IDENTIFICAÇÃO E GERAÇÃO DE ORÁCULOS DE *BAD SMELLS* EM SOFTWARE

RAFAEL PRATES FERREIRA TRINDADE

IDENTIFICAÇÃO E GERAÇÃO DE ORÁCULOS DE *BAD SMELLS* EM SOFTWARE

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha
Coorientadora: Kecia Aline Marques Ferreira

Belo Horizonte
Agosto de 2020

Prates Ferreira Trindade, Rafael

D1234p Identificação e Geração de Oráculos de *Bad Smells* em Software / Rafael Prates Ferreira Trindade. — Belo Horizonte, 2020
xix, 83 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientadora: Mariza Andrade da Silva Bigonha

1. *Bad smell*. 2. *Code Smell*. 3. Oráculo. 4. *Benchmark*. I. Orientadora. II. Coorientadora. III. Título.

CDU 519.6*82.10

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Identificação e Geração de Oráculos de Bad Smells em Software

RAFAEL PRATES FERREIRA TRINDADE

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Orientadora
Departamento de Ciência da Computação - UFMG

PROFA. KÉCIA ALINE MARQUES FERREIRA - Coorientadora
Departamento de Computação - CEFET-MG

PROF. MARCELO DE ALMEIDA MAIA
Faculdade de Computação - UFU

PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 21 de Agosto de 2020.

Agradecimentos

Às professoras Mariza Andrade da Silva Bigonha e Kecia Aline Marques Ferreira por todos os ensinamentos e incentivos que foram indispensáveis para a conclusão do presente trabalho. Pelos momentos de descontração durante nossas reuniões e as conversas agradáveis que tivemos.

Aos professores da Universidade Federal de Minas Gerais pelo conhecimento transmitido e incentivo à pesquisa.

À Universidade Federal de Minas Gerais pelo apoio concedido e pelo ambiente acolhedor.

Agradeço Cristiano Neiva Abrantes, aluno da Iniciação Científica do CEFET-MG, que atuou na implementação da ferramenta `InspectBSmell`, e ao CNPq que proveu sua bolsa de pesquisa.

Ao Eterno Deus que me concedeu saúde e capacidade para concluir o trabalho de Mestrado.

Aos meus pais pelas muitas vezes que me pouparam de distrações para que me dedicasse ao trabalho e por compreenderem minhas ausências.

Aos meus irmãos e aos demais familiares por todo amor e carinho que recebo sempre.

Agradeço Acacio Fernando de Oliveira, Guilherme Saulo Alves e Roberta Coeli Neves Moreira por todas palavras de apoio e por terem disponibilizado o tempo tão escasso de cada um para me auxiliarem na execução do presente trabalho.

Resumo

Um *bad smell* é um sintoma de que em um determinado trecho de código pode existir um problema mais profundo de projeto que pode comprometer a manutenibilidade do software. Muitos dos estudos sobre *bad smell* se baseiam em técnicas e ferramentas de detecção de *bad smells*. A validade desses estudos são, então, dependentes da eficácia dessas abordagens. A eficácia dessas propostas, em geral, é avaliada por meio de comparação com oráculos, que são conjuntos de dados de referência sobre os *bad smells* existentes em um software. Os oráculos são, portanto, essenciais para a realização de estudos sobre o assunto. Oráculos produzidos via inspeção manual de software são difíceis de serem encontrados e produzidos. Com isso, muitas vezes, os oráculos utilizados são gerados a partir de uma abordagem automática de detecção de *bad smell*. Porém, a própria avaliação dessas abordagens automáticas depende de comparação com dados de oráculos. Isso, consequentemente, pode afetar a credibilidade dessas abordagens automáticas e ameaça a validade dos estudos que as utilizam. Este trabalho visa contribuir para a solução deste problema identificando oráculos disponíveis na literatura, bem como construindo um oráculo de *bad smells*. Para isso, (1) foi realizada uma revisão sistemática da literatura que identificou quatro oráculos disponíveis e produzidos via abordagem manual; (2) foi construída uma ferramenta para auxiliar na identificação de *bad smells* via inspeção manual; (3) foi construído um oráculo de abordagem manual para cinco *bad smells*: *Large Class*, *Long Method*, *Data Class*, *Feature Envy* e *Refused Bequest*; e (4) utilizando o oráculo criado, foi realizado um estudo comparativo de três ferramentas de detecção de *bad smells*: `JDeodorant`, `JSpIRIT` e `FindSmells`. O oráculo criado contém dados de cinco sistemas de software Java de código aberto. Os resultados deste trabalho contribuem com um oráculo sólido de livre consulta para a construção e avaliação de abordagens, ferramentas e sistemas de recomendação que envolvam a detecção de *bad smells*.

Palavras-chave: *Bad Smell*, *Code Smell*, *Design Anomaly*, *Benchmark*, Oráculo, Revisão Sistemática da Literatura.

Abstract

A bad smell is a symptom of a deeper design problem in a piece of software that may harm the system's maintainability. Many of the studies on bad smells rely on techniques and tools for detecting bad smells. Hence, the validity of such studies are affected by the efficacy of these techniques and tools. In general, the efficacy of such proposals is assessed by comparing their results with oracles, i.e., a set of reference data regarding the bad smells existing in a software system. Therefore, oracles are essential for studies on this subject. Oracles produced via manual inspection of code are difficult to find and produce. For this reason, usually, the oracles are generated by an automatic approach for detecting bad smell. On the other hand, the assessment of those approaches themselves are done by comparing their results with oracles. This, consequently, may affect the automatic approaches' credibility and threat the validity of studies that rely on them. This work aims to contribute to solve this problem by identifying oracles available in the literature, as well as creating an oracle of bad smells. For this purpose, we (1) carried out a systematic literature review that identified four oracles produced by a manual approach and available for download; (2) developed a tool to assist manual inspection of software to identify bad smells; (3) created an oracle by means of a manual approach regarding five bad smells: Large Class, Long Method, Data Class, Feature Envy, and Refused Bequest; (4) applied the generated oracle to carry out a comparative study of three tools for bad smell detection: `JDeodorant`, `JSpIRIT`, and `FindSmells`. The oracle created in this work has data form five Java based open source systems. The results of this study contribute with a reliable oracle of bad smells available for free consultation to the construction and evaluation of approaches, tools, and recommending systems involving bad smells.

Keywords: Bad Smell, Code Smell, Design Anomaly, Benchmark, Oracle, Systematic Literature Review.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Os esforços durante o desenvolvimento de software não são efêmeros e definitivos. Uma versão de um programa nasce com data de validade que pode ser determinada por uma série de fatores. Essa dinamicidade é postulada por Lehman na primeira das suas leis acerca da evolução de software conforme esclarece Sommerville [2011]. Os motivos da interferência em sistemas são "para simples ajustes pós-implantação, ou por melhorias substanciais, por força da legislação e, finalmente, por estar gerando erros" [Rezende, 2005]. Além dos problemas impostos pela necessidade de dar continuidade ao desenvolvimento, existe uma razão causal entre a manutenção e o declínio na qualidade do programa. Tal relação é definida por Lehman et al. [1997] conforme a segunda lei deles que trata da complexidade crescente, afirmando que complexidade de um sistema aumenta ao longo de sua evolução, a menos que seja feito algum trabalho para mantê-lo ou reduzi-lo [Sommerville, 2011].

Um dos indicadores de problemas na qualidade de software é o que se chama de *bad smells* ou *code smells*. De acordo com Fowler, o termo *code smell* foi cunhado por Beck ao longo da escrita do livro deles sobre *refatoração* de código (*refactoring*) [Fowler et al., 1999], [Fowler, 2018]. Eles definem *code smells* como "estruturas no código que sugerem a possibilidade de refatoração" [Fowler et al., 1999]. A refatoração trata-se do esforço de reengenharia de software para mitigar os problemas que são identificados no nível de código e, consequentemente, aumentar a qualidade e manutenabilidade de um sistema.

As definições mais difundidas de *bad smells* na literatura são postuladas por Fowler et al. [1999], sendo um total de 22 *bad smells*. Por exemplo, o *bad smell* denominado *duplicate code* ocorre quando uma mesma estrutura de código aparece em dois ou mais lugares no código, comprometendo a manutenabilidade do sistema, pois, qualquer mudança feita na estrutura deverá ser replicada nos trechos em que aparecem

suas cópias. Fowler et al. [1999] detalham uma série de medidas de refatoração distintas para cada um dos *bad smells*. Por exemplo, *duplicate code* pode demandar que seja feita a extração de um método para reúso em diversas classes e, se possível for, colocar os campos utilizados em uma superclasse. Brown et al. [1998] estabelecem três momentos em que se pode encontrar *bad smells* durante o desenvolvimento de software: na gestão, na arquitetura ou no desenvolvimento. O primeiro se dá durante a gestão de pessoas e de processos, o segundo quando uma estratégia de arquitetura é definida e o último é proveniente das práticas de programação. Eles definem 14 *bad smells* na fase de desenvolvimento e suas possíveis refatorações, sendo sete considerados miniaturas de *bad smell*.

Uma quantidade excessiva de *bad smells* em um software dificulta a manutenção e a evolução dele [Sharma & Spinellis, 2018]. Identificar os trechos do sistema que devem ser refatorados é necessário para controlar a complexidade da arquitetura de software [Lanza et al., 2005]. A detecção de *bad smells* pode ser automática ou manual. A abordagem manual exige a inspeção do código por uma pessoa experiente que consiga discernir o que se trata de um *bad smell* ou não. Identificar *bad smells* manualmente em grandes sistemas de software não é viável. Assim, para superar essas dificuldades, muitos estudos têm sido realizados para definir estratégias e ferramentas para a detecção automática de *bad smells*. A abordagem automática mais implementada é baseada em métricas de qualidade de software e seus respectivos valores referência [Fernandes et al., 2016]. Um dos primeiros estudos nesse sentido é o de Marinescu [2001], que propõe uma abordagem baseada em métricas para detectar *bad smells*. O estudo de Fernandes et al. [2016] encontrou 84 ferramentas, que visam detectar ao todo 61 *bad smells*. No geral, essas ferramentas aplicam seis técnicas diferentes de detecção de *bad smells*. Os autores compararam quatro ferramentas de detecção dos *bad smells* *Large Class* e *Long Method*. Eles identificaram divergências entre os resultados das ferramentas ao inspecionar o mesmo conjunto de sistemas de software.

Os estudos geralmente utilizam *oráculos* para avaliar a eficiência das abordagens e ferramentas de detecção *bad smells*. Nesse contexto, um oráculo é um conjunto de dados considerado como referência, que é comparado a um conjunto resultante de mesma espécie para avaliar a qualidade desse conjunto resultante. Em relação aos *bad smells*, um oráculo é um conjunto de dados que relata instâncias reais de *bad smells* em um software. Por exemplo, o estudo de Fernandes et al. [2016] usou um oráculo de dois *bad smells* de um único sistema de software.

De acordo com Lavoie & Merlo [2011], bons oráculos precisam lidar com sistemas grandes o suficiente e dependem de uma análise minuciosa do código para ter algum interesse prático. Pode-se gerar um oráculo de *bad smells* por meio de (i) inspeção

manual de um sistema, (ii) varredura automática ou (iii) combinação de ambas as técnicas. Em uma abordagem manual, especialistas apontam instâncias de *bad smells* no código-fonte. Como os oráculos dependem da inspeção manual nessa técnica, sua subjetividade é alta (Palomba et al. [2015]). Portanto, é complexo gerar um oráculo confiável. Por outro lado, a avaliação de abordagens automáticas exige um oráculo anterior que precisa ser comprovadamente preciso. Portanto, a existência de oráculos confiáveis de *bad smells* é de importância ímpar no desenvolvimento de técnicas automáticas para detectar *bad smells*.

1.1 Definição do Problema

Existe uma extensa gama de ferramentas que visam detectar *bad smells* em sistemas de software. Fernandes et al. [2016] identificaram 84 ferramentas que, conjuntamente, afirmam encontrar 61 *bad smells* utilizando-se, ao menos, seis técnicas de detecção. No mesmo estudo, os resultados de quatro ferramentas se mostraram divergentes para os mesmos exemplares de dois sistemas de software. Sendo assim, paira uma dúvida sobre a eficácia das ferramentas propostas até então.

A avaliação de técnicas e ferramentas de detecção de *bad smells* pode ser realizada via comparação dos seus resultados com um oráculo. Dessa forma, a existência de oráculos de *bad smells* confiáveis é essencial para a realização de estudos sobre o assunto. Oráculos produzidos via inspeção manual de software são difíceis de serem encontrados e produzidos. Com isso, muitas vezes, os oráculos utilizados são gerados a partir de uma abordagem automática de detecção de *bad smell*. Porém, a própria avaliação dessas abordagens automáticas depende de comparação com dados de oráculos. Dezenas de trabalhos que se baseiam em detecção de *bad smells* utilizam tais ferramentas para coleta de dados e, consequentemente, desconhecer com maior precisão a eficácia delas é uma importante ameaça à validade de tais estudos [Fontana et al., 2012; Ouni et al., 2016a; Kaur et al., 2016]. Este trabalho visa contribuir para a solução deste problema, identificando e criando oráculos de *bad smells*.

1.2 Objetivos

O objetivo deste trabalho é identificar, criar e disponibilizar oráculos de *bad smells* para software orientado por objetos, visando contribuir com os estudos que são realizados sobre detecção de *bad smells*.

Os objetivos específicos deste trabalho são o seguintes:

1. Realizar de uma revisão sistemática da literatura para identificar os oráculos de *bad smells* disponíveis.

2. Desenvolver uma ferramenta para auxiliar a inspeção manual de software para identificação de *bad smells*.

3. Criar e disponibilizar um oráculo de *bad smells* para software Java.

1.3 Projeto do Estudo

Este trabalho foi realizado de acordo com os seguintes passos metodológicos:

1. Inicialmente, foi realizada uma SLR com o objetivo de catalogar os oráculos disponíveis.

2. Foi desenvolvida uma ferramenta, denominada `InspectBSmell`, para facilitar o especilista a assinalar ocorrências de *bad smell* dentro do próprio ambiente de inspeção, Eclipse. A ferramenta foi desenvolvida em Java.

3. A ferramenta foi aplicada para geração do oráculo proposto nesta dissertação. Ela foi usada por quatro especialistas para a inspeção de cinco programa Java, visando a identificação manual dos bad smells. Neste trabalho, foram considerados os bad smell *Large Class*, *Long Method*, *Data Class*, *Feature Envy* e *Refused Bequest*. Os quatro primeiros *bad smells* escolhidos figuram entre os mais populares, segundo a SLR realizada no primeiro passo metodológico. Acrescentamos *Refused Bequest* por ser um *bad smell* relevante para o quel não existe oráculo gerado via inspeção manual que o identifique.

4. O oráculo criado foi construído a partir das análises realizadas pelos especialistas.

5. Visando demonstrar e avaliar a aplicação do oráculo proposto, ele foi utilizado para avaliar os resultados gerados por três ferramentas de detecção de *bad smells* que têm sido empregadas em trabalhos experimentais sobre bad smells: `JDeodorant` ([Arcelli Fontana et al., 2012], [Christopoulou et al., 2012], [Terra et al., 2018]), `JSpIRIT` ([Fernandes et al., 2016], [Paiva et al., 2017]), `FindSmells` ([Sousa et al., 2017]).

1.4 Contribuições

Nesta dissertação, foram analisados cinco sistemas de software de código aberto em Java para identificar cinco *bad smells*: *Long Method*, *Large Class*, *Data Class*, *Feature Envy* e *Refused Bequest*. Após a identificação dos *bad smells*, realizamos um estudo empírico para verificar a acurácia e precisão de três ferramentas que afirmam fazer esse tipo de identificação: `JDeodorant` [Fokaefs et al., 2011], `JSpIRIT` (Vidalet al. [2015]) e `FindSmells` (Sousa et al. [2017]).

O processo de identificação de *bad smells* foi possibilitado por uma ferramenta desenvolvida no escopo do trabalho para facilitar o processo de inspeção manual de um software. A ferramenta `InspectBSmell` permite ao usuário assinalar presença de um *bad smell* em um trecho de código, deixar um comentário e fazer exportação em *JSON* e `csv`.

Uma Revisão Sitemática da Literatura (SLR) identificou a presença de apenas 12 oráculos de *bad smells* disponíveis *online*. Dentre esses, o oráculo desenvolvido por Palomba et al. [2015] é considerado o mais amplo, provendo recursos que permitem a inclusão de novas instâncias de *bad smells*. O oráculo produzido por este trabalho foi incluído no repositório online Landfill [Palomba et al., 2015] e está disponível para consulta.

Oracles of Bad Smells - a Systematic Literature Review - artigo aceito para publicação na trilha *Research* do Simpósio Brasileiro de Engenharia de Software de 2020 (SBES 2020).

Os resultados deste trabalho poderão ser utilizado pela comunidade acadêmica e até mesmo pela indústria por fornecer um oráculo verificado de *bad smells* para ser aproveitado nos testes e validação de ferramentas que se propõem a identificar *bad smells* em sistemas.

1.5 Estrutura da Dissertação

Neste capítulo, introduziu-se o problema, os objetivos do trabalho e as contribuições desta dissertação de mestrado. A seguir é apresentada a organização dos demais capítulos.

Capítulo 2 detalha os fundamentos teóricos indispensáveis para a compreensão do trabalho: os conceitos de *bad smell* e oráculo. Esse capítulo também detalha as definições de 35 *bad smells* amplamente difundidas no meio acadêmico.

Capítulo 3 apresenta a Revisão Sistemática da Literatura sobre Oráculos, as cinco questões de pesquisa que direcionaram essa revisão, a metodologia aplicada e os

resultados obtidos.

Capítulo 4 mostra o projeto e desenvolvimento da ferramenta `InspectBSmell`, criada para auxiliar a geração de um oráculo de *bad smells*.

Capítulo 5 descreve o oráculo de *bad smells* proposto e desenvolvido neste trabalho de dissertação e discute seus resultados.

Capítulo 6 faz uma avaliação de três ferramentas de identificação de *bad smells* quando confrontadas com o oráculo de *bad smells* construído pelo presente trabalho.

Capítulo 7 identifica as ameaças à validade existentes no presente trabalho e os aprendizados colhidos no processo.

Capítulo 8 apresenta a conclusão, as contribuições deste trabalho de dissertação e também propõe os trabalhos futuros.

Capítulo 2

Fundamentação Teórica

O presente capítulo aborda os principais conceitos aplicados neste trabalho. Ele é dividido em duas seções: *Bad Smell* e Oráculo. Dentre os trabalhos existentes sobre *bad smells*, duas grandes obras foram escolhidas para trazerem as definições e especificações de um conjunto relevante de *bad smell*, as de Fowler et al. [1999] e Brown et al. [1998]. É importante notar que o tema principal de ambas as obras é a refatoração do código e, consequentemente, sua melhoria em termos de qualidade. Porém, a identificação dos *bad smells* é a primeira etapa no processo de abrandar as anomalias de um sistema. Seção 2.2 mostra a descrição de um oráculo de *bad smells* e apresenta alguns dos principais oráculos disponíveis na literatura. Finalizando este capítulo, Seção 2.3 discute sobre os oráculos apresentados e o porquê da proposição de um novo oráculo nesta dissertação de mestrado.

2.1 *Bad Smell*

Bad smells são sintomas que podem indicar um problema mais profundo no projeto de um software. Segundo Brown et al. [1998], um *bad smell* pode ocorrer quando um desenvolvedor ou arquiteto de software desconhece soluções eficientes para a solução de um problema e acabam por optar a fazê-lo de forma errada para o contexto envolvido. Pode ocorrer também do projetista utilizar uma solução bem conhecida e funcional, porém em cenário errado. Por outro lado, é possível que uma classe apresente todos os indícios de um *bad smell* e não necessariamente aquela implementação precisa ser refatorada. Um exemplo disso, dado por Khomh et al. [2011], são os analisadores sintáticos gerados automaticamente. Apesar de apresentarem vários sintomas de *Spaghetti Code*, i. e., classes com uma quantidade considerável de métodos longos e complexos, somente um analista de qualidade consegue determinar se a implementação de um analisador

sintático possui um *bad smell* ou não.

Fowler et al. [1999] e Fowler [2018], após a análise dos códigos de diversos autores e domínios, determinaram uma série de estruturas de código que sugerem a possibilidade de refatoração. Todavia, em ambas referências, os autores salientam que os critérios apresentados por eles são apenas indicadores e que nada, nem mesmo uma série de métricas, pode substituir a análise subjetiva humana.

