	Method No Parameter	Multiple Interface	No Inheritance	No Polymorphism	Not Abstract	Not Complex	One Child Class	Parent Class Provides Protected	Rare Overriding
(125)																										
(216)																										
(126)																										
(217)																										
(218)																										
(127)																										
(124)																										
(219)																										
	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Paper ID	Two Inheritance	Incomplete Library Class	Comments	Simulation of Multiple Inheritance	Useless Field	Useless Method	Useless Class	Empty Catch Blocks	BaseClassKnowsDerivedClass	BaseClassShouldBeAbstract	ManyFieldAttributesButNotComplex	Junk Material	Borrowed Pointcut	Various Concerns	Abstract Method Introduction	God Package	Extraneous Adjacent Connector	Unused Interface	Promiscuous Package	Distorted Hierarchy	Smells in MVC Arq. (Specific)	Smells in CSS (Specific - [1]DSL)	Smells in JavaScript (Specific - [1]DSL)	Obsolete Parameter	Annotation Bundle	Smells in Puppet (Specific - [1]DSL)	#
(125)																						X					1
(216)																											1
(126)																							X				1
(217)																									X		1
(218)																											1
(127)																										X	1
(124)																											1
(219)																											
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

[1]Domain Specific Language - DSL.

°Co-ocorrência de *smells*; •Co-estudo de *smells* (é um subconjunto de co-ocorrências).