Sobrinho et al. [2018] conduziram uma extensiva Revisão Sistemática da Literatura para determinar qual o estado da arte sobre *bad smells* entre 351 estudos sobre o tema produzidos entre os anos 1990 e 2017. Dentre os resultados, os autores identificaram que existem divergências a respeito do impacto de *bad smells* na manutenibilidade de software. Em alguns casos, um código com uma instância de *bad smell* é a melhor opção de desenvolvimento. O mesmo estudo indica que a motivação mais comum dos trabalhos analisados, mais de 30% do conjunto de estudos, consiste na criação de uma ferramenta ou abordagem de detecção de *bad smells*.

2.1.1 *Bad Smells* Definidos por Fowler et al.

De acordo com Fowler et al. [1999], um software pode estar sujeito a qualquer um dentre os 22 *bad-smells* encontrados e descritos por eles. Para cada *bad smell* levantado, os autores também especificaram um conjunto de técnicas de refatoração, descrevendo quais estratégias de refatoração são aplicadas para resolvê-los. Recentemente, Fowler [2018] propôs uma das listas mais completas contendo 24 *bad smells*; além de disponibilizar um dos mais conhecidos e completos catálogos com uma lista de 61 refatorações, apesar de sua página na Web[1] exibir um catálogo com 66 refatorações. A seguir, é dada uma breve explicação do que se trata cada um dos *bad smell* segundo Fowler et al. [1999].

- ***Duplicate Code***: ocorre quando uma mesma estrutura de código aparece em dois ou mais lugares é sinal de que o código necessitará ser refatorado. Se for necessário fazer uma alteração em um dos trechos da estrutura, provavelmente ela deverá ser replicada para as demais cópias.

- ***Long Method***: métodos longos podem ser um problema porque quanto maior um procedimento, mais difícil é para ele ser entendido.

- ***Large Class***: está relacionado a classes que tentam fazer muitos serviços e que frequentemente possuem um grande número de variáveis.

[1] https://refactoring.com/catalog/

- ***Long Parameter List***: listas de parâmetros muito grande são difíceis de serem compreendidas.
- ***Divergent Change***: um software necessita ser estruturado de forma a facilitar a sua manutenção. Se uma classe é modificada em diferentes formas por diferentes razões, talvez valha a pena dividi-la em duas, para que cada nova classe se relacione com um tipo diferente da mudança.
- ***Shotgun Surgery***: se um tipo de alteração em um programa necessita de muitas modificações pequenas em várias classes, será difícil encontrar os lugares certos para fazer toda manutenção necessária.
- ***Feature Envy***: quando um método em uma classe parece mais interessado em atributos, usualmente dados, de outra classe que não é a sua própria, talvez o método devesse estar na outra classe.
- ***Data Clumps***: ocorre quando um mesmo grupo de dados é sempre visto junto em vários lugares, por exemplo, campos de uma classe, parâmetros para vários métodos ou dados locais. Talvez esses dados devessem ser agrupados em uma mesma pequena classe.
- ***Primitive Obsession***: vale a pena tornar um tipo primitivo de dados em uma pequena classe para deixar mais claro quais os seus propósitos e as operações que lhe são permitidas. Por exemplo, criar uma classe `Data` em vez de utilizar uma variável do tipo inteiro para dia, mês e ano.
- ***Switch Statements***: declarações de *switch* tendem a causar duplicação de código. Provavelmente, mesmas operações de *switch* são utilizadas em mais de um lugar.
- ***Parallel Inheritance Hierarchies***: ocorre sempre que for necessário criar uma subclasse de uma classe e, consequentemente, uma nova subclasse também precisará ser criada em outra classe para equivaler à modificação.
- ***Lazy Class***: classes que não estão fazendo muitos serviços deverão ser eliminadas.
- ***Speculative Generality***: frequentemente métodos ou classes que são planejados para executar tarefas que de fato não são necessárias.
- ***Temporary Field***: pode ser difícil determinar quando algumas das variáveis de uma classe são utilizadas ocasionalmente.

- ***Message Chains***: um cliente pede um objeto de outro objeto, o qual solicita outro objeto que, por sua vez, também é solicitado por outro, dificultando o entendimento do fluxo de um programa.
- ***Middle Man***: delegação é frequentemente útil, mas, às vezes, não é possível ir muito além. Se uma classe está atuando como um delegado, mas está realizando trabalho inútil, provavelemente é possível removê-la da hierarquia.
- ***Inappropriate Intimacy***: ocorre quando classes parecem gastar muito tempo analisando partes privadas de outras.
- ***Alternative Classes with Different Interfaces***: classes que executam tarefas similares, mas possuem assinaturas diferentes, devem ser modificadas para compartilhar um protocolo comum.
- ***Incomplete Library Class***: não é uma boa prática alterar uma biblioteca, mas às vezes ela não faz tudo o que deveria ser feito.
- ***Data Class***: classes que possuem apenas campos e métodos acessíveis, mas nenhum comportamento real. Se um dado é público, o correto é que se torne privado.
- ***Refused Bequest***: se uma subclasse não deseja ou precisa de todo o comportamento de sua classe base, provavelmente a estrutura da hierarquia está errada.
- ***Comments***: se comentários são presentes em um código por se tratar de um código ruim, é necessário melhorá-lo.

2.1.2 *Bad Smells* Definidos por Brown et al.

Brown et al. [1998] classificam os *bad smells* sob três perspectivas: a perspectiva do desenvolvedor, do arquiteto e do gerente de projeto. Os problemas sob o prisma do desenvolvedor envolvem problemas técnicos e soluções que são encontradas durante a codificação. Apesar da importância das demais perspectivas, por se referirem à estruturação de sistemas, processos e o desenvolvimento das organizações, o presente trabalho de mestrado tem como foco *bad smells* referentes a código fonte, que são descritos a seguir:

- ***Blob***: uma estrutura procedimental que encaminha para um único objeto a maior parte das responsabilidades, enquanto o restante dos objetos somente servem para prover dados e executar processos simples.

- ***Continuous Obsolescence***: alterações que são justificadas pela atualização de tecnologias. Cada nova atualização dificulta a compreensão do código pelos desenvolvedores.

- ***Lava Flow***: trechos de código morto e informações de projeto esquecidos no código que são mantidos mesmo após atualizações de versão.

- ***Ambiguous Viewpoint***: *viewpoints* que não permitem a separação fundamental entre interfaces e detalhes da implementação, o qual é um dos benefícios primários do paradigma da orientação por objetos.

- ***Functional Decomposition***: esse *bad smell* é o resultado de uma implementação feita por programadores experientes e que desconhecem o padrão da orientação por objeto. O código acaba por reproduzir uma linguagem estruturada como, por exemplo, Pascal e FORTRAN em uma estrutura de classes.

- ***Poltergeists***: são classes com papéis limitados e ciclo de vida efetivo. Frequentemente são usadas para disparar processos para outros objetos.

- ***Boat Anchor***: é um trecho de um software que não possui qualquer propósito válido no projeto.

- ***Golden Hammer***: é uma tecnologia ou conceito aplicado obsessivamente para muitos problemas do software.

- ***Dead End***: aparece devido a modificação de um componente reusado quando o componente modificado não recebe mais manutenção e suporte pelo fornecedor.

- ***Spaghetti Code***: estrutura *ad hoc* que acaba por deixar o código difícil de ser estendido e de ser otimizado.

- ***Input Kludge***: um software que falha em testes de comportamento simples, que ocorre quando algoritmos são empregados para processar a entrada de programas.

- ***Walking through a Minefield***: inúmeros *bugs* são encontrados em *releases* de software.

- ***Cut-and-Paste Programming***: reúso de software pela cópia de código encaminha o programa para problemas significantes de manutenção.

2.2 Oráculos

O termo "oráculo" é utilizado na literatura para denotar um conjunto de dados considerados verdadeiros acerca de determinada característica de um software. Oráculo é, portanto, um *benchmark* para se avaliar determinado aspecto de software. Por exemplo, na área de testes de software, o oráculo é um mecanismo capaz de determinar se o resultado de um teste está ou não de acordo com os valores esperados. Frequentemente, assume-se que o próprio projetista do teste seja o responsável por esta tarefa. Li & Offutt [2017] definem, nesse contexto, que uma estratégia de oráculo para teste é uma regra ou um conjunto de regras, definido por uma precisão e uma frequência, que especifica quais estados de um programa devem ser verificados.

Oráculos são utilizados para verificar e validar o desenvolvimento de ferramentas de detecção de *bad smells*. Um oráculo para *bad smells* é gerado via inspeção de uma versão de um sistema pela varredura, manual, automática ou uma combinação das duas, de seu código fonte na tentativa de encontrar *bad smells*. Ao conjunto de *bad smells* e sua localização no projeto dá-se o nome oráculo, porque, assim como um oráculo para teste de software, a coleção de *bad smells* já está pré-definida e constitui-se da saída esperada após a execução correta de uma ferramenta que alega ter encontrado *bad smells*. Um bom oráculo, segundo Lavoie & Merlo [2011], deve lidar com um sistema grande o suficiente para ter qualquer interesse prático e dependerá da análise exaustiva do código. De qualquer forma, vale estabelecer a lógica inversamente proporcional entre o tamanho de um software, que determina a relevância do oráculo, e a facilidade em criá-lo.

Os oráculos existentes para consulta variam de domínio, tamanho, linguagem de programação, idade do software e dos conjuntos de *bad smells* que foram escolhidos para análise dentre todos os outros *bad smells* existentes. Sobrinho et al. [2018] apontaram uma ampla diversidade de trabalhos *open source* utilizados nos estudos empíricos sobre *bad smell*. O trabalho também indica que existe uma lacuna de oráculos de *bad smell* representativos. Por fim, os autores apontam para a necessidade de criar novos oráculos de *bad smell* em software.

Há trabalhos que criaram oráculos manualmente, dentre eles destacam-se os de Khomh et al. [2009b], Vale & Figueiredo [2015], Palomba et al. [2013a], Palomba et al. [2014] e Palomba et al. [2015]. Outros trabalhos também criaram oráculos por meio de abordagens mistas ou puramente automáticas. Um exemplo de trabalho que criou um oráculo pela conjução de ferramentas automáticas e a revisão manual é de Palomba et al. [2017]. Nesse estudo, os autores investigaram a relação existente entre a ocorrência de *bad smell* em projetos de software, alteração de software e propensão a erros. Outros

trabalhos utilizaram de uma abordagem automática, como o de Palomba et al. [2017a] e Saboury et al. [2017], comumente via métricas de qualidade de software. Geralmente os trabalhos que definem esses oráculos têm como objetivo principal propor ferramentas de coletas de *bad smells*.

Muitas técnicas para detecção de *bad smells* são desenvolvidas e a maioria delas é validada por uma comparação dos *bad smells* candidatos com um oráculo previamente montado manualmente. Entretanto, poucos são os conjuntos de *bad smells* manualmente validados [Palomba et al., 2015].

Vários outros projetos realizam uma análise de *bad smells* no nível de código, porém não geram ou não indicam quaisquer recursos disponíveis para consulta desses dados. São os casos dos trabalhos de Aniche et al. [2016], Yamashita [2014a], Mantyla et al. [2004], Oizumi et al. [2016a], Ouni et al. [2015a] e Fontana et al. [2012]. O último trabalho, por exemplo, utilizou-se de ferramentas automáticas de detecção de *bad smells*, *iPlasma* (Marinescu et al. [2005]), *Google CodePro Analytix* (Clayberg & Rubel [2008]) e *EclipseMetrics* (Walton [1999]), para 12 sistemas de software de código aberto desenvolvidos em Java. Apesar de existir uma tabela com os valores sumarizados de ocorrência dos *smells*, não foram encontrados detalhes sobre em quais pontos do software estavam os *bad smells*.

Os trabalhos citados nesta seção, dentre outros, foram identificados por meio da Revisão Sistemática da Literatura, que foi conduzida com o propósito de identificar quais são os oráculos de *bad smells* disponíveis na literatura. Os resultados da SLR estão apresentados no Capítulo 3.

2.3 Considerações Finais

O presente capítulo tratou da definição de 36 *bad smells* oriundos de dois livros sobre a refatoração de código. Dado que a qualidade de um software tende a cair ao longo do processo de manutenção do sistema, após a primeira entrega, se faz necessário um conjunto de medidas para a melhoria do código. A refatoração consiste de uma coleção de procedimentos que propõe-se a extinguir práticas ruins de codificação. Por exemplo, o reúso por cópia de código é uma abordagem que, ao longo do tempo, dificulta a manutenção. Este trabalho não aprofundará nas práticas de refatoração, porém é possível encontrar conteúdo mais aprofundado nas bibliografias citadas [Fowler et al., 1999; Fowler, 2018; Brown et al., 1998].

Este capítulo apresentou também a conceituação de oráculos e, mais especificamente, oráculos de *bad smells*. Construir um oráculo demanda a inspeção manual

de cada uma das classes de um sistema procurando por instâncias de *bad smells*. As abordagens puramente manuais são dispendiosas em termos de tempo e esforço. Palomba et al. [2015] identificam como principais dilemas a serem vencidos na construção de um oráculo manual, a subjetividade, dificuldade de comparar técnicas diferentes, a generalização dos resultados e o esforço envolvido na construção de um oráculo. Frequentemente os especialistas não chegam a um consenso sobre um *bad smell* e, por essa razão, pode ser que oráculos gerados para um mesmo software possuam divergências.

Considerando todas as dificuldades existentes para gerar um oráculo, existem poucos trabalhos que o fazem. Palomba et al. [2015] afirmam que desconhecem repositório de *bad smells* mais amplo que o gerado por eles, cujo volume compreende 20 sistemas de software. Além disso, nos demais trabalhos, a maioria dos oráculos disponíveis estão limitados a pequenos conjuntos de *bad smells*. Assim, com o objetivo de formar um catálogo de oráculos de *bad smell* mais abrangente, este trabalho realizou uma Revisão Sistemática da Literatura (SLR) sobre oráculos que é apresentada no Capítulo 3. Além disso, com o intuito de contribuir com a literatura, aumentando a coleção de *bad smells* validados manualmente, neste trabalho foi criado um oráculo referente a cinco bad smells com dados de cinco sistemas Java de código aberto. O oráculo é apresentado no Capítulo 5.

Capítulo 3

Revisão Sistemática da Literatura sobre Oráculos de *Bad Smells* em Software

O presente capítulo apresenta uma Revisão Sistemática da Literatura (SLR) conduzida neste trabalho para encontrar os oráculos de *bad smells* disponíveis na literatura e suas características. O objetivo desta SLR é identificar os oráculos de *bad smells* propostos na literatura, como foram construídos, suas principais características e onde estão disponíveis. Para tanto, realizamos uma Revisão Sistemática da Literatura. Como resultado, identificamos 51 estudos que produzem direta ou indiretamente oráculos de *bad smells*. Pelo menos seis linguagens de programação foram o foco dos oráculos: Java, C, C++, JavaScript e Scratch. Também descobrimos que a abordagem automática é a mais popular, *i.e.*, normalmente os oráculos de *bad smells* dependem de ferramentas. JDeodorant é a ferramenta mais usada para criar oráculos de *bad smells* [Fokaefs et al., 2011]. Além disso, cada oráculo encontrado na literatura concentra-se em *bad smells* específicos, onde o *bad smell Blob* é o mais usado. Dessa forma, a principal contribuição deste estudo é um catálogo de oráculos de *bad smells* que pode ser útil para pesquisas sobre *bad smell*, especialmente estudos que propõem ferramentas ou estratégias de detecção de *bad smells*.

Organizamos o restante deste capítulo da seguinte forma. Seção 3.1 descreve o protocolo utilizado para conduzir esta SLR e apresenta as questões da pesquisa. Seção 3.2 relata os resultados obtidos e responde as questões de pesquisa. Seção 3.3 discute sobre os oráculos encontrados com a SLR realizada. Seção 3.4 contem as considerações finais.

3.1 Metodologia

Segundo Kitchenham & Charters [2007], o objetivo de uma Revisão Sistemática da Literatura é "identificar, avaliar e interpretar todas as evidências disponíveis relevantes para uma questão específica, área temática ou fenômeno de interesse". Os estudos identificados por uma SLR podem ser classificados como estudos primários, enquanto a própria SLR é um estudo secundário. Neste trabalho, os estudos primários são publicações científicas.

Realizamos a SLR em três etapas: planejamento, execução e análise. Seção 3.1.1, apresenta a etapa de planejamento, as questões de pesquisa e a *string* definida para a busca. Seção 3.1.2 descreve a execução, mostrando as etapas e os resultados do processo de seleção de estudos primários. Seção 3.1.3 exibe os pontos mais relevantes de alguns trabalhos selecionados.

3.1.1 Etapa de Planejamento

Nesta etapa, definimos todos os elementos necessários para a elaboração da SLR:

- as questões de pesquisa que investigaremos
- a cadeia de pesquisa que usaremos
- os critérios de inclusão e exclusão dos estudos primários, e
- os repositórios digitais a serem pesquisados.

Questões de Pesquisa. Este estudo investiga cinco questões de pesquisa (QPn), como segue.

QP1 - Quais oráculos de bad smells foram propostos na literatura?

QP2 - Em quais linguagens de programação os sistemas que compõem os oráculos são implementados?

QP3 - Qual é o tamanho dos sistemas que compõem os oráculos?

QP4 - Quais bad smells são considerados pelos oráculos?

QP5 - Quais abordagens foram usadas para criar os oráculos?

***String* de Busca.** A *string* de busca é usada para encontrar estudos primários nos repositórios digitais selecionados. Para definir a *string* de busca, identificamos as palavras-chave mais relevantes relacionadas às questões de pesquisa propostas e os sinônimos dessas palavras-chave.

"*Bad smell*" não é o único termo usado na literatura. Outros termos, como "*code smell*" e "*anomaly*", têm o mesmo significado; portanto, eles também foram incluídos na definição da *string* de busca da forma que se segue.

> *((“oracle"OR “benchmark") AND ((“anti-pattern"AND “software") OR (((“bad"OR “design"OR “code"OR “architecture") AND (“smell")) OR ((“design"OR “code") AND “anomaly"))))*

Critérios de inclusão e exclusão. Esses critérios nos permitem classificar cada artigo encontrado como candidato a ser incluído ou excluído da SLR, permitindo que ela se restrinja ao tópico explorado. A tabela 3.1 resume os critérios de inclusão e exclusão aplicados neste trabalho.

Tabela 3.1. Critérios de Inclusão e Exclusão da SLR

Critérios de Inclusão	**Critérios de exclusão**
Publicações em Ciência da Computação	Artigos Duplicados
Trabalhos disponíveis em formato eletrônico	Documentos classificados como tutoriais, pôsteres, painéis, palestras ou workshops
Trabalhos escritos em inglês	Artigos que não foram encontrados
Trabalhos relacionados ao tema deste estudo	Capítulo de livros publicados em conferências ou revistas
Publicações em conferências ou revistas	

Para pesquisar em diferentes repositórios digitais, aplicamos a *string* de busca apenas em metadados, como título, resumo e palavras-chave. Durante o processo de decisão da *string* de busca, fizemos alguns ajustes nos termos considerados. Após a primeira rodada de pesquisa, adicionamos o termo *benchmark* como sinônimo de *oracle*. O termo *anti-pattern* foi o último inserido na *string*, pois percebemos que existem trabalhos que utilizam o termo como sinônimo de *bad smell*. No entanto, para limitar o universo em que esse termo aparece, inserimos o termo software. As pesquisas foram realizadas com todas as propostas de *string* de busca e consideramos seus resultados como base dos estudos primários incluídos na SLR.

Repositórios Digitais. Usamos os seguintes repositórios digitais: ACM Digital Library[1], IEEE Xplore[2], Science Direct[3](SD), Scopus[4], Springer[5], Web of Science[6], Engi-

[1]http://dl.acm.org/
[2]http://ieeexplore.ieee.org/
[3]http://www.sciencedirect.com/
[4]http://www.scopus.com/
[5]https://link.springer.com/
[6]http://webofknowledge.com/

neering Village[7], e Google Scholar[8]. Nós os escolhemos porque são repositórios eletrônicos que possuem uma extensa coleção de artigos completos publicados em conferências e periódicos relevantes para a comunidade acadêmica.

3.1.2 Etapa de Execução

Esta etapa consiste em aplicar a *string* de busca nos repositórios digitais escolhidos e os critérios de inclusão e exclusão dos estudos.

Processo de Pesquisa. Aplicamos a *string* de busca ao mecanismo de pesquisa de cada repositório digital e incorporamos os resultados em uma única planilha para análise subsequente. Nos resultados do `Science Direct` e `Web of Science`, os dados foram incluídos na planilha manualmente, porque essas bibliotecas digitais não exportaram dados para o formato de planilha. Para recuperar dados do `Google Scholar`, usamos o programa Publish or Perish[9], que fornece uma lista de artigos relevantes que se encaixam na *string* de busca. Observe que o usamos porque o `Google Scholar` não fornece a funcionalidade para baixar os resultados como um todo.

No total, foram retornados 4.948 estudos primários, conforme exibido na Tabela 3.2. O `Science Direct` retornou 1.272, quase oito vezes mais que o `Web of Science`, que retornou o menor número de estudos, 164.

Tabela 3.2. Número de trabalhos retornados por cada repositório digital

Digital Library	**Número de trabalhos**
Google Scholar (Google)	901
Scopus (SC)	804
Engineering Village (EV)	240
ACM Digital Library (ACM)	210
IEEE Xplore (IEEE)	348
Web of Science (WoS)	164
Springer (SP)	1009
ScienceDirect (SD)	1272

Processo de Seleção. Após realizar o processo de busca, selecionamos os estudos primários a serem considerados. Para esse fim, usamos os critérios de inclusão e exclusão definidos.

[7]http://www.engineeringvillage.com/
[8]https://scholar.google.com.br/
[9]https://harzing.com/resources/publish-or-perish

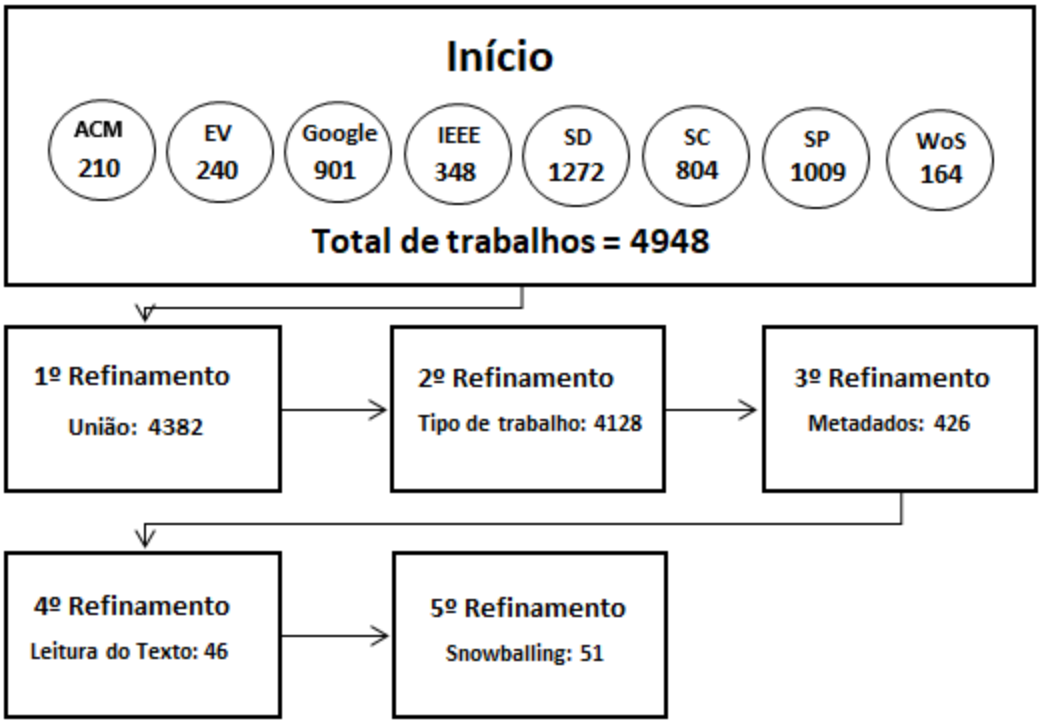


Figura 3.1. Fases de seleção dos trabalhos.

A Figura 3.1 ilustra as fases definidas para a seleção dos trabalhos após a execução da sequência de pesquisa nos oito repositórios digitais. A pesquisa inicial retornou 4.948 estudos. Realizamos cinco refinamentos durante a etapa de execução. O objetivo desses refinamentos é selecionar um conjunto final de estudos primários que apresentem um oráculo de *bad smells*. O primeiro refinamento removeu documentos duplicados, resultando em 4382 estudos. Descartamos a duplicação de estudos, ou seja, trabalhos com o mesmo título, ano e autores. O segundo refinamento eliminou estudos cujo tipo de trabalho não enquadrava no escopo desta SLR. Removemos estudos representando capítulos de livros ou registros de patentes. Após essa etapa, restaram 4.128 estudos. No terceiro refinamento, analisamos os demais estudos, verificando se eles atendiam ao objetivo desta revisão sistemática da literatura. Para isso, lemos os títulos, resumos e conclusões dos artigos. No fim dessa fase, restavam 426 estudos.

Posteriormente, realizamos o quarto refinamento. Nesta fase, lemos todos os estudos restantes completamente para garantir que eles comprende o escopo da nossa SLR. A leitura do texto foi conduzida com o intuito de encontrar estudos que produziam oráculos de *bad smell*. Nesse contexto, foram procurados trechos do texto que descrevessem algum processo de análise de software e a criação de uma lista indicando a ocorrência de um *bad smell* em um método ou classe. O conjunto resultante possui 46

estudos. Em seguida, realizamos o procedimento conhecido como *snowballing* (Wohlin [2014]) com os 46 estudos. Neste quinto refinamento, verificamos a lista de referência de cada estudo. O objetivo foi encontrar outros estudos não incluídos no conjunto existente, mas relacionados a oráculos de *bad smells*. Foram incluídos cinco estudos após esta etapa, resultando em 51 estudos. Selecionamos esses 51 estudos primários para a Etapa de Análise.

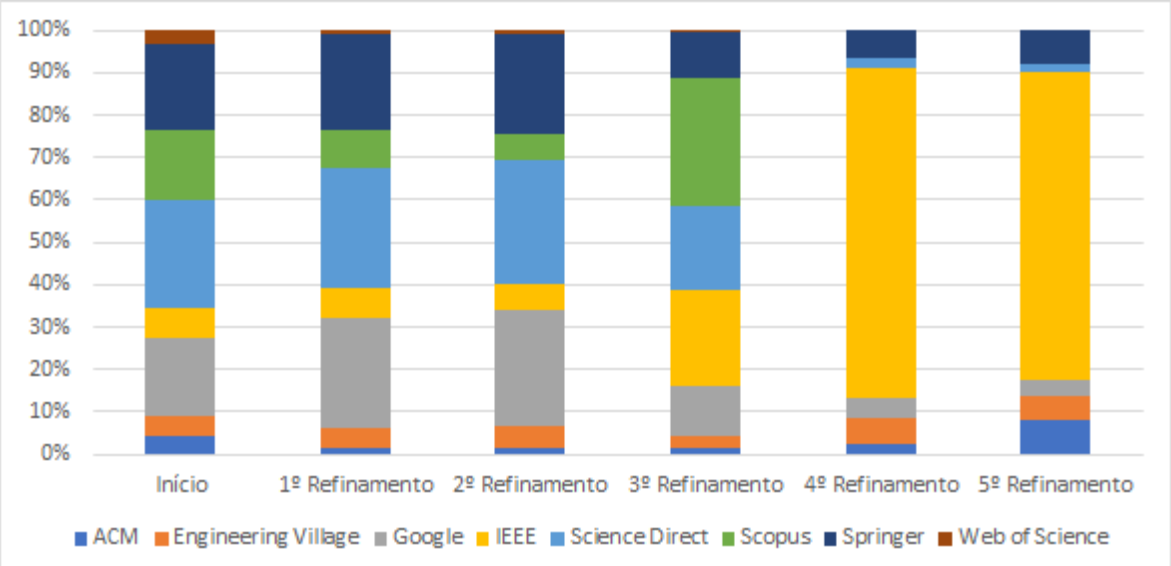


Figura 3.2. Distribuições de trabalho após cada fase de refinamento

A Figura 3.2 ilustra a distribuição dos trabalhos retornados após cada fase de refinamento agrupados os dados por repositório digital. O `IEEE Xplore` é a biblioteca que mais contribuiu para a pesquisa, representando mais da metade do total. Apesar de retornar a maior quantidade de resultados iniciais, o `Science Direct` fornece apenas um artigo para o estudo. Os repositórios digitais `Scopus` e `Web of Science` acabaram não tendo nenhum artigo considerado para o estudo. Vale ressaltar que o primeiro refinamento levou em consideração a ordem alfabética do repositório, o que pode ter mascarado as contribuições.

3.1.3 Etapa de Análise

Nesta etapa, analisamos os resultados para responder às questões de pesquisa com os dados extraídos dos 51 estudos primários selecionados na Etapa de Execução.

Extração de Dados. Realizamos uma leitura cuidadosa dos 51 estudos primários. Para cada estudo, escrevemos um resumo compilando os dados necessários para respon-

der às questões de pesquisa. Em seguida, criamos uma planilha contendo o nome do *bad smell*, a abordagem usada para identificar o *bad smell* e as características dos sistemas de software considerados no estudo. Nesta fase, descobrimos que só era possível saber se o oráculo está disponível *online* quando o artigo relata essa informação.

Resultado Final. Após o quinto refinamento, foram encontrados 51 artigos distribuídos entre os repositórios digitais, conforme Figura 3.3. Dois repositórios digitais não retornaram nenhuma contribuição para o tema de pesquisa desta SLR, a saber `Scopus` e `Web of Science` e mais de 70 % dos estudos são do `IEEE`. No entanto, não é possível afirmar que os estudos são resultados exclusivos do `IEEE`, já que a primeira etapa do refinamento classificou as bases em ordem alfabética, o que pode ter beneficiado de alguma forma o `IEEE`.

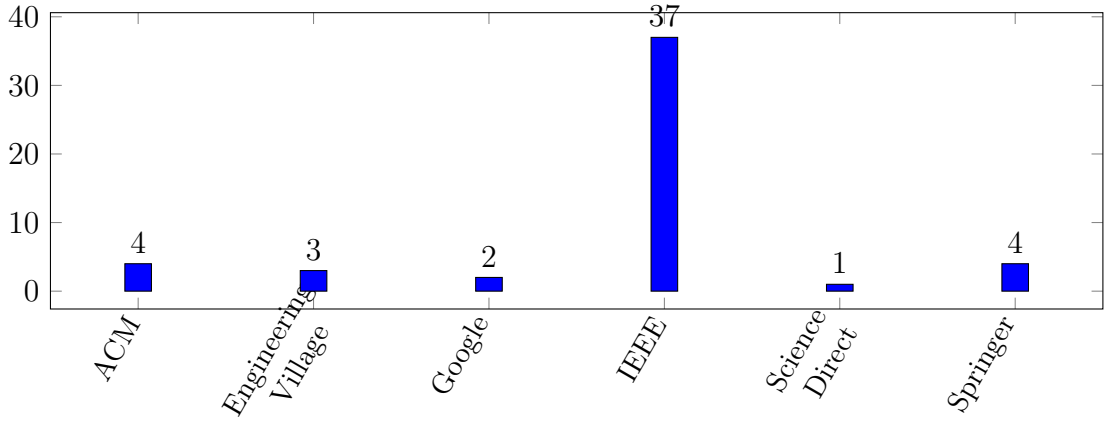


Figura 3.3. Distribuições finais dos estudos entre os repositórios digitais

Estudos publicados fora dos ambientes acadêmicos convencionais são conhecidos por "literatura cinzenta" ([Börjesson, 2016]). Esse tipo de estudo primário pode ser encontrado por ferramentas como o Google Scholar. Google Scholar também é capaz de encontrar artigos disponibilizados por outros repositórios digitais. Sabendo que todos os resultados são filtrados pelas etapas de refinamento, fomos capazes de determinar quais eram aqueles que possuiam relevância para responder as questões de pesquisa. O uso do Google Scholar é justificado por que o repositório "digitar"cooperou com dois estudos no conjunto final de estudos primários considerados.

A Figura 3.4 mostra a distribuição dos trabalhos selecionados publicados ao longo dos anos. Essa análise considera todo o conjunto de estudos primários, incluindo os cinco encontrados pelo processo de *snowballing*. Observamos que pesquisas relacionadas aos oráculos de *bad smells* são recentes. O ápice das publicações ocorreu entre 2015

e 2016. O primeiro oráculo encontrado foi publicado em 2004, cinco anos após as definições de *bad smells* fornecidas por Fowler et al. [1999].

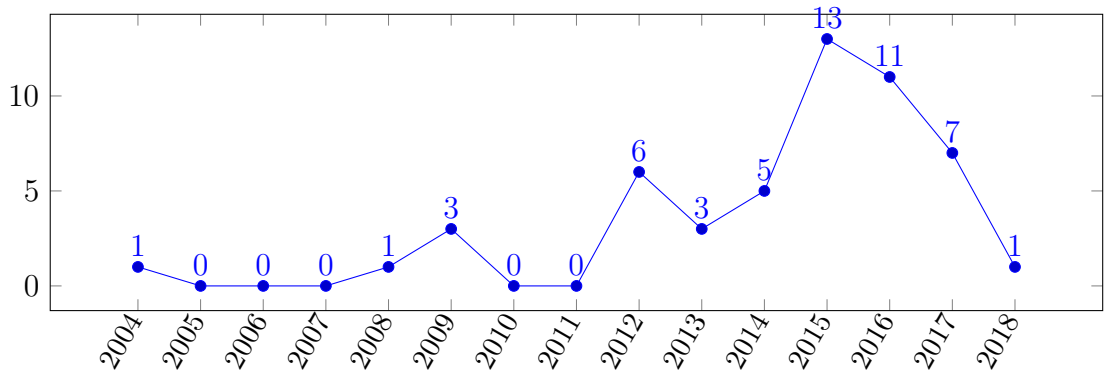


Figura 3.4. Distribuição dos estudos por ano de publicação

3.2 Resultados

Esta seção apresenta os resultados do estudo, respondendo às perguntas da pesquisa.

QP1: Quais oráculos de bad smells foram propostos na literatura?

Todos os 51 estudos primários recuperados na busca apresentam um oráculo de *bad smell*. No entanto, nem todos os oráculos têm dados disponíveis *online*. Em alguns casos, os autores esclarecem que podem deixar os oráculos disponíveis se receberem um pedido formal. A Tabela 3.3 apresenta os estudos que definem os oráculos de *bad smells*. Apenas 12 oráculos estão *online*. Entre eles, quatro usaram uma abordagem manual para sua geração, seis oráculos foram geradas por ferramentas e outros dois usaram uma abordagem mista para sua geração. A Tabela 3.4 e a Tabela 3.5 apresentam as características deles. Da esquerda para a direita, as colunas indicam:

- referência ao trabalho que propôs o oráculo
- os sistemas de software nos quais o oráculo está baseado
- os *bad smells* considerados no oráculo
- a abordagem aplicada para construir o oráculo - manual, ferramenta ou misto -
- o *link* para o oráculo.

Em alguns casos, mesmo quando um artigo menciona um *link* para um oráculo, ele pode não ser encontrado. Isso acontece no caso de Maiga et al. [2012] cuja página está disponível, mas nenhum oráculo foi encontrado, mesmo após uma pesquisa minuciosa.

Tabela 3.3. Lista de artigos que não possuem oráculo *online*

Abordagens	**Referências**
Ferramentas	Fard & Mesbah [2013], Yamashita [2014b], Macia et al. [2012], Mannan et al. [2016], Szőke et al. [2014], Wagey et al. [2015], Aniche et al. [2016], Sirikul & Soomlek [2016], Mantyla et al. [2004], Oizumi et al. [2016b], Danphitsanuphan & Suwantada [2012], d. Reis et al. [2016], Arcelli Fontana et al. [2016], Kessentini & Ouni [2017], Chen et al. [2016], Abílio et al. [2015], Peters & Zaidman [2012], Amorim et al. [2015], Ouni et al. [2015b], Yamashita et al. [2015], Fontana et al. [2012], Ouni et al. [2016b], Kaur et al. [2016], Palomba et al. [2018], Yamashita & Moonen [2013], Fontana et al. [2015], Oizumi et al. [2014]
Mista	Vale et al. [2015], Olbrich et al. [2009], Fenske et al. [2015]
Manual	Hermans & Aivaloglou [2016], Olbrich et al. [2009], Vale & Figueiredo [2015], Sahin [2016], Zhao et al. [2015], Falke et al. [2008], Palomba et al. [2015], Palomba [2015], Dhambri et al. [2008]

O *link* fornecido inicialmente por Palomba et al. [2015] não está mais disponível, mas encontramos o *link* atual[16] após a pesquisa.

Observamos que o pesquisador Fabio Palomba é coautor de cinco dos doze trabalhos que resultaram em oráculos online (Palomba et al. [2018], Palomba et al. [2013b], Palomba et al. [2015], Palomba et al. [2017b], Palomba et al. [2016]). Os oráculos nessas obras têm muitos sistemas de software e *bad smells* em comum. Portanto, eles podem compartilhar dados semelhantes.

Os oráculos de *bad smell* disponíveis organizam os resultados por software. A maioria dos estudos fornece os dados em planilhas com uma linha por classe e os *bad smells* encontrados na classe (Khomh et al. [2009b], Palomba et al. [2016], Khomh et al. [2009a], Palomba et al. [2013b], Palomba et al. [2017b], Palomba et al. [2018]). O oráculo de Hecht et al. [2016] indica o método do *bad smell*. O estudo de Chen & Jiang [2017] é o único que apresenta um oráculo cujos dados são exibidos em um arquivo de texto, usando caracteres especiais para separar as classes com *bad smell*.

[10]http://www.ptidej.net/downloads/experiments/qsic09/
[11]http://dx.doi.org/10.6084/m9.figshare.1590962
[12]http://sofa.uqam.ca/paprika/mobilesoft16.phpCodeSmells
[13]http://www.ptidej.net/downloads/experiments/prop-WCRE09
[14]http://www.rcost.unisannio.it/mdipenta/papers/ase2013/
[15]https://dibt.unimol.it/staff/fpalomba/reports/maltesque/
[16]http://soft.vub.ac.be/landfill/
[17]https://dibt.unimol.it/staff/fpalomba/reports/badSmell-analysis/index.html
[18]http://nemo9cby.github.io/icse2017.html
[19]http://www.ptidej.net/download/experiments/ase12/
[20]http://www.ptidej.net/research/decor/
[21]http://www.iro.umontreal.ca/ sahraouh/papers/ASE2010/

Tabela 3.4. Lista de artigos que propõem oráculos e estão disponíveis online

Referências	Programas	*Bad Smells*	Abordagens	Link
2009b	GanttProject v1.10.2 e Xerces v2.7.0	Blob (God Class)	Manual	[10]
2016	Apache Ant 1.8.0, aTunes 2.0.0, Eclipse Core 3.6.1, Apache Hive 0.9, Apache Ivy 2.1.0, Apache Lucene 3.6, JVLT 1.3.2, Apache Pig 0.8, Apache Qpid 0.18, Apache Xerces 2.3.0	Long Method, Feature Envy, Blob, Promiscuous Package e Misplaced Class	Ferramenta	[11]
2016	SoundWaves Podcast 0.112, Terminal Emulator for Android 1.0.70	Internal Getter/Setter (IGS), Member Ignoring Method (MIM) e HashMap Usage (HMU)	Ferramenta	[12]
2009a	Azureus e Eclipse	AbstractClassHasChildren, LargeClass, LargeClassOnly, LongMethod, LongParameterListClass, LowCohesionOnly, ManyAttributes, MessageChainsClass, MethodNoParameter, MultipleInterface, NoInheritance, NoPolymorphism, NotAbstract, NotComplex, OneChildClass, ParentClassProvidesProtected, RareOverriding, TwoInheritance	Ferramenta	[13]
2013b	Apache Ant e Tomcat, jEdit e Android API [framework-opt-telephony, frameworks-base, frameworks-support, sdk, tool-base]	Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob e Feature Envy	Manual	[14]
2017b	ArgoUML, Ant, aTunes, Cassandra, Derby, Eclipse Core, Elastic Search, FreeMind, hadoop, HSQLDB, Hbase, Hibernate, Hive, Incubating, Ivy, Lucene, JEdit, JFreeChart, JBoss, JVlt, jSL, Karaf, Nutch, Pig, Qpid, Sax, Struts, Wicket, Xerces	Class Data Should Be Private (CDSBP), Complex Class, Feature Envy, God Class, Inappropriate Intimacy, Lazy Class, Long Method, Long Parameter List, Message Chain, Middle Man, Refused Bequest, Spahetti code, Speculative Generality	Mista	[15]

Tabela 3.5. Continuação da lista de artigos que propõem oráculos e estão disponíveis online

Referências	Programas	*Bad Smells*	Abordagens	Link
2015	Apache Ant, Apache Tomcat, jEdit, Android API [framework-opt-telephony, frameworks-base, frameworks-support, sdk, tool-base], Apache[Commons Lang, Cassandra, Commons Codec, Derby], Eclipse Core, Apache James Mime4j, Google Guava, Aardvark, And engine, Apache Commons IO, Apache Commons Logging, Mongo DB	Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob e Feature Envy	Manual	[16]
2018	ArgoUML, Ant, aTunes, Cassandra, Derby, Eclipse Core, Elastic Search, FreeMind, Hadoop, HSQLDB, Hbase, Hibernate, Hive, Incubating, Ivy, Lucene, JEdit, JHotDraw, JFreeChart, JBoss, JVlt, jSL, Karaf, Nutch, Pig, Qpid, Sax, Struts, Wicket, Xerces	Class Data Should Be Private (CDSBP), Complex Class, Feature Envy, God Class, Inappropriate Intimacy, Lazy Class, Long Method, LPL, Message Chain, Middle Man, Refused Bequest, Spaghetti Code, Speculative Generality	Mista	[17]
2017	ActiveMQ, Hadoop e Maven	nullable objects, explicit cast, wrong verbosity level, Duplication with a method's definition, duplication with a local variable's definition, Malformed Output	Ferramenta	[18]
2012	ArgoUML, Azureus e Xerces	Blob, Functional Decomposition, Spaghetti Code e Swiss Army Knife	Ferramenta	[19]
2010	Xerces	Blob, Functional Decomposition, Spaghetti Code e Swiss Army Knife	Manual	[20]
2010	GanttProject v1.10.2, Xerces v2.7.0, e JHotdraw v7.1	Spaghetti Code, Blob, Functional Decomposition	Ferramenta	[21]

QP2: Em quais linguagens de programação os sistemas que compõem os oráculos são implementados?

Os oráculos de *bad smells* geralmente são baseados em sistemas de software desenvolvidos em uma linguagem de programação específica. Com esta questão de pesquisa, objetivamos identificar as linguagens de programação utilizadas para desenvolver os sistemas dos oráculos propostos na literatura. A Figura 3.5 mostra o número de oráculos por linguagem de programação. Observe que Java é a linguagem de programação mais usada, incluindo sistemas de software baseados em Java Android, Java Web ou Java Ahead. Dois oráculos usam Java e C, resultando em 45 oráculos para Java.

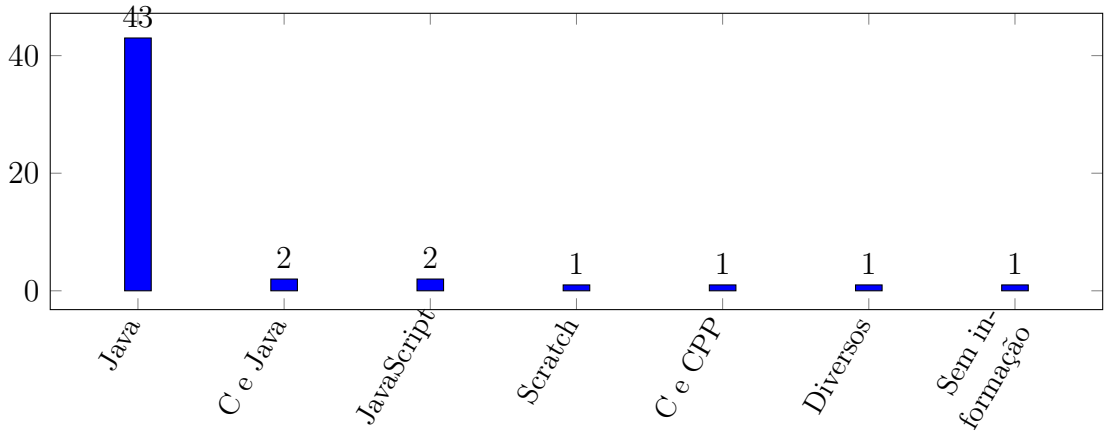


Figura 3.5. Linguagens de programação dos sistemas de software que formam os oráculos.

Observamos que as definições tradicionais de *bad smells* estão associadas a linguagens de programação orientadas por objetos. No entanto, alguns estudos usam linguagens de outros paradigmas (Hermans & Aivaloglou [2016], Fard & Mesbah [2013], Fenske et al. [2015]), mas geralmente a maioria deles faz adaptações nas definições dos *bad smells*.

Existem apenas três oráculos para C, dois para JavaScript e um para o Scratch. Um artigo menciona um oráculo, mas não indica em qual linguagem de programação ele se baseia. Em outro artigo (Macia et al. [2012]), o oráculo é construído com quatro linguagens: C ++, Java / AspectJ, C #.

QP3: Qual é o tamanho dos sistemas que compõem os oráculos?

A maioria dos programas usados nos oráculos é de grande porte. Mannan et al. [2016], por exemplo, usa um sistema de software com mais de 16 milhões de linhas de

Tabela 3.6. Número de ocorrências dos programas mais populares em oráculos

Programa	**Citação**	**#classes**
Apache Xerces	15	736
Eclipse	8	1181-17167
Apache Ant	7	846
GanttProject	7	188-245
JFreeChart	7	86-775
ArgoUML	6	777-1415
JHotDraw	6	159-679
Apache Lucene	6	1762-2246
Apache Nutch	6	183-259
Apache Cassandra	5	305-826
jEdit	5	228-520

Tabela 3.7. Lista de *bad smells* e seus termos alternativos

Bad Smell	**Termos Alternativos**
Duplicated Code	*Code Clone*, *Clone Class*, *Clone Code*, *Cloned Code*, *Code Duplication*, *Duplicated Prerequisites*
Large Class	*Blob*, *Big Class*, *Complex Class*, *God Class*
Long Method	*Brain Method*, *God Method*

código. Por outro lado, existem pequenos sistemas usados em alguns oráculos, como o oráculo baseado em `Scratch` (Hermans & Aivaloglou [2016]).

Alguns oráculos usam os mesmos programas. A Tabela 3.6 mostra os programas mais citados em ordem e tamanho decrescente, considerando o número de classes. Observamos que os projetos da `Apache Software Foundation` são amplamente utilizados para gerar oráculos de *bad smell*.

QP4: Quais bad smells são considerados pelos oráculos?

A maioria dos estudos propõem oráculos para os *bad smells* definidos por Fowler et al. [1999] (Khomh et al. [2009b], Palomba et al. [2016], Palomba et al. [2013b], Palomba et al. [2017b]).

Em alguns casos, há mais de uma definição para o mesmo *bad smell*; por exemplo, *Code Clone* ou *Code Duplicated* referem-se a *Duplicated Code*. Para superar esse problema, consideramos os *bad smells* e seus termos alternativos, conforme representado na Tabela 3.7.

Um total de 126 termos para *bad smells* apareceram. A Tabela 3.8 exibe os *bad smells* mais considerados pelos oráculos. *Blob* aparece em mais de 85 % dos trabalhos, enquanto *Long Method* aparece em 46 % dos oráculos. Observe que os dois *bad smells* aparecem na Tabela 3.7. Portanto, não necessariamente, esses são os termos usados nos artigos. A Tabela 3.4 reporta todos os *bad smells* considerados pelos oráculos que

Tabela 3.8. Lista dos *bad smells* mais populares

Bad Smell	**Número de artigos**
Blob	45
Long Method	26
Feature Envy	22
Data Class	17
Long Parameter List	15
ShotGun Surgery	15
Spaghetti Code	11
Duplicated Code	10
Functional Decomposition	10

estão disponíveis online.

QP5: Quais abordagens foram usadas para criar os oráculos?

Identificamos três abordagens usadas para criar os oráculos: manual, por ferramenta e mista. A abordagem manual consiste em inspecionar cada classe no sistema. A abordagem por ferramenta consiste no uso de algumas ferramentas para determinar onde estão os *bad smells* em um sistema de software. A abordagem mista é uma combinação das abordagens manual e por ferramenta. Por exemplo, algumas classes são desconsideradas na análise manual quando os dados anteriores reunidos pelas ferramentas de métricas de software indicam que as classes são bem construídas. A Figura 3.6 mostra como os oráculos são distribuídos entre essas abordagens. A abordagem mais comum para gerar oráculos é usar uma ferramenta para detectar o *bad smell*. Ferramentas foram usadas para criar 33 oráculos.

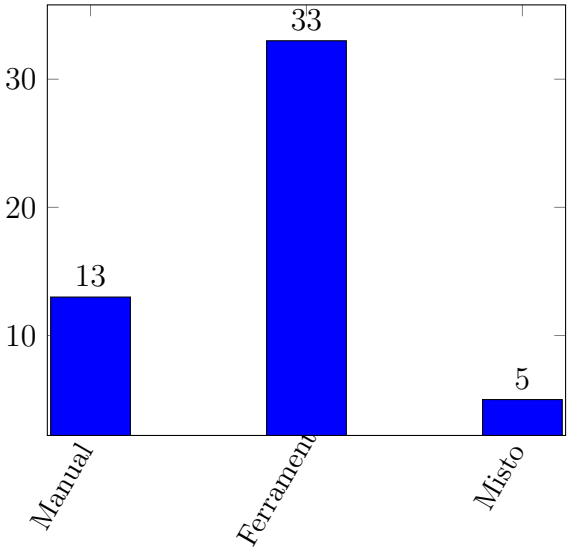


Figura 3.6. Abordagens usadas para criar oráculos

Entre as abordagens que usam ferramentas, muitas aplicaram a mesma ferramenta. A ferramenta mais comum é o `JDeodorant` (Fokaefs et al. [2011]), usado em quatro dos oráculos. As segundas ferramentas mais utilizadas são `inFusion` (inFusion Hydrogen [2012]) e `PMD` (PMD [2012]), utilizadas em três estudos. O `CodePro Analytix` (Clayberg & Rubel [2008]) e o `iPlasma` (Marinescu et al. [2005]) são usados em dois trabalhos. Nos trabalhos analisados neste estudo mais de 40 ferramentas são citadas explicitamente.

3.3 Discussão

Oráculos podem ser usados para avaliar estratégias e ferramentas para detecção de *bad smells*. A linguagem de programação mais considerada em oráculos disponíveis na literatura é Java, que aparece em 45 artigos. Portanto, as estratégias baseadas em Java têm mais opções de oráculos para avaliarem sua eficiência.

No entanto, a maioria desses oráculos foram definidos em projetos Java de código aberto, restringindo seus usos em estudos que consideram outros contextos. Ademais, apenas alguns oráculos estão disponíveis *online*, encontramos somente 12. A indisponibilidade de oráculos *online* têm dois impactos principais: retrabalho, uma vez que os pesquisadores produzem oráculos para aplicar em seus trabalhos e não replicabilidade dos estudos. Uma possível razão para os pesquisadores produzirem seus oráculos é que eles não conhecem oráculos propostos anteriormente na literatura. Acreditamos que os resultados deste estudo possam servir de referência nesse contexto.

Oráculos de *bad smell* em software devem ser considerados um *ground truth* para aqueles que os utilizam. Entretanto, oráculos produzidos por ferramentas não atendem essa prerrogativa, uma vez que os resultados passam pelo filtro da técnica de detecção implementada. O uso de um oráculo produzido por ferramenta para validar os resultados de outra ferramenta apenas serve para garantir que são iguais. A abordagem mista surge com o intuito de garantir que os resultados obtidos pela ferramenta são verdadeiros, porém a análise do especialista somente após a seleção prévia das classes pela ferramenta coloca em dúvida a existência de *bad smells* no restante de classes do software. Se comparado às demais abordagens, os oráculos obtidos pela inspeção manual ganha maior relevância uma vez que cada software é analisado em sua totalidade pelo crivo do especialista.

Os oráculos disponíveis *online* cobrem um grande número de *bad smells* e usam muitos sistemas de software. No entanto, observamos que a terminologia para *bad smells* nesses oráculos varia. A maioria dos oráculos se baseia em resultados das fer-

ramentas. Esse fato é paradoxal, pois as ferramentas precisam de um oráculo para validar seus resultados.

Encontramos quatro oráculos baseados na abordagem manual e dois oráculos gerados por uma abordagem mista. A principal ameaça de aplicar esses tipos de oráculo é a subjetividade da avaliação humana.

3.4 Considerações Finais

Este capítulo apresentou uma Revisão Sistemática da Literatura para identificar oráculos de *bad smells*. A principal motivação para este estudo foi o fato de *bad smells* terem sido amplamente estudados, contudo grande parte desses estudos se basearam em oráculos de *bad smells* obtidos por ferramentas que ainda não foram adequadamente comprovadas como precisas. As pesquisas que realizamos nos repositórios digitais retornaram 4.948 estudos primários. No entanto, um pequeno número desses estudos gerou algum tipo de oráculo, ou seja, 51. Descobrimos que 12 oráculos de *bad smells* estão disponíveis em algum repositório *online*. Os pesquisadores podem se beneficiar dos resultados deste capítulo para produzir novos estudos, cruzar dados e avaliar suas propostas. Os *bad smells* definidos por Fowler et al. [1999] são os mais usados nos oráculos.

Os resultados deste capítulo evidenciam uma lacuna a ser preenchida na literatura, por exemplo:

1. apenas alguns oráculos identificam os métodos onde o *bad smell* está localizado;
2. a maioria dos oráculos é de fato gerado utilizando ferramentas, o que é uma ameaça relevante à sua validade.

Com o objetivo de contribuir para contornar esses problemas, neste trabalho foi criado um oráculo baseado em inspeção manual de software. Os capítulos seguintes descrevem o processo de geração desse oráculo e os seus resultados.

Capítulo 4

A Ferramenta `InspectBSmell`

O principal objetivo deste trabalho de dissertação é criar e disponibilizar um oráculo confiável e verificado de *bad smells*. Para atingir este objetivo foi projetada e implementada uma ferramenta denominada `InspectBSmell` para a identificação manual de instâncias de *bad smells*. A ferramenta criada é um *plug-in* para a IDE Eclipse que provê funcionalidades para cadastrar novos *bad smells* e inserir comentários para cada uma das instâncias identificadas. `InspectBSmell` foi especificada e projetada pelo autor deste trabalho e implementada com sua participação no âmbito de um trabalho de iniciação científica. Este capítulo apresenta a arquitetura de `InspectBSmell`, a metodologia usada para seu desenvolvimento, incluindo decisões de projeto, as funcionalidades oferecidas e a sua implementação.

Organizamos o capítulo da seguinte forma. Seção 4.1 apresenta a descrição das funcionalidades da ferramenta. Seção 4.2 explica as decisões de implementação da ferramenta, quais foram as ferramentas externas utilizadas e sua arquitetura. Seção 4.3 explica como fazer a instalação da ferramenta e como utilizar cada uma de suas funcionalides. Por fim, a Seção 4.4 faz as considerações finais a respeito da ferramenta.

4.1 Descrição de `InspectBSmell`

A inspeção manual de software na identificação de *bad smells* é um processo longo e sujeito a falhas humanas. Por se tratar de um processo longo, é importante a existência de um recurso que permita que a inspeção possa ser dividida em períodos de tempo não sequenciais. Para isso, é necessário armazenar, de forma incremental, os resultados parciais da inspeção, com dados tais como quais classes já foram inspecionadas e quais *bad smells* foram encontrados nelas. A ferramenta `InspectBSmell` visa apoiar a

inspeção manual de *bad smells* em software Java realizada dessa forma.

A ferramenta `InspectBSmell` consiste em um *plug-in* para o Eclipse. Suas funcionalidades foram pensadas para automatizar alguns dos passos tomados pelo especialista no momento de fazer a inspeção manual de sistemas de software. As funcionalidade são as seguintes:

1. Identificar uma ocorrência de *bad smell* destacando o trecho de código de evidência.

2. Comentar os motivos para a identificação de uma instância de *bad smell*.

3. Assinalar a conclusão da leitura de uma classe.

4. Exportar os dados do oráculo para formatos de planilha e JSON (*JavaScript Object Notation*).

5. Verificar quais foram as instâncias de *bad smell* encontradas, bem como o progresso da análise do software.

4.2 Implementação de `InspectBSmell`

Definidas as funcionalidades de `InspectBSmell`, o próximo passo foi trabalhar na sua estrutura. A primeira etapa para o desenvolvimento de `InspectBSmell` foi a construção do *plug-in*.

O *plug-in* foi construído em Java, na IDE Eclipse, utilizando-se do ambiente de desenvolvimento de *plug-in* PDE - *Plug-in Development Environment*), cujo uso provê ferramentas para criar e desenvolver *plug-ins* para o Eclipse. O PDE disponibiliza:

1. **PDE UI**: modelos e editores para facilitar o desenvolvimento de *plug-in* para o Eclipse.

2. **PDE API Tools**: ferramentas para manutenção de APIs integradas à IDE.

3. **PDE Build**: *scripts* e ferramentas para facilitar a automação do processo de *build*.

4. **PDE Incubator**: ambiente para desenvolvimento de novas ferramentas que escapam do escopo do SDK do Eclipse.

4.2.1 Ferramentas Externas

Uma vez definido o ambiente de desenvolvimento do *plug-in*, fez-se necessário incluir ao projeto ferramentas externas para manipular e coletar os dados de um projeto aberto no Eclipse. São elas:

Java do Eclipse (JDT - *Java development tools*). Consiste em um projeto que fornece suporte ao desenvolvimento de *plug-in*s para o Eclipse provendo APIs que podem ser estendidas (Foundation [2020b]). A importação da ferramenta ao projeto inclui uma perspectiva Java no Eclipse Workbench e várias visualizações, editores, assitentes, construtores e ferramentas de mesclagem ou refatoração de código.

JFace. Um conjunto de ferramentas de interface gráfica para as tarefas mais comuns na programação de interfaces gráficas para o usuário (Foundation [2010]). O JFace é um sistema de janelas independentes que inclui componentes de imagem e registro de fontes, texto, caixas de diálogo, preferências e retornos de progresso para operações longas.

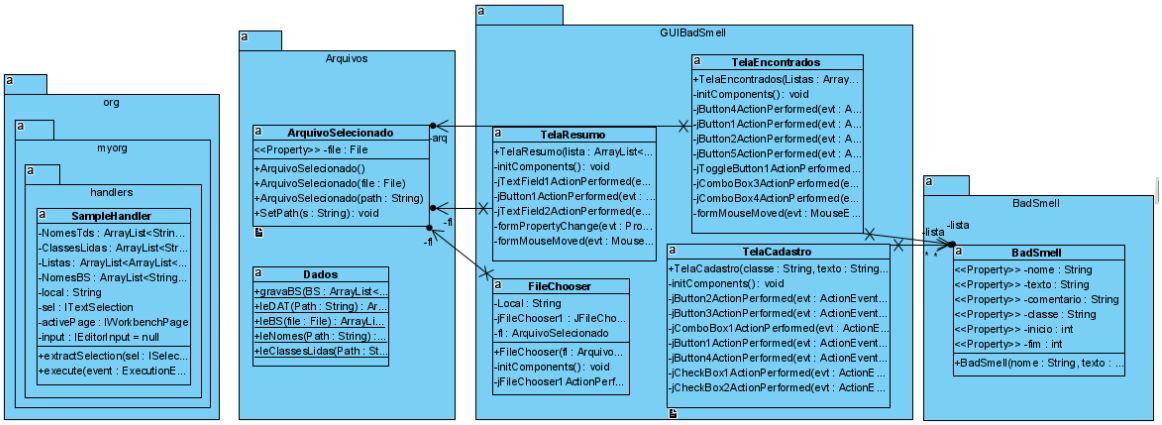
org.eclipse.core. Provê funcionalidades do núcleo do Eclipse, a saber: `commands, resources` e `runtime`. `Commands` dispõe definições de trechos abstratos de funcionalidades (Foundation [2020a]). `Resources` permite gerenciar arquivos, projetos, pastas e demais recursos de um projeto. `Runtime` libera métodos úteis para os processos em tempo de execução da plataforma.

Java Swing. Disponibiliza elementos gráficos para serem usados na plataforma e foram utilizadas para criar as interfaces do *plug-in* (Technology [2010]).

4.2.2 Arquitetura de `InspectBSmell`

O projeto para o *plug-in* faz manutenção de dados, para o registro de *bad smells* e suas instâncias, cria interfaces gráficas e manuseia dados provenientes da IDE Eclipse. Para tanto, as classes da ferramenta `InspectBSmell` foram divididas entre quatro pacotes: `data`, `badsmell`, `GUIbadsmells` e `JFace`. A arquitetura da ferramenta pode ser visualizada no diagrama de classes exposto na Figura 4.1.

Figura 4.1. Arquitetura da Ferramenta



O pacote `data` é o responsável por fazer o gerenciamento do registro de dados da ferramenta. O armazenamento dos dados é feito por meio de arquivos do formato `dat`. Para o registro dos dados colhidos durante a inspeção do código utiliza-se o arquivo `read.dat`. Nesse arquivo é contida uma lista de objetos do tipo *BadSmell*. As classes que foram registradas como lidas, ou seja, que tiveram a sua inspeção finalizada pelo usuário, ficam registradas no arquivo `ClassesLidas.dat`. O pacote contém uma única classe com métodos que retornam os dados armazenados em cada um desses arquivos e outro método que gera uma planilha no formato `xls` com os dados do oráculo.

As entidades que definem um *bad smell* e suas instâncias à nível de projeto ficam localizados no pacote `badsmell`. Na implementação, a classe `BadSmell` necessita apenas de um nome, e a classe `BSInstancia` trata da ocorrência de um *bad smell* dentro de um projeto. A última possui como atributos um texto, um comentário, o nome da classe, a linha de início e a linha de fim da instância do *bad smell*. O relacionamento `1 x N` entre as classes é necessário, pois um tipo de *bad smell* pode ter uma coleção de ocorrências. Portanto, a inserção e a exclusão de uma instância *bad smell* estão implementadas na classe `BadSmell`.

As interfaces que dão corpo ao *plug-in* estão localizadas no pacote `GUIBadSmell`. Ao todo são três interfaces: interface de cadastro, interface de resumo e a interface que exibe as instâncias encontradas. Cada classe implementa um `JFrame` e recebe como parâmetro a lista de objetos para fazer sua exibição na tela.

O último pacote `handlers` possui apenas uma classe, a classe `SampleHandler`. Essa classe faz as tratativas quando o usuário do Eclipse aciona um atalho de teclado ou de barra de ferramentas para a abertura das janelas do *plug-in*.

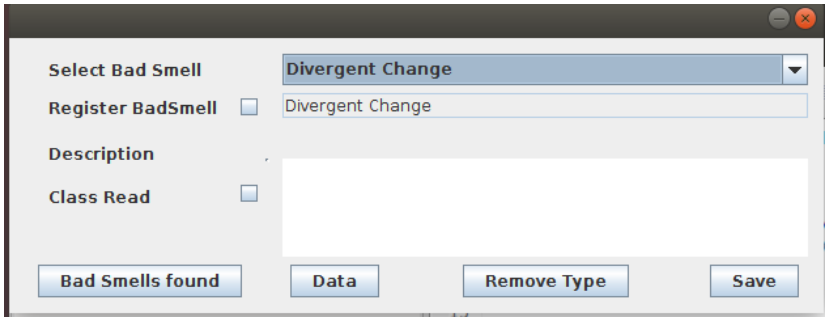
4.3 Instalação e Uso

A ferramenta `InspectBSmell` está disponível online para *download* e livre instalação para os usuários.[1] O processo de instalação é feita de acordo com os seguintes passos:

1. Tenha a versão do `Eclipse IDE` for Java Developers instalada em sua máquina.
2. Baixe o conteúdo do *plug-in* e extraia seus arquivos.
3. No *Eclipse*, vá ao menu Help and Install New software.
4. No *popup*, clique em *Add...* e clique no botão Local.
5. No gerenciador de arquivos, selecione a pasta dos arquivos baixados no Item 3.
6. Selecione a pasta *update site* e clique *Add*.
7. Clique em *Select All* e clique em *Next* até chegar na última etapa e selecione *Finish*.
8. Após finalizado, aparecerá uma mensagem de segurança. Selecione a opção *Install anyway*.

Após a instalação, a IDE precisa ser reinicializada. No menu de acesso da IDE, é possível encontrar a opção *Bad Smells* que ao ser clicada abre a opção *Add Bad Smell* $\cdots$. Nesse momento, o acesso à janela principal da ferramenta `InspectBSmell` é liberado, conforme exibe a Figura 4.2.

Figura 4.2. Janela principal de `InspectBSmell`



[1] https://drive.google.com/file/d/1OjwNFtaSC66ZPAjdH3mSwGM2awih9ABS/view

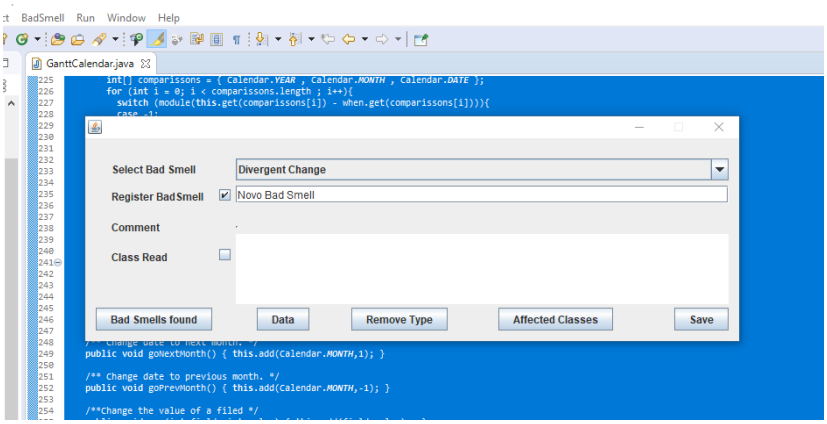
4.3.1 Funcionalidades de `InspectBSmell`

O *plug-in* permite que o usuário assinale uma ocorrência de *bad smell* no código, marque uma classe como inspecionada, cadastre um novo tipo de *bad smell* e faça um acompanhamento do progresso da inspeção manual de um software. Para facilitar o acesso à página principal da Figura 4.2 é possível usar o atalho `Ctrl + 6` no Eclipse. A descrição dos passos para executar cada uma das funcionalidades considera esse comando em vez de solicitar ao usuário que clique no item *Bad Smell* do menu e na opção *Add Bad Smell...*.

4.3.1.1 Cadastro e Exclusão de um Tipo de *Bad Smell*

`InspectBSmell` possui cadastrados os *bad smells* clássicos definidos por Fowler et al. [1999], porém caso o usuário necessite inserir um novo tipo de *bad smells* é possível fazê-lo na interface exibida pela Figura 4.3. Para tanto, basta somente marcar a opção `Register Bad Smell` e o campo a direita ficará habilitado para edição. Uma vez registrada a ocorrência do *bad smells* o novo tipo ficará disponível para a escolha em `Select Bad Smell`.

Figura 4.3. Manutenção dos Tipos de *Bad Smell*



O processo para a exclusão de um tipo de *bad smells* também é disponibilizado nessa mesma janela da Figura 4.3. Para tanto é preciso que o usuário selecione o *bad smells* no primeiro item do formulário e clique em `Remove Type`.

4.3.1.2 Cadastro de uma Instância de *Bad Smell* .

No momento em que o usuário estiver fazendo a inspeção manual do sistema, se ele identificar a ocorrência de um *bad smells*, é necessário seguir os passos a seguir para assinalar a instância desse *bad smells* no código.

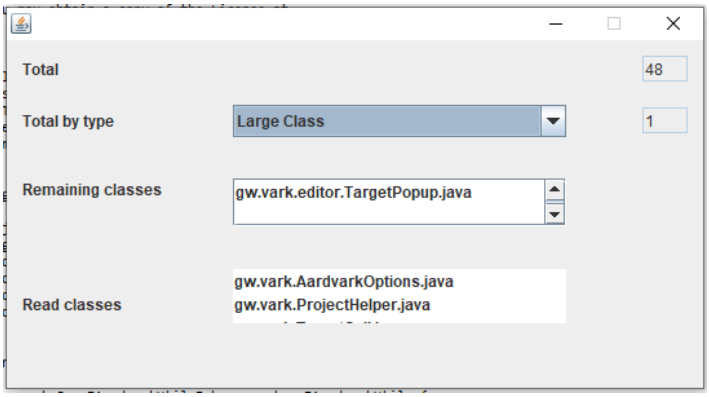
1. Selecione o trecho de código que evidencia o *bad smells* e execute o comando `Ctrl + 6`
2. Selecione o *bad smells*, coloque um comentário e clique em `Save`.

Quando for o momento de assinalar que a classe já foi lida e que não é mais necessária a sua leitura, basta marcar a opção `Class Read` e clicar em `Save`.

4.3.1.3 Acompanhar o Progresso da Inspeção Manual de um Software

Para verificar quantas classes foram inspecionadas e quantas ainda necessitam ser lidas, é necessário abrir a tela principal do *plug-in*, (veja Figura 4.2), e depois clicar em `Data`. Feito isso, a tela da Figura 4.4 abrirá permitindo verificar em `Remaining Classes` quais são as classes que faltam ser inspecionadas. Em `Read classes` aparece a lista de classes que já foram assinaladas como lidas. Por último, o campo numérico `Total` indica a quantidade de instâncias de *bad smells* encontrados pelo oráculo e `Total by type` a quantidade filtrada pelo tipo de *bad smells* selecionado.

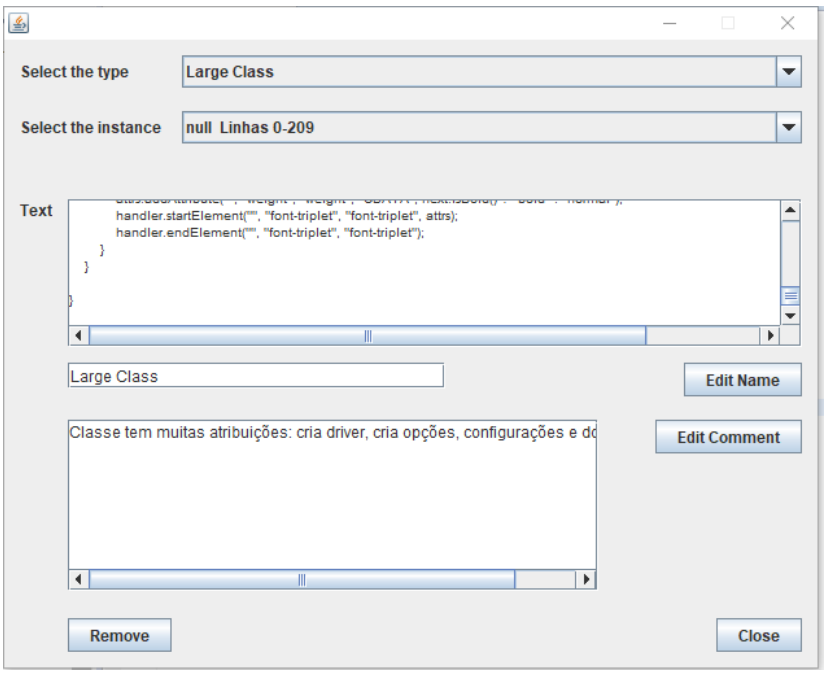
Figura 4.4. Tela de progresso da inspeção



4.3.1.4 Verificar as Instâncias de *Bad Smell*

Caso o usuário necessite ter uma visão mais detalhada de uma instância de *bad smells* registrada no *plug-in*, é preciso acionar o comando `Ctrl + 6` e selecionar o botão `Bad Smells found`. A interface da Figura 4.5 abrirá e será possível encontrar cada uma das instâncias do tipo de *bad smells* selecionado no primeiro campo da tela. A lista de instâncias será disponibilizada no segundo campo da tela e os dados da instância aparecerão nos demais campos. A edição do texto, do nome do tipo de *bad smells* e do comentário estão disponíveis nas caixas de textos habilitados do formulário. Por último, se desejar descartar a instância do *bad smells* selecionado é necessário somente clicar no botão `Remove`.

Figura 4.5. Detalhe de uma instância de bad smell



4.3.1.5 Exportação

Os dados do oráculo criado podem ser visualizados por meio de um conjunto de arquivos que são salvos automaticamente no diretório raiz do projeto. O primeiro arquivo traz os dados de *bad smells* sumarizados indicando quantas instâncias de *bad smells* foram encontrados para cada um dos tipos de *bad smell*, conforme Figura 4.6. Além da planilha, são criadas outras exportações com arquivos nos formatos `csv` e `json` indicando em cada linha uma instância de *bad smell* e a classe em que está localizada.

Figura 4.6. Detalhe de uma instância de *bad smell*

	A	B	C	D	E	F
1	Nome	Classes	Quantidade	Numero de Classes		
2	Divergent Change		0	0		
3	Large Class	org.apache.commons.codec.binary.	13	13		
4	Long Method	org.apache.commons.codec.binary.	15	14		
5	Shotgun Surgery		0	0		
6	Long Parameter Lis		0	0		
7	Data Class	org.apache.commons.codec.CharEn	4	4		
21	Comments		0	0	Total de classes: 85	Total Smells: 32
22						

4.4 Considerações Finais

Este capítulo descreveu a ferramenta `InspectBSmell`, criada neste trabalho para auxiliar a identificação manual de *bad smells* em software Java. O *plug-in* foi desenvolvido em linguagem Java para o ambiente Eclipse e seu uso proporciona ao usuário todas as funcionalidades necessárias na inspeção manual de um software.

O *plug-in* foi utilizado no processo de geração do oráculo de *bad smells* criado neste trabalho. A ferramenta foi usada para a análise de cinco sitemas de software Java por quatro avaliadores. Cada um desses avaliadores utilizou-se do *plug-in* para registrar as ocorrências de *bad smells* identificados nos sistemas de software. A geração do oráculo é descrita no Capítulo 5.

Capítulo 5

Um Oráculo de *Bad Smells*

Este capítulo apresenta o oráculo de *bad smells* criado neste trabalho e cujo objetivo principal é contribuir com a literatura provendo um repositório de *bad smells* validado manualmente. Seção 5.1 especifica a metodologia utilizada na construção do oráculo e detalha as etapas que abrangeram todo o desenvolvimento do projeto. Seção 5.2 apresenta os resultados obtidos. Seção 5.3 apresenta as considerações finais.

5.1 Metodologia

O desenvolvimento deste trabalho de pesquisa compreendeu cinco etapas.

Etapa 1. Esta etapa consistiu na seleção dos *bad smells*.

Etapa 2. Na segunda etapa, os sistemas de software a serem analisados foram selecionados.

Etapa 3. Nesta etapa, os códigos-fonte dos sistemas de software selecionados foram inspecionados para identificação de *bad smells* neles, utilizando-se de `InspectBSmell`, a ferramenta descrita no Capítulo 4.

Etapa 4. Nesta etapa, o oráculo foi consolidação.

Etapa 5. Nesta etapa, ocorreu a disponibilização dos resultados no `Landfill`.

A seguir são descritas cada uma dessas etapas.

5.1.1 Etapa 1 - Seleção dos *Bad Smells*

Para selecionar os *bad smells* considerados neste trabalho dentre os 36 principais descritos na literatura [Fowler et al., 1999; Brown et al., 1998; Fowler, 2018], levamos em consideração aqueles que figuram no topo da lista dos mais populares entre os que

possuem trabalhos identificados no Capítulo 3. São eles: *Blob* ou *Large Class*, *Long Method*, *Feature Envy* e *Data Class*. Além desses *bad smells*, foi incluído nesta pesquisa o *Refused Bequest* porque no Capítulo 3 há dois trabalhos que o considera.

Sabe-se que os *bad smells* variam de escopo, alguns afetam uma classe por inteira e outros afetam apenas métodos. O *plug-in* desenvolvido neste trabalho de pesquisa permite que seja assinalado a ocorrência de um *bad smell* desde o nível mais interno, métodos, até o nível mais externo possível, classe. A Tabela 5.1 exibe os *bad smells* considerados nesse trabalho, identificando seu nível de ocorrência.

Tabela 5.1. *Bad Smells* escolhidos para a inspeção

Bad Smells	**Nível da ocorrência**
Blob ou *Large Class*	Classe
Long Method	Método
Feature Envy	Método
Data Class	Classe
Refused Bequest	Método

5.1.2 Etapa 2 - Escolha dos *Sistemas de Software*

Os critérios enumerados a seguir foram aplicados para a escolha dos sistemas de software analisados para a composição do oráculo:

1. Possuir código aberto.
2. Estar disponível para *download*.
3. Ser grande o suficiente para ter alguma relevância, mas não tão grande que impeça a inspeção manual.
4. Ser relevante para a comunidade, ou seja, ter sido alvo de pesquisa em algum outro estudo da área.

A SLR realizada e documentada no Capítulo 3 nos forneceu os insumos necessários para encontrar a base dos sistemas de software que seriam alvo da inspeção manual desenvolvida no presente capítulo. Ao todo, foram utilizados cinco sistemas de software abertos, disponíveis no repositório `GitHub`.[1] A Tabela 5.2 exibe o nome dos software e os seu endereço de `URL`. A seguir, uma breve descrição de cada um deles é apresentada.

`Aardvark`: é uma ferramenta para a construção de projetos. É baseada no `Gosu` e no `Apache Ant` e sua finalidade é prover um ambiente de criação de *scripts* enquanto mantém todas as funcionalidades que as tarefas `Ant` disponibiliza.

[1] https://github.com/

Tabela 5.2. URL do repositório de cada um dos sistemas de software

Software	URL
Aardvark	https://github.com/vark/Aardvark.git
And engine	https://github.com/nicolasgramlich/AndEngine.git
Apache Commons Codec	https://github.com/apache/commons-codec.git
Apache Commons Logging	https://github.com/apache/commons-logging.git
GanttProject	https://github.com/sootparser/ganttproject-1.10.2.git

`AndEngine`: é uma ferramenta de jogo "2D". Ela permite que desenvolvedores de jogos, experientes ou inexperientes, desenvolvam jogos para a plataforma `Android`. A documentação da ferramenta afirma que ele disponibiliza funcionalidades suficientes para criar qualquer jogo que seja de duas dimensões.

`Apache Commons Codec`: é um software que provê implementações comuns de codificação e decodificação para formatos comuns no desenvolvimento de software, tais como `Base64, Hex, Phonetic` e `URLs`.

`Apache Commons Logging`: é um orquestrador entre diferentes implementações de `log`. Uma biblioteca que usa o `commons-logging API` pode ser utilizada com qualquer implementação de `log` em tempo de execução.

`GanttProject`: é um software Java desenvolvido na *University of Marne-la-Vallée* (França). Esse software permite dividir um projeto em subtarefas, em que cada tarefa possui uma data de início, uma duração, notas e dependências. O software provê uma interface gráfica para realizar a manutenção dos dados.

Palomba et al. [2015] utiliza dos quatro primeiros sistemas de software dispostos na Tabela 5.2 para compor o conjunto de oráculos manuais do projeto `Landfill`. O oráculo de Palomba et al. [2015] considera os *bad smells Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob* e *Feature Envy*. O oráculo proposto e criado neste trabalho de Mestrado analisa as incidências dos *bad smells Blob* ou *Large Class*, *Long Method*, *Feature Envy*, *Data Class* e *Refused Bequest*. O oráculo produzido pelo presente trabalho pode ser estendido com o resultado do projeto `Landfill` para as ocorrências de *Blob* e *Feature Envy* e poderá incorporar os resultados do restante dos *bad smells*.

A SLR descrita no Capítulo 3 identificou duas ocorrências de inspeções do software `GanttProject` apresentada na literatura. Khomh et al. [2009b] utilizou a mesma versão 1.10.2 do software `GanttProject` considerada no presente projeto para identificar as ocorrências de *Blob*. Por outro lado, o trabalho de Moha et al. [2010] realizou

o teste de seu algoritmo nessa mesma versão do software e validou o resultado com uma análise manual das classes suspeitas para instâncias de *Large Class*, *Functional Decomposition*, *Spaghetti Code* e *Swiss Army Knife*.

5.1.3 Etapa 3 - Inspeção Manual dos Sistemas de Software

A inspeção manual dos sistemas de software foi otimizada pelo uso da ferramenta `InspectBSmell` desenvolvida no presente trabalho. A tarefa consiste na inspeção exaustiva de cada uma das classes que fazem parte dos sistemas de software identificados na Tabela 5.2. Ao encontrar uma ocorrência de um dos *bad smells* da Tabela 5.1, o avaliador assinala o local onde é evidenciado o *bad smell* via `InspectSmells` e prossegue na inspeção.

Essa etapa do trabalho foi realizada com a colaboração de mais três especialistas além do autor do presente trabalho. O autor inspecionou todos os sistemas de software escolhidos para o oráculo. Cada software foi também inspecionado por um outro especialista. Ou seja, todos os sistemas de software foram inspecionados por dois especialistas, o autor dessa dissertação mais um colaborador. Todos os avaliadores são formados em curso superior na área de Computação, sendo que um deles é Mestre em Ciência da Computação, conhecem *bad smells* e possuem experiência de mercado, com mais de quatro anos em desenvolvimento de software. A formação e tempo de experiência como desenvolvedor de software no mercado são indicados na Tabela 5.3.

Tabela 5.3. Detalhes dos colaboradores

Colaborador	**Formação**	**Tempo de Mercado**
Colaborador 1	Graduado em Sistemas de Informação pela UFMG	4 anos
Colaborador 2	Graduado em Engenharia da Computação pelo Cefet-MG	6 anos
Colaborador 3	Graduado em Engenharia da Computação pelo Cefet-MG e Mestre em Ciências da Computação pela UFMG	5 anos

Para garantir o nivelamento dos colaboradores com o tema do estudo, cada um deles recebeu um arquivo contendo a descrição de cada *bad smell* de Fowler et al. [1999]. Com base nesse nivelamento, os avaliadores realizaram a avaliação de forma independente, tendo liberdade para utilizarem seus próprios critérios, de acordo com suas experiências, uma vez que as definições de *bad smells* são subjetivas e podem permitir mais de uma interpretação.

Também foi disponibilizado para cada autor a sequência de comandos do `GitHub` para fazer o *clone* do projeto pela sua respectiva `URL` (Tabela 5.2) e ir para o *snapshot* exibido na Tabela 5.4. O comando `git clone` seguido pela `URL` do projeto em algum repositório `GitHub` permite que o usuário copie para um diretório local os arquivos que fazem parte do corpo de um projeto.

Após o *clone*, o outro comando necessário para garantir que os projetos inspecionados sejam idênticos é o `checkout` que deve ser seguido pelo nome do *snapshot* desejado. A lista com seus nomes, os *snapshots* considerados no presente trabalho e tamanho pode ser visualizada na Tabela 5.4.

Por fim, foi disponibilizado, para cada avaliador, um arquivo contendo as instruções para o *download* e instalação do *plug-in*, adicionado dos passos para utilizá-lo. Cada avaliador gerou os arquivos `BadSmells.xls` e `read.dat`. Esses arquivos, fornecidos por `InspectBSmell`, contém os dados do oráculo gerado por cada um dos avaliadores.

Tabela 5.4. Sistemas de software que compõem o oráculo proposto

Software	***Snapshot***	**# de Classes**	**KLOC**
Aardvark	ff98d508	103	25
And engine	f25236e4	596	20
Apache Commons Codec	c6c8ae7a	103	23
Apache Commons Logging	d821ed3e	61	23
GanttProject	6d01a59	188	31

Ao longo da inspeção, algumas decisões, descritas a seguir, foram tomadas.

- Classes de testes foram consideradas no conjunto de arquivos a serem verificados, porém as interfaces do software não. A justificativa para descartar as interfaces é de que os *bad smells* no nível de método não podem ser encontrados em uma interface uma vez que não existe nenhuma implementação. Da mesma forma, os *bad smells* que incidem sobre uma classe não podem ser aplicados a uma interface sem que exista a definição de campos. Essa escolha não traz prejuízo aos resultados, pois uma classe que implementa uma interface com mais de uma responsabilidade, por exemplo, acabará sendo assinalada por *Large Class*.
- Raciocínio semelhante foi aplicado para as definições de enumeradores do sistema. Um *Enum* por definição poderia ser classificado como um *Data Class* por se tratar de uma estrutura que somente armazena valores, entretanto como a classificação foi feita somente com classes, esse tipo de estrutura foi desconsiderada da inspeção.

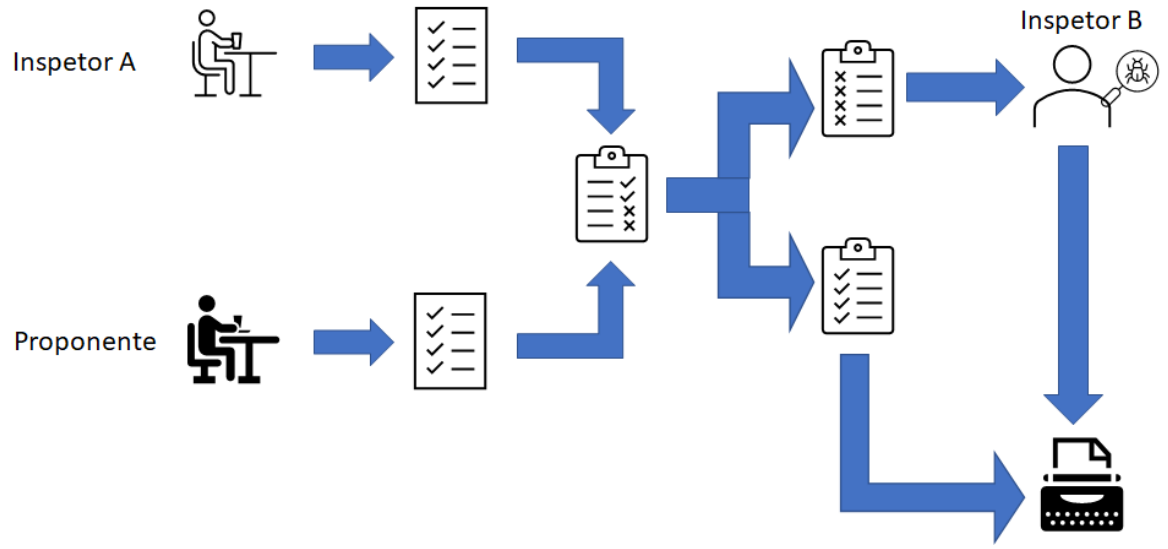
- Uma outra escolha trata das heranças do projeto. O *Refused Bequest* ocupa-se do comportamento das subclasses quanto aos recursos que estão sendo concedidos por uma herança, porém na inspeção foram avaliadas apenas as heranças das classes-mãe que também estavam no projeto e cujo código fonte era facilmente encontrado no momento da inspeção.

5.1.4 Etapa 4 - Consolidação do Oráculo

Uma vez finalizada a inspeção e obtido os arquivos do oráculo gerado por cada um dos colaboradores, os resultados da inspeção realizada por cada avaliador foram comparadas. Nos casos em que houve coincidência de avaliação, ou seja, quando ambos avaliadores indicaram uma ocorrência de um mesmo *bad smell* em um mesmo elemento, a ocorrência foi incluída no oráculo. Aquelas ocorrências em que houve divergência nos resultados das análises dos especialistas foram novamente analisadas na etapa que chamamos Consolidação.

Cada divergência foi analisada por um terceiro avaliador. Por exemplo, se um avaliador considerou que a Classe A é uma instância de *Large Class*, porém o outro avaliador não, um terceiro avaliador analisou a mesma classe para se obter um parecer final sobre ela. Todo o processo de geração do oráculo está representado pela Figura 5.1. Além do autor do texto, foram incluídos dois outros colaboradores, Inspetor A e Inspetor B. No diagrama, o Inspetor A também faz a inspeção manual do software e seus resultados são mesclados com os resultados do Proponente. Se os dados forem iguais, são inseridos no oráculo e caso ocorra divergência, passa pela an álise do Inspetor B onde será descartado ou inserido no oráculo final.

Figura 5.1. Procedimentos para a criação de um oráculo



A divisão das inspeções entre os sistemas de software e o responsável pela consolidação foi distribuída conforme exibido na Tabela 5.5. Considerando que o autor deste trabalho fez a inspeção de todos os sistemas de software, ele foi excluído da tarefa de consolidação.

Tabela 5.5. Divisão da inspeção dos sistemas de software

Software	Inspeção	Consolidação
Aardvark	Autor + Colaborador 1	Colaborador 2
And engine	Autor + Colaborador 3	Colaborador 2
Apache Commons Codec	Autor + Colaborador 2	Colaborador 1
Apache Commons Logging	Autor + Colaborador 1	Colaborador 2
GanttProject	Autor + Colaborador 2	Colaborador 1

5.1.5 Etapa 5 - Disponibilização dos Resultados no `Landfill`

O oráculo gerado foi exportado para o formato aceito pelo repositório `Landfill`, criado por Palomba et al. [2015], onde ficará disponível para a livre consulta de qualquer pessoa interessada. A ferramenta `InspectBSmell` fornece um arquivo `JSON` ao fim das inspeções, porém esse arquivo leva em conta apenas os resultados obtidos por uma inspeção. Dessa forma, foi necessário, com base no arquivo `read.dat`, mesclar os resultados em um mesmo arquivo e fazer a submissão no repositório *online*. O arquivo `JSON` segue a estrutura exposta no *Listing* 5.1.

Listing 5.1. Estrutura do `JSON` aceito pelo `Landfill`

```
{ "name":"Oracle",
   "description":"An oracle",
   "authors":["Rafael Prates","Other"],
   "systems":[
                {
                "name":"Aardvark",
                 "snapshot":"g34f2",
                 "smells": [
                      {"type":"Blob","body":"gw.vark.annotations.Depends"},
                      {"type":"ParallelInheritance","body":" gw.vark.annotations.Target"}
                          ]
                 }                    ]
}
```

5.2 Resultados

Esta seção apresenta os resultados do oráculo de *bad smell* segmentado por cada um dos sistemas de software analisados neste trabalho. Os dados apresentados também

discriminam os resultados obtidos por meio da inspeção individual de cada um dos colaboradores. É importante ressaltar que os dados são classificados por ocorrência, significando que cada um dos valores mostrados na presente seção indicam a existência de uma instância podendo ser ou não dentro de uma mesma classe para aqueles *bad smells* que estão no nível de um método, conforme exibido na Tabela 5.1(Seção 5.1.1). As Tabelas 5.6, 5.7, 5.8 , 5.9 e 5.10 mostram as quantidades de ocorrências de cada *bad smell* nos sistemas de software `Aardvark`, `And Engine`, `Apache Commons Codec`, `Apache Commons Logging` e `GanttProject`, respectivamente.

Tabela 5.6. Resultado da inspeção do `Aardvark`

Bad smell	**Autor**	**Colaborador 1**	**Consolidado**
Data Class	0	0	0
Feature Envy	0	0	0
Large Class	1	4	4
Long Method	5	3	6
Refused Bequest	1	1	1

Tabela 5.7. Resultado da inspeção do `And Engine`

Bad smell	**Autor**	**Colaborador 3**	**Consolidado**
Data Class	2	34	16
Feature Envy	0	24	23
Large Class	5	24	9
Long Method	5	54	24
Refused Bequest	20	8	13

Tabela 5.8. Resultado da inspeção do `Apache Commons Codec`

Bad smell	**Autor**	**Colaborador 2**	**Consolidado**
Data Class	2	4	2
Feature Envy	0	0	0
Large Class	0	13	6
Long Method	4	15	9
Refused Bequest	0	0	0

5.3 Considerações Finais

O presente capítulo detalhou cada um dos processos e escolhas que foram tomadas ao longo do desenvolvimento do oráculo de *bad smells*. O oráculo foi desenvolvido com base na análise de cinco sistemas de software Java por quatro avaliadores. Cada

Tabela 5.9. Resultado da inspeção do `Apache Commons Logging`

Bad smell	**Autor**	**Colaborador 1**	**Consolidado**
Data Class	1	0	0
Feature Envy	0	0	0
Large Class	2	0	1
Long Method	1	1	2
Refused Bequest	0	1	0

Tabela 5.10. Resultado da inspeção do `GanttProject`

Bad smell	**Autor**	**Colaborador 1**	**Consolidado**
Data Class	22	3	15
Feature Envy	18	0	13
Large Class	11	45	18
Long Method	43	37	48
Refused Bequest	0	0	0

avaliador utilizou o *plug-in* para registrar as ocorrências de *bad smells* identificados nos sistemas de software. Foram gerados os arquivos `JSON` no formato adequado para ser importado na plataforma `Landfill` e disponibilizado para o livre acesso da comunidade acadêmica.

Estudos empíricos também poderão utilizar-se dos dados disponibilizados, evitando que tais estudos utilizem de resultados provenientes de ferramentas automáticas e que podem não ter sido avaliadas adequadamente. Por fim, os dados levantados podem ser utilizados para definir refatoração dos sistemas de software escolhidos e, por conseguinte, aprimorar a qualidade deles. Neste trabalho, o oráculo gerado foi aplicado para avaliar três ferramentas de detecção de *bad smells*. Essa avaliação é apresentada no próximo capítulo.

Capítulo 6

Aplicação do Oráculo na Avaliação de Ferramentas de Detecção de *Bad Smells*

Neste capítulo, é apresentado um estudo comparativo entre três ferramentas de detecção automática de *bad smells*: `JDeodorant`, `JSpIRIT` e `FindSmells`. O estudo é realizado com base no oráculo criado neste trabalho. Para tal, realizamos uma comparação entre os dados desse oráculo com os resultados gerados pelas ferramentas que identificam ocorrências de *bad smells*.

Organizamos este capítulo da seguinte forma: Seção 6.1 descreve a metodologia adotada na coleta dos resultados das ferramentas `JDeodorant`, `JSpIRIT` e `FindSmells`. Em seguida, apresentamos brevemente essas ferramentas na Seção 6.2 e descrevemos os critérios para a escolha feita. Seção 6.3 traz os resultados retornados por cada uma dessas ferramentas e o confronto com os dados do oráculo. Seção 6.4 apresenta as considerações finais.

6.1 Metodologia

Os sistemas de software escolhidos na Seção 5.1.2 - `Aardvark, Logging, GanttProject, And Engine` e `Codec` - foram inspecionados manualmente para encontrar as instâncias dos cinco *bad smells* escolhidos na Seção 5.1.1. Tabela 6.1 mostra o resultado do oráculo consolidado, indicando as quantidades de *bad smells* identificados nos sistemas de software. Os dados do oráculo foram confrontados com os resultados gerados pelas ferramentas que identificam *bad smells*.

Tabela 6.1. Número de *bad smells* por software

Bad smells	**Aardvark**	**Logging**	**GanttProject**	**And Engine**	**Codec**
Data Class	0	0	15	16	2
Feature Envy	0	0	13	23	0
Large Class	4	1	18	9	6
Long Method	6	2	48	24	9
Refused Bequest	1	0	0	13	0

Na comparação entre os resultados reportados por uma ferramenta com os dados do oráculo, são identificados os casos descritos as seguir:

- Falsos Positivos (FP: casos em que a ferramenta indica a presença de um *bad smell* em uma determinada classe ou em um método, mas o oráculo não aponta essa ocorrência.

- Falso Negativo (FN): casos em que uma ferramenta não indica a presença do *bad smell* na classe ou no método, porém o oráculo informa que há.

- Verdadeiro Positivo (VP): refere-se à situação em que tanto a ferramenta quanto o oráculo indicam a presença de um determinado *bad smell* em uma classe ou método.

- Verdadeiro Negativo (VN): ocorre quando a ferramenta não detecta determinado *bad smell* na classes ou no método e o oráculo também não.

A Tabela 6.2 mostra a representação de como essa classificação ocorre para cada instância de *bad smell* segundo os critérios dessa avaliação.

Algumas ferramentas não discriminam os *bad smells* no nível de método, ao contrário do oráculo, que tem informações nesse nível. Nesses casos, para comparação do resultado, foi considerada a classe em que o método se localiza. Dessa forma, se o oráculo indica a ocorrência de um determinado *bad smell* em um método `m` pertencente a uma classe `A` e a ferramenta reporta a ocorrência do mesmo *bad smell* na classe `A` e não indica o método, considera-se um Verdadeiro Positivo (VP), i.e., uma situação em que o resultado da ferramenta está em concordância com o oráculo.

Métricas de avaliação estatística servem para mensurar o desempenho e assertividade das ferramentas de *bad smells* quando seus resultados são comparados com o oráculo. As métricas aplicadas para a avaliação das ferramentas foram *Precision*, *Recall* e *F-measure*. Azeem et al. [2019] cita essas métricas como as mais utilizadas em estudos que identificam *bad smells* por *Machine Learning*. Também foram as métricas

Tabela 6.2. Classificação de uma classe ou método pelo confronto entre os resultados das ferramentas e o oráculo

	Instância de *bad smell* identificada pela ferramenta	**Instância de *bad smell* não identificada pela ferramenta**
Instância de *bad smell* identificada pelo oráculo	Verdadeiro Positivo (VP)	Falso Negativo (FN)
Instância de *bad smell* não identificada pelo oráculo	Falso Positivo (FP)	Verdadeiro Negativo (VN)

adotadas nos estudos de Palomba et al. [2017], Palomba & Zaidman [2019] e Gadient et al. [2018]. Essas métricas são calculadas a partir do número de Falsos Positivos, Falsos Negativos e Verdadeiro Positivos. *Precision* (Precisão) indica qual foi a porcentagem de acerto das classificações dadas pelas ferramentas (Hossin M [2015]). O valor da *Precision* é um indicativo de que os resultados estão dentro do que é esperado e é calculada por meio da Fórmula 6.1.

$$Precision = \frac{VP}{VP + FP} \tag{6.1}$$

Recall (Revocação) é calculado conforme a Fórmula 6.2. *Recall* mostra dentre todas as situações de positivo, quantas foram identificadas corretamente (Hossin M [2015]).

$$Recall = \frac{VP}{VP + FN} \tag{6.2}$$

F-measure indica a exatidão dos resultados da ferramenta. Seu cálculo, determinado pela Fórmula 6.3, é dado pela média harmônica entre *Recall* e *Precision* (Hossin M [2015]).

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{6.3}$$

6.2 Escolha das Ferramentas

Os critérios de seleção das ferramentas foram:

1. Devem estar disponíveis para *download* gratuito pela *Internet*.
2. Devem ser capazes de identificar *bad smells* em projetos Java.

3. Devem:
   a) fornecer um ambiente próprio para a identificação e/ou coleta de dados ou
   b) ser possível adicioná-las ao ambiente `Eclipse` para serem executadas.
4. Devem ser capazes de identificar os *bad smells* presentes na lista de *bad smells* escolhidos na Seção 5.1.1.

Três ferramentas atenderam esses critérios e, portanto, foram selecionadas. São elas: `JDeodorant` (Fokaefs et al. [2011]), `JSpIRIT` (Vidal et al. [2015]) e `FindSmells` (Sousa et al. [2017]). A seguir, apresentamos suas principais características.

`JSpIRIT` é um acrônimo de *Java Smart Identification of Refactoring opportunITies*. `JSpIRIT` é um *plug-in* para a `IDE Eclipse` que visa auxiliar o desenvolvedor a identificar 10 *bad smells* e priorizá-los utilizando as estratégias de Lanza & Marinescu [2010]. São eles:

- *Brain Class*
- ***Brain Method***
- ***Data Class***
- *Disperse Coupling*
- ***Feature Envy***
- ***God Class***
- *Intensive Coupling*
- ***Refused Parent Bequest***
- *Shotgun Surgery*
- *Tradition Breaker*

Dos 10 *bad smells* que `JSpIRIT` identifica, destacamos em negrito cinco deles. Esses cinco fazem parte do escopo do oráculo que foi gerado neste trabalho. Para fins de avaliação, as ocorrências de *Brain Method* são comparadas com os de *Long Method*, uma vez que as definições do primeiro, dadas por Lanza et al. [2005], são semelhantes às do segundo, conforme aponta Arcelli Fontana et al. [2012]. Utilizamos para a inspeção a Versão 1.0.0.201410081128 que fornece o nome ao arquivo `jar` adicionado ao Eclipse.

`JDeodorant` é um *plug-in* do `Eclipse` que visa identificar *bad smells* e os resolver via a estratégia de refatoração apropriada. `JDeodorant` identifica cinco tipos de *bad smells*, são eles:

- ***Feature Envy***

- *Type Checking*
- ***Long Method***
- ***God Class***
- *Duplicated Code*

Dos cinco *bad smells* que `JDeodorant` identifica, três fazem parte do escopo do projeto e estão destacados na lista em negrito. Utilizamos a Versão 5.0.76 da ferramenta.

`FindSmells` é uma ferramenta que visa detectar *bad smells*. As estratégias de detecção são baseadas em métricas. A ferramenta tem como entrada um arquivo XML compativel com o Metrics, uma ferramenta de coleta de métricas que funciona como um *plug-in* do Eclipse. O arquivo contém as métricas do software a ser avaliado. Com essa ferramenta, o usuário pode definir suas próprias estratégias de detecção, considerando os valores das métricas recebidas pelo arquivo XML. `FindSmells` disponibiliza cinco estratégias de detecção, definidas por Sousa et al. [2017], para identificar os seguintes *bad smells*, que são também considerados neste trabalho:

- ***Feature Envy***
- ***Long Method***
- ***God Class***
- ***Refused Bequest***
- ***Data Class***

A Tabela 6.3 mostra quais das três ferramentas - `JDeodorant`, `JSpIRIT` e `FindSmells` - conseguem identificar os cinco *bad smells* escolhidos para gerar o oráculo. Conforme podemos observar, a escolha das três ferramentas nos oferece um conjunto de opções para identificação de *bad smells*. Isso nos permite comparar os resultados obtidos para cada *bad smell* por, pelo menos, duas ferramentas diferentes. `JSpIRIT` identifica *Brain Method*, porém o consideramos como *Long Method*, conforme explicado anteriormente, no Item `JSpIRIT`. Esse caso está indicado com um (*) na tabela.

6.3 Resultados

Esta seção apresenta os resultados encontrados por cada ferramenta para os sistemas de software escolhidos. Durante o processo, tivemos alguns problemas associados a cada uma das ferramentas. O `JSpIRIT` não sinaliza que os códigos precisam ser compiláveis,

Tabela 6.3. Tipos de *bad smells* identificados por ferramenta.

Bad smell	**JSpIRIT**	**JDeodorant**	**FindSmells**
Data Class	Identifica	Não identifica	Identifica
Feature Envy	Identifica	Identifica	Identifica
Large Class	Identifica	Identifica	Identifica
Long Method	Identifica*	Identifica	Identifica
Refused Bequest	Identifica	Não Identifica	Identifica

porém, assim como as duas outras ferramentas, produz resultados somente nos casos em que o projeto seja compilável. A restrição de compilação do `FindSmells` decorre do fato de que ele precisa da importação de um arquivo `xml` para fazer a classificação dos *bad smells* e esse arquivo é gerado pelo *plug-in* `Metrics` do `Eclipse`, cujo uso exige que o código seja executável. Encontramos a Versão 1.3.6 do `Metrics`, no entanto esta, por sua vez, exige uma versão mais antiga do `Eclipse`. O *plug-in* é capaz de calcular várias métricas de código durante o ciclo de *build* e avisa o usuário, na visão de *Problems View*, as violações de cada uma das métricas.

A preparação dos projetos para torná-los executáveis não foi simples e demandou algumas alterações no código. Foi necessário remover classes de testes dos projetos `Aardvark` e `Codec`, pois as mesmas possuiam dependências que não foram encontradas. A versão do primeiro software é do ano de 2012, enquanto a do segundo é de 2011 e não foi possível determinar as versões das bibliotecas externas necessárias para execução desses projetos. Essas dificuldades impediram que os projetos ficassem livre de erros de compilação o que, por consequência. impossibilitou suas inspeções completas pelas ferramentas já que foram excluídas as classes de testes.

Na extração dos resultados, observamos que a ferramenta `JDeodorant` identifica oportunidades de refatoração de código, portanto alguns dados ficaram duplicados. Por exemplo, um *Long Method* pode possuir três oportunidades de extração de método em seu código e, nesse caso, o resultado virá com três instâncias para o mesmo *bad smell* e mesmo método. Entretanto, o oráculo desenvolvido nesse trabalho não utilizou-se do mesmo critério para sua criação, assim o método do exemplo dado aparece somente uma vez. Com isso, foi necessário retirar essas duplicidades e deixar apenas um resultado por instância, evitando que a ferramenta tenha discrepância nos valores de Falsos Positivos. Outro ponto importante é que nenhum construtor foi considerado uma instância de *Long Method* pelo `JDeodorant`, o que não ocorreu com as outras ferramentas e nem mesmo pelo oráculo gerado neste trabalho.

Os resultados de *Feature Envy* obtidos pela ferramenta `FindSmells` são reportados no nível de classe, o que o diferencia das outras duas ferramentas e do próprio

oráculo que trazem as ocorrências por método. Nesse caso, os Verdadeiros Positivos foram sinalizados quando existia a presença de *Feature Envy* na classe e os Falsos Negativos também foram consolidados por classe, agrupando-se os métodos de uma mesma classe. Portanto, as somas dos Verdadeiros Positivos e dos Falsos Negativos são divergentes dos valores encontrados pelas outras ferramentas.

Todos os resultados apresentados foram obtidos usando a mesma máquina, executada em ambiente Windows 10 Home com um processador Intel(R) Core(TM) i5-8250U CPU 1.60GHz 1.80 GHz possuindo 12.0 GB de memória. A versão 2019-12 (4.14.0) do Eclipse foi utilizada para as ferramentas `JSpIRIT` e `JDeodorant`. Por sua vez, a ferramenta *Metrics* foi utilizada na versão Eclipse Juno 4.2.2.

As seções 6.3.1, 6.3.2 e 6.3.3 trazem os dados encontrados na comparação do oráculo com os resultados de, respectivamente, `JDeodorant`, `JSpIRIT` e `FindSmells`. Os dados são exibidos em função do número de Falsos Positivos (FP), Verdadeiros Positivos (VP) e Falsos Negativos (FN) que servem de insumos para os cálculos estatísticos. Os Falsos Negativos (FN) não são assinalados, mas representam o restante de classes ou métodos do software. Também são discriminados os resultados por cada um dos *bad smells* e software.

As tabelas exibem os valores de cada um dos *bad smells* utilizando uma sigla para representá-los. *Long Method* representado por LM, *Feature Envy* por FE, *Data Class* por DC, *Large Class* por LC e *Refused Bequest* por RB.

6.3.1 JDeodorant

Os resultados para cada um dos sistemas de software que compõem o oráculo estão exibidos na Tabela 6.4. Observamos que a ferramenta encontrou maior número de *bad smells* para o `GanttProject`, com um total de 227 instâncias, distribuídos da seguinte forma: 197, *Long Method*; 5, *Feature Envy*; e 25, *Large Class*. `GanttProject` é seguido por `And Engine` com 107 instâncias de *bad smells*, `Logging` com 52 instâncias de *bad smells*, `Codec` com 32 instâncias de *bad smells* e `Aardvark` com 15 instâncias de *bad smells*. Conforme pode ser visto na Tabela 6.4, nenhum Verdadeiro Positivo para *Feature Envy* foi encontrado por `JDeodorant`, porém, no caso de `Logging`, esse era também o comportamento esperado.

Os dados estatísticos para aferir o desempenho de `JDeodorant` foram extraídos a partir dos valores exibidos na Tabela 6.4. O cálculo foi feito para cada *bad smell* e software. Seus valores estão exibidos na Tabela 6.5. `JDeodorant` apresentou melhor *Precision* para o `Aardvark` com o valor de 0,33, que por sua vez obteve o quarto melhor *Recall* com um valor de 0,50. No caso especial do *Feature Envy* para o sistema

Tabela 6.4. Valores extraídos da comparação do resultado de JDeodorant com as instâncias do oráculo.

	And Engine			*Aardvark*			*Codec*			*GanttProject*			*Logging*		
	FN	FP	VP	FN	FP	VP	FN	FP	VP	FN	FP	VP	FN	FP	VP
LM	14	79	10	4	7	2	2	12	7	22	171	26	0	46	2
FE	23	3	0	0	1	0	0	10	0	13	5	0	0	0	0
LC	7	13	2	1	2	3	3	0	3	10	17	8	1	4	0
Total	44	95	12	5	10	5	5	22	10	45	193	34	1	50	2

`Logging`, a *Precision* e o *Recall* foram iguais a 1.00 indicando que o conjunto esperado e o encontrado foram idênticos, no caso citado, um conjunto vazio.

Tabela 6.5. Valores estatísticos de JDeodorant para cada sistema e cada *bad smell*

		LM	*FE*	*LC*	*Média*
And Engine	*Precision*	0.11	0.00	0.13	0.11
	Recall	0.42	0.00	0.22	0.21
Aardvark	*Precision*	0.22	0.00	1.00	0.33
	Recall	0.33	1.00	0.75	0.50
Codec	*Precision*	0.37	0.00	1.00	0.31
	Recall	0.78	1.00	0.50	0.67
GanttProject	*Precision*	0.13	0.00	0.32	0.15
	Recall	0.78	1.00	0.50	0.67
Logging	*Precision*	0.04	1.00	0.00	0.04
	Recall	1.00	1.00	0.00	0.67

A Tabela 6.6 exibe as médias dos resultados por cada um dos *bad smells* indicando que a `JDeodorant` tem melhor capacidade de indicar *Large Class* com *Precision Média* de 0,49, *Recall Médio* de 0,39 e *F-measure* de 0,44. O desempenho para determinar *Long Method* foi o pior dentre os outros *bad smells* com *Precision* de 0,18, *Recall* de 0,66 e *F-measure* de 0,28.

Tabela 6.6. Valores estatísticos médios de JDeodorant por *bad smell*

	Média		
	Precision	*Recall*	*F-measure*
LM	0.18	0.66	0.28
FE	0.20	0.80	0.32
LC	0.49	0.39	0.44
Média	0.29	0.62	0.39

6.3.2 JSpIRIT

Os resultados para cada um dos sistemas de software que compõem o oráculo estão exibidos na Tabela 6.7. Observamos que `JSpIRIT` encontrou maior número de *bad smells* para o `GanttProject` em um total de 146 instâncias, distribuídos da seguinte forma: 105, *Feature Envy*; 20, *Long Method*; 10, *Large Class*; 8, *Refused Bequest*; e 4, *Data Class*. `GanttProject` é seguido por `And Engine` com 113 instâncias de *bad smells*, `Codec` com 61 instâncias de *bad smells*, `Logging` com 43 instâncias de *bad smells* e `Aardvark` com 23 instâncias de *bad smells*.

Tabela 6.7. Valores extraídos da comparação do resultado de JSpIRIT com as instâncias do oráculo.

	And Engine			*Aardvark*			*Codec*			*GanttProject*			*Logging*		
	FN	FP	VP	FN	FP	VP	FN	FP	VP	FN	FP	VP	FN	FP	VP
LM	22	6	2	4	0	2	3	0	6	34	6	14	0	3	2
FE	19	51	4	0	16	0	0	37	0	12	104	1	0	31	0
DC	16	2	0	0	0	0	2	1	0	15	4	0	0	1	0
LC	5	6	4	2	2	2	1	5	5	10	2	8	0	2	1
RB	9	34	4	0	0	1	0	7	0	0	7	0	0	3	0
Total	71	99	14	6	18	5	6	50	11	71	123	23	0	40	3

Os dados estatísticos de desempenho da ferramenta `JSpIRIT` foram extraídos a partir dos valores exibidos na Tabela 6.7. O cálculo foi feito para cada *bad smells* e software. Seus valores estão exibidos na Tabela 6.8. `JSpIRIT` apresentou melhor *Precision* para o `Aardvark` que por sua vez teve o segundo melhor *Recall* com um valor de 0,77.

Tabela 6.8. Valores estatísticos de JSpIRIT para cada sistema e cada *bad smell*

		LM	*FE*	*DC*	*LC*	*RB*	*Média*
And Engine	*Precision*	0.25	0.07	0.00	0.40	0.11	0.17
	Recall	0.08	0.17	0.00	0.44	0.31	0.20
Aardvark	*Precision*	1.00	0.00	1.00	0.50	1.00	0.70
	Recall	0.33	1.00	1.00	0.50	1.00	0.77
Codec	*Precision*	1.00	0.00	0.00	0.50	0.00	0.30
	Recall	0.67	1.00	0.00	0.83	1.00	0.70
GanttProject	*Precision*	0.70	0.01	0.00	0.80	0.00	0.30
	Recall	0.67	1.00	0.00	0.83	1.00	0.70
Logging	*Precision*	0.40	0.00	0.00	0.33	0.00	0.15
	Recall	1.00	1.00	1.00	1.00	1.00	1.00

A Tabela 6.9 exibe os resultados médios para cada um dos *bad smells* indicando que a `JSpIRIT` tem melhor capacidade de indicar *Large Class* e *Large Method* ambos

com *F-measure* médio de 0,60, porém o primeiro possui maior *Precision* média, 0,67. O desempenho para determinar *Feature Envy* foi o pior dentre os outros *bad smells* com *F-measure* de 0,03.

Tabela 6.9. Valores estatísticos médios de JSpIRIT por *bad smell*

	Média		
	Precision	*Recall*	*F-measure*
LM	0.67	0.55	0.60
FE	0.02	0.83	0.03
DC	0.20	0.40	0.27
LC	0.51	0.72	0.60
RB	0.22	0.86	0.35
Média	0.32	0.67	0.44

6.3.3 FindSmells

Os resultados para cada um dos sistemas de software que compõem o oráculo estão exibidos na Tabela 6.10. Observamos que a `FindSmells` encontrou maior número de *bad smells* para o `GanttProject` em um total de 205 instâncias, distribuídos da seguinte forma: 33, *Long Method*; 58, *Feature Envy*; 51, *Data Class*; 13, *Large Class*; 50, *Refused Bequest*. `GanttProject` é seguido por `And Engine` (142 instâncias de *bad smells*), `Logging` (38 instâncias de *bad smells*), `Codec` (21 instâncias de *bad smells*) e `Aardvark` (20 instâncias de *bad smells*).

Tabela 6.10. Valores extraídos da comparação do resultado de FindSmells com as instâncias do oráculo.

	And Engine			Aardvark			Codec			GanttProject			Logging		
	FN	FP	VP	FN	FP	VP	FN	FP	VP	FN	FP	VP	FN	FP	VP
LM	17	6	7	4	3	2	2	2	7	30	15	18	0	9	2
FE	13	31	1	0	4	0	0	3	0	5	56	2	0	3	0
DC	15	37	1	0	4	0	2	5	0	8	44	7	0	2	0
LC	9	0	0	3	1	1	5	0	1	8	3	10	1	1	0
RB	4	50	9	0	4	1	0	3	0	0	50	0	0	21	0
Total	67	124	18	7	16	4	9	13	8	59	170	35	1	36	2

Os dados estatísticos de desempenho de `FindSmells` foram extraídos a partir dos valores exibidos na Tabela 6.10. O cálculo foi feito para cada *bad smells* e software e seus valores estão exibidos na Tabela 6.11. A ferramenta apresentou melhor média de *Precision* para o Aardvark que por sua vez teve o segundo melhor *Recall* com um valor de 0,70.

Tabela 6.11. Valores estatísticos de FindSmells para cada sistema e cada *bad smell*

		LM	*FE*	*DC*	*LC*	*RB*	*Média*
And Engine	*Precision*	0.54	0.03	0.03	1.00	0.15	0.35
	Recall	0.29	0.07	0.06	0.00	0.69	0.22
Aardvark	*Precision*	0.40	0.00	1.00	0.50	0.20	0.42
	Recall	0.33	1.00	1.00	0.25	1.00	0.72
Codec	*Precision*	0.78	0.00	0.00	1.00	0.00	0.36
	Recall	0.78	1.00	0.00	0.17	1.00	0.59
GanttProject	*Precision*	0.55	0.03	0.14	0.77	0.00	0.30
	Recall	0.78	0.29	0.00	0.17	1.00	0.45
Logging	*Precision*	0.18	0.00	0.00	0.00	0.00	0.04
	Recall	1.00	1.00	1.00	0.00	1.00	0.80

A Tabela 6.12 exibe a média dos resultados por cada um dos *bad smells* indicando que as ferramentas têm melhor capacidade de identificar *Large Method* com *F-measure* de 0,55. O desempenho para determinar *Feature Envy* foi o pior dentre os *bad smells*, com *F-measure* médio de 0,55. É importante lembrar que `FindSmells` indica uma classe com *Feature Envy* enquanto o oráculo o fez por método.

Tabela 6.12. Valores estatísticos médios de FindSmells por *bad smell*

	Média		
	Precision	*Recall*	*F-measure*
LM	0.49	0.64	0.55
FE	0.01	0.67	0.03
DC	0.23	0.41	0.30
LC	0.65	0.12	0.20
RB	0.07	0.94	0.13
Média	0.29	0.58	0.38

6.3.4 Análise de *F-Measure*

Com base nos resultados obtidos, a Tabela 6.13 exibe o *F-Measure* para cada uma das ferramentas usadas para a identificação dos *bad smells Long Method, Feature Envy, Data Class* e *Refused Bequest*. *F-Measure* é uma medida de acurácia de resultados da ferramenta em relação ao oráculo. Os valores dessa métrica vão de 0 a 1, sendo que quanto maior o valor, melhor, pois mais balanceados estão a precisão e a revocação.

`JSpIRIT` apresentou a maior exatidão na identificação dos *bad smells Long Method, Large Class* e *Refused Bequest*. Os resultados indicam também que a capacidade de `JSpIRIT` em identificar *Long Method* não foi impactada pela diferença

de definição entre os *bad smells Brain Method* e o *Long Method*. Por outro lado, `JDeodorant` apresentou a melhor exatidão para identificar *Feature Envy*, enquanto `FindSmells` apresentou a melhor exatidão para identificar *Data Class*.

No entanto, o maior valor observado para *F-Measure* foi 0,60, o que não é um valor alto. Os demais valores são ainda menores, 0,30 e 0,32. Esse resultado sugere que as ferramentas, em geral, não possuem acurácia alta na identificação de *bad smells*.

Tabela 6.13. Ferramenta com melhor acurácia para identificar cada *bad smell*

	Ferramenta	*F-measure*
Long Method	JSpIRIT	0.60
Feature Envy	JDeodorant	0.32
Data Class	FindSmells	0.30
Large Class	JSpIRIT	0.60
Refused Bequest	JSpIRIT	0.35

6.4 Considerações Finais

Este reportou um estudo aplicando o oráculo criado neste trabalho para avaliar três ferramentas que identificam *bad smells*: `JDeodorant`, `Find Smells` e `JSpIRIT`. Algumas alterações de código precisaram ser realizadas para `Aardvark` e `Codec` serem compilados. Para esses dois sistemas de software foi necessário retirar todas as classes de testes. Essa alteração pode ter influenciado nos valores de Falsos Negativos encontrados, que por sua vez, pode ter tido impacto no valor de *Recall* e consequentemente no *F-measure* nos casos correspondentes a esses sistemas de software.

Durante o cálculo dos valores de *Recall* e *Precision*, deparamos com algumas situações em que a equação possuia uma divisão por 0. Nesse caso, adotamos o valor 1 como resultado, pois os resultados encontrados foram iguais ao esperado, um conjunto vazio. O presente capítulo optou por fazer a análise dos resultados pela sua média, pois quando consideramos as somas de valores de Falsos Positivos, Falsos Negativos e Verdadeiros Positivos os valores estatísticos finais não sofreriam impactos para os casos onde a ferramenta encontra corretamente um conjunto vazio de *bad smells*.

Os resultados indicam que `JSpIRIT` é a ferramenta que apresenta, no geral, a melhor acurácia, na identificação de três dos *bad smells*: *Long Method, Large Class* e *Refused Bequest*, enquanto `JDeodorant` e `FindSmell` tiveram maior acurácia em um *bad smell*, *Feature Envy* e *Data Class*, respectivamente. Entretanto, os valores de acurácia das ferramentas não são altos. *JDeodorant* tem 0,39 de *F-measure* média, *JSpIRIT* tem 0,44 e *FindSmells*, 0,38.

Observando-se os dados de *Precision* detalhados, constata-se que, de forma geral, são baixos. *JDeodorant* tem 0,29 de precisão média, *JSpIRIT* tem 0,32 e *FindSmells*, 0,29. O *Recall* médio das ferramentas é melhor do que a precisão delas, mas, ainda assim, não são altos. *JDeodorant* tem 0,62 de revocação média, *JSpIRIT* tem 0,67 e *FindSmells*, 0,58. Dessa forma, de acordo com o oráculo criado neste trabalho, as ferramentas analisadas geram resultados com acurácia, precisão e revocação baixos, no geral.

Capítulo 7

Ameaças à Validade do Estudo

Segundo Wohlin et al. [2012], é praticamente impossível evitar todas as ameaças à validade de um estudo, sendo importante identificá-las e minimizá-las. Este capítulo discute as ameaças à validade deste estudo e as estratégias que adotamos para mitigá-las. O capítulo está organizado como segue. Seção 7.1 apresenta as possíveis formas de ameaça à validade de um trabalho experimental. Seção 7.2 destaca as ameças internas deste trabalho e como as mitigamos. Por fim, a Seção 7.3 apresenta as ameaças de conclusão deste trabalho e como as mitigamos.

7.1 Classificação de Ameaças à Validade

Segundo Wohlin et al. [2012], existem quatro ameaças à validade de um estudo experimental de Engenharia de Software divididas entre a teoria e observação. São elas:

- ***Validade de Conclusão.*** Diz respeito à relação entre o tratamento e o resultado. Por exemplo, uma relação estatística com um determinado significado.
- ***Validade Interna.*** O tratamento causa o resultado (o efeito). Devemos garantir que um relacionamento causal não é resultado de um fator que não foi controlado ou mensurado.
- ***Validade de Construção.*** Diz respeito a relação entre a teoria e a observação. Por exemplo, se um relacionamento é causal, devemos garantir que o tratamento reflete a construção da causa e que o resultado reflete a construção do efeito.
- ***Validade Externa.*** Trata das generalizações dos resultados. Por exemplo, se existe uma relação causal, ela pode ser generalizada fora do escopo do estudo.

As ameaças à validade dos resultados deste estudo estão distribuídas entre *Validade Interna* e *Validade de Conclusão*.

7.2 *Validade Interna*

Baseamos a Revisão Sistemática da Literatura (SLR) em oito repositórios digitais. No entanto, outras bases podem existir e conter trabalhos propondo oráculos de *bad smell*. Para mitigar essa ameaça usamos repositórios digitais conhecidos e confiáveis. Também com esse intuito, empregamos o *Google Scholar* como fonte de estudos primários abrangendo dessa forma uma gama de trabalhos que excede aqueles dos principais repositórios digitais.

A SLR utiliza uma ferramenta de terceiros, *Publish and Perish*, para obter os resultados do *Google Scholar* em planilhas. Embora não tenha sido realizada uma verificação sistemática dos dados, os valores brutos do resultado são os mesmos no formato *online* e o retornado pela ferramenta,

Na comparação entre os resultados obtidos pelas ferramentas de detecção de *bad smells* `JSpIRIT`, `JDeodorant` e `FindSmells`, os resultados são obtidos em diferentes formatos de arquivo e planilhas. Foi necessário agregar todos os dados em formato `Excel`. A agregração foi realizada manualmente. Para o caso do `JDeodorant` foi necessário retirar as linhas em duplicidade já que, diferentemente do oráculo, poderia possuir mais de uma instância em mesmo método. De modo a facilitar a comparação entre resultados, os dados foram ordenados em ordem alfabéticas de ambas bases de dados facilitando a conferência.

A pessoa que realiza a inspeção de um software está sujeita a perder de instâncias de *bad smells* durante a leitura de código ou pular e ignorar classes que deveriam ser analisadas. Para mitigar essa ameaça relacionada com os problemas na inspeção, desenvolvemos a ferramenta `InspectBSmell`. A utilização da ferramenta permite que o colaborador armazene em arquivo cada instância de *bad smell* que identificar, além de assinalar uma classe como lida após sua análise.

A geração do oráculo foi realizada manualmente. Durante a geração do oráculo, a subjetividade do avaliador é uma ameaça inerente à abordagem manual e depende da experiência e conhecimento do avaliador. Para mitigar essa ameaça, a análise foi realizada por quatro avaliadores. Foi realizado um nivelamento de conhecimento sobre *bad smells* para os avaliadores. Cada software passou integralmente por duas avaliações, sendo cada uma realizada por um avaliador distinto. Os casos de discordância de avaliação deles foram analisados por um terceiro avaliador, para se obter um parecer conclusivo.

7.3 *Validade de Conclusão*

Um dos problemas enfrentados ao realizar a SLR foi determinar quais abordagens foram usadas nos estudos primários para gerar os oráculos, uma vez que, em alguns casos, os documentos não trazem explicitamente essas informações. Para mitigar esta ameaça, quando nenhuma ferramenta era citada, classificávamos o trabalho como uma abordagem manual. Caso contrário, nós a classificávamos como mista quando (i) os estudos primários se referiam a qualquer ferramenta e (ii) indicavam que os dados foram tratados manualmente. Por fim, a abordagem era classificada por "ferramenta" quando nenhum tratamento de dados estava explícito no texto.

Existem estudos primários da SLR que usaram projetos proprietários e, portanto, seus autores evitam expor dados que geralmente são sensíveis. Nesses casos, contamos com a terminologia dos autores para determinar o tamanho do software, pois não temos acesso ao código-fonte.

As conclusões acerca da avaliação das ferramentas utilizando o oráculo dependem das métricas e indicadores utilizados na avaliação. Neste trabalho, utilizamos indicadores numéricos que são recorrentemente utilizados em estudos semelhantes na literatura: *Precision*, *Recall* e *F-measure*.

Capítulo 8

Conclusão

A manutenção é a atividade responsável pelo maior custo total de um sistema. Medidas que visem diminuir esse custo devem ser tomadas durante todo o ciclo de vida do sistema. Conhecendo quais são os pontos que supostamente pode ocorrer uma falha de projeto, intervenções podem ser realizadas para facilitar a manutenção e evitar caminhos que elevam os riscos de dispêndio de recursos materiais e humanos. Com esse objetivo, muitos trabalhos têm sido realizados sobre *bad smells* em software, especialmente estudos para propor estratégias e ferramentas de detecção automática de *bad smells* e estudos empíricos que aplicam essas estratégias e ferramentas. Todavia, a acurácia dessas abordagens constituem importante ameaça aos estudos que as aplicam. A avaliação apropriada dessas estratégias e ferramentas depende da existência de oráculos de *bad smells* .

O objetivo da pesquisa desenvolvida nesta dissertação de mestrado foi catalogar os oráculos disponíveis na literatura, bem como criar um oráculo de um conjunto de *bad smells* identificados em um conjunto de sistemas de software. Para catalogar os oráculos existentes, foi realizada um Revisão Sistemática da Literatura.

O oráculo criado neste trabalho compreende cinco *bad smells*: *Long Method*, *Large Class*, *Feature Envy*, *Data Class*, *Refused Bequest*. Os dados do oráculo são de cinco sistemas de software Java abertos: `Aardvark`, `And Engine`, `Apache Commons Codec`, `Apache Commons Logging` e `GanttProject`.

A abordagem manual foi escolhida para a inspeção do código fonte e criação do oráculo, justificada pelo número escasso de oráculos que implementam tal abordagem e são disponíveis *online*, conforme resultado encontrado pela Revisão Sistemática da Literatura realizada neste trabalho. O oráculo foi construído com base na análise dos sistemas de software por quatro especialistas, visando mitigar os impactos da subjetividade da abordagem manual nos resultados do estudo. O oráculo está disponí-

vel em *www.llp.dcc.ufmg.br/products/oraculo/Resultado.xlsx* e no ambiente `LandFill` (Palomba et al. [2015]).

A ferramenta `InspectBSmell` foi projetada e implementada neste trabalho para apoiar a inspeção manual dos sistemas pelos avaliadores. A ferramenta pode ser utilizada para facilitar o trabalho de quem deseja fazer uma inspeção manual de sistema através do Eclipse. Os arquivos necessários para sua instalação da ferramenta estão disponíveis *online* e os procedimentos para fazê-lo descritos no corpo do presente trabalho.

O oráculo foi aplicado para avaliar três ferramentas de detecção de *bad smells*: `JDeodorant` (Fokaefs et al. [2011]), `JSpIRIT` (Vidal et al. [2015]) e `FindSmells` (Sousa et al. [2017]). Os resultados indicam que `JSpIRIT` é a ferramenta que apresenta a melhor acurácia na identificação de três *bad smells*: *Long Method, Large Class* e *Refused Bequest*. `JDeodorant` apresentou maior acurácia na identificação de *Feature Envy* e `FindSmell`, de *Data Class*. Dessa forma, ferramenta que aprensentou o resultado mais aproximado ao que foi identificado pelo oráculo do trabalho foi `JSpIRIT`. Todavia, de forma geral, a precisão, a revocação e a acurácia das ferramentas foram baixas em todas as ferramentas.

As contribuições deste trabalho podem ser úteis para os pesquisadores que realizam estudos sobre *bad smells*, pois provê um arcabouço de *bad smells* confiáveis e validados que podem ser empregados na criação de abordagens de detecção, na criação de ferramentas, na condução de trabalhos empíricos sobre o assunto. A ferramenta `InspectBSmell` pode ser empregada para a criação de novos oráculos de *bad smells*. Os resultados também podem ser utilizados no ensino sobre o assunto, pois fornece um conjunto de exemplos reais de ocorrências de *bad smells*.

8.1 Lições Aprendidas

Algumas lições foram colhidas ao longo do desenvolvimento do trabalho e servem de sugestão para pesquisadores que desejam realizar estudos semelhantes ao do presente trabalho.

Ao longo do processo foi importante a prática de tomar notas sobre as escolhas que foram tomadas em cada etapa do trabalho, especialmente durante a Revisão Sistemática da Literatura (SLR). A primeira rodada de obtenção de artigos dos repositórios digitais ocorreu ainda em 2017. No decorrer do tempo, foi necessário fazer novas rodadas de pesquisa para manter os resultados da SLR atualizados. Os novos resultados foram confrontados com os antigos para identificar somente o que eram os resultados

novos e facilitar o seu processamento.

A necessidade de cooperação de terceiros para a criação do oráculo é um fator limitante que deve ser compartilhada por qualquer trabalho da natureza deste. É importante que a escolha dos participantes leve em conta o nível de liberdade que o pesquisador terá para cobrar engajamento e resultados. Inicialmente, além dos três colaboradores na inspeção dos sistemas de software, esperávamos contar com a participação de mais duas pessoas, porém a colaboração não foi concretizada e acarretou atrasos no cronograma do trabalho.

Outra lição importante é quanto à dificuldade de importação de sistemas de software abertos para ambientes de desenvolvimento local. Neste trabalho, tivemos dificuldades desse tipo com dois projetos, o que também gerou impacto no cronograma do trabalho.

8.2 Trabalhos Futuros

A ferramenta `InspectBSmell` pode ser aprimorada com recursos que concedam a coloboração entre os usuários, por exemplo, permitindo que inspeções do software ocorra de forma simultânea e compartilhada. A ferramenta também pode ter mecanismos próprios para identificar diferenças entre inspeções e permitir que a consolidação do oráculo anscorresse dentro do próprio ambiente Eclipse.

O oráculo criado neste trabalho pode ser estendido com mais sistemas e com dados de outros *bad smells*. Além disso, o oráculo pode ser utilizado para orientar a melhoria de técnicas e ferramentas de detecção de *bad smells*, uma vez que serve como elemento de avaliação delas.

A comparação entre os resultados de ferramentas de detecção de *bad smell* em software com um oráculo de *bad smell* pode prover *insights* para determinar onde os métodos de detecção falham. Uma fase de dissecação pode ser conduzida para conferir a existência de um padrão de falha entre o conjunto de instâncias de *bad smells* não identificados por uma ferramenta. Os resultados devem guiar um esforço de otimização do código para melhorar a precisão das ferramentas de detecção de *bad smell* em software.

Referências Bibliográficas

Abílio, R.; Padilha, J.; Figueiredo, E. & Costa, H. (2015). Detecting code smells in software product lines – an exploratory study. Em *2015 12th International Conference on Information Technology - New Generations*, pp. 433–438. ISSN .

Amorim, L.; Costa, E.; Antunes, N.; Fonseca, B. & Ribeiro, M. (2015). Experience report: Evaluating the effectiveness of decision trees for detecting code smells. Em *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 261–269. ISSN .

Aniche, M.; Bavota, G.; Treude, C.; Deursen, A. V. & Gerosa, M. A. (2016). A validated set of smells in model-view-controller architectures. Em *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 233–243. ISSN .

Aniche, M.; Bavota, G.; Treude, C.; Deursen, A. V. & Gerosa, M. A. (2016). A validated set of smells in model-view-controller architectures. Em *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 233–243. ISSN .

Arcelli Fontana, F.; Braione, P. & Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11.

Arcelli Fontana, F.; Mäntylä, M. V.; Zanoni, M. & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Softw. Engg.*, 21(3):1143--1191. ISSN 1382-3256.

Azeem, M. I.; Palomba, F.; Shi, L. & Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115 – 138. ISSN 0950-5849.

Börjesson, L. (2016). Research outside academia? an analysis of resources in extra-academic report writing. ASIST '16, USA. American Society for Information Science.

Brown, W. J.; Malveau, R. C.; McCormick, H. W. S. & Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis: Refactoring Software, Architecture and Projects in Crisis*. John Wiley & Sons, 1. auflage edição. ISBN 0471197130.

Chen, B. & Jiang, Z. M. J. (2017). Characterizing and detecting anti-patterns in the logging code. Em *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pp. 71--81, Piscataway, NJ, USA. IEEE Press.

Chen, Z.; Chen, L.; Ma, W. & Xu, B. (2016). Detecting code smells in python programs. Em *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pp. 18–23. ISSN .

Christopoulou, A.; Giakoumakis, E. A.; Zafeiris, V. E. & Soukara, V. (2012). Automated refactoring to the strategy design pattern. *Inf. Softw. Technol.*, 54(11):1202–1214. ISSN 0950-5849.

Clayberg, E. & Rubel, D. (2008). Code pro analytix. https://wiki.eclipse.org/images/7/75/CodeProDatasheet.pdf acessado dia 25/06/2020.

d. Reis, J. P.; Brito e Abreu, F. & d. F. Carneiro, G. (2016). Code smells incidence: Does it depend on the application domain? Em *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pp. 172–177. ISSN .

Danphitsanuphan, P. & Suwantada, T. (2012). Code smell detecting tool and code smell-structure bug relationship. Em *2012 Spring Congress on Engineering and Technology*, pp. 1–5. ISSN .

Dhambri, K.; Sahraoui, H. & Poulin, P. (2008). Visual detection of design anomalies. Em *2008 12th European Conference on Software Maintenance and Reengineering*, pp. 279–283. ISSN 1534-5351.

Falke, R.; Frenzel, P. & Koschke, R. (2008). Empirical evaluation of clone detection using syntax suffix trees. *Empirical Softw. Engg.*, 13(6):601--643. ISSN 1382-3256.

Fard, A. M. & Mesbah, A. (2013). Jsnose: Detecting javascript code smells. Em *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 116–125. ISSN .

Fenske, W.; Schulze, S.; Meyer, D. & Saake, G. (2015). When code smells twice as much: Metric-based detection of variability-aware code smells. Em *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 171–180. ISSN .

Fernandes, E.; Oliveira, J.; Vale, G.; Paiva, T. & Figueiredo, E. (2016). A review-based comparative study of bad smell detection tools. Em *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pp. 18:1--18:12, New York, NY, USA. ACM.

Fokaefs, M.; Tsantalis, N.; Stroulia, E. & Chatzigeorgiou, A. (2011). Jdeodorant: identification and application of extract class refactorings. Em *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1037–1039.

Fontana, F. A.; Ferme, V. & Spinelli, S. (2012). Investigating the impact of code smells debt on quality code evaluation. Em *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 15–22. ISSN .

Fontana, F. A.; Ferme, V. & Spinelli, S. (2012). Investigating the impact of code smells debt on quality code evaluation. Em *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 15–22. ISSN .

Fontana, F. A.; Ferme, V.; Zanoni, M. & Roveda, R. (2015). Towards a prioritization of code debt: A code smell intensity index. Em *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pp. 16–24. ISSN .

Foundation, E. (2010). Jface. https://wiki.eclipse.org/JFace acessado dia 25/06/2020.

Foundation, E. (2020a). Core. https://www.eclipse.org/eclipse/platform-core/ acessado dia 25/06/2020.

Foundation, E. (2020b). Eclipse java development tools (jdt). https://www.eclipse.org/jdt acessado dia 25/06/2020.

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Fowler, M.; Beck, K.; Brant, J.; Opdyke, W. & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gadient, P.; Ghafari, M.; Frischknecht, P. & Nierstrasz, O. (2018). Security code smells in android ICC. *CoRR*, abs/1811.12713.

Hecht, G.; Moha, N. & Rouvoy, R. (2016). An empirical study of the performance impacts of android code smells. Em *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pp. 59--69, New York, NY, USA. ACM.

Hermans, F. & Aivaloglou, E. (2016). Do code smells hamper novice programming? a controlled experiment on scratch programs. Em *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–10. ISSN .

Hossin M, S. M. (2015). A review on evaluation metrics for data classification evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5(2):1--11. ISSN 2231-007X.

inFusion Hydrogen (2012). Code pro analytix. https://marketplace.eclipse.org/content/infusion-hydrogen acessado dia 25/06/2020.

Kaur, A.; Kaur, K. & Jain, S. (2016). Predicting software change-proneness with code smells and class imbalance learning. Em *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 746–754. ISSN .

Kaur, A.; Kaur, K. & Jain, S. (2016). Predicting software change-proneness with code smells and class imbalance learning. Em *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 746–754. ISSN .

Kessentini, M. & Ouni, A. (2017). Detecting android smells using multi-objective genetic programming. Em *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 122–132. ISSN .

Kessentini, M.; Vaucher, S. & Sahraoui, H. (2010). Deviance from perfection is a better criterion than closeness to evil when identifying risky code. Em *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pp. 113--122, New York, NY, USA. ACM.

Khomh, F.; Di Penta, M. & Guéhéneuc, Y.-G. (2009a). An exploratory study of the impact of code smells on software change-proneness. pp. 75–84.

Khomh, F.; Vaucher, S.; Guéhéneuc, Y. G. & Sahraoui, H. (2009b). A bayesian approach for the detection of code and design smells. Em *2009 Ninth International Conference on Quality Software*, pp. 305–314. ISSN 1550-6002.

Khomh, F.; Vaucher, S.; Guéhéneuc, Y.-G. & Sahraoui, H. (2011). Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559 – 572. ISSN 0164-1212. The Ninth International Conference on Quality Software.

Kitchenham, B. & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering.

Lanza, M. & Marinescu, R. (2010). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Publishing Company, Incorporated, 1st edição. ISBN 3642063748.

Lanza, M.; Marinescu, R. & Ducasse, S. (2005). *Object-Oriented Metrics in Practice*. Springer-Verlag, Berlin, Heidelberg. ISBN 3540244298.

Lavoie, T. & Merlo, E. (2011). Automated type-3 clone oracle using levenshtein metric. Em *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pp. 34--40, New York, NY, USA. ACM.

Lehman, M. M.; Ramil, J. F.; Wernick, P. D.; Perry, D. E. & Turski, W. M. (1997). Metrics and laws of software evolution-the nineties view. Em *Proceedings Fourth International Software Metrics Symposium*, pp. 20–32. IEEE.

Li, N. & Offutt, J. (2017). Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395. ISSN 0098-5589.

Macia, I.; Arcoverde, R.; Garcia, A.; Chavez, C. & von Staa, A. (2012). On the relevance of code anomalies for identifying architecture degradation symptoms. Em *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 277–286. ISSN 1534-5351.

Maiga, A.; Ali, N.; Bhattacharya, N.; Sabane, A.; Guéhéneuc, Y.-G.; Antoniol, G. & Aïmeur, E. (2012). Support vector machines for anti-pattern detection. *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 278–281.

Mannan, U. A.; Ahmed, I.; Almurshed, R. A. M.; Dig, D. & Jensen, C. (2016). Understanding code smells in android applications. Em *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pp. 225--234, New York, NY, USA. ACM.

Mantyla, M. V.; Vanhanen, J. & Lassenius, C. (2004). Bad smells - humans as code critics. Em *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 399–408. ISSN 1063-6773.

Mantyla, M. V.; Vanhanen, J. & Lassenius, C. (2004). Bad smells - humans as code critics. Em *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 399–408. ISSN 1063-6773.

Marinescu, C.; Marinescu, R.; Mihancea, P. F. & Wettel, R. (2005). iplasma: An integrated platform for quality assessment of object-oriented design. Em *In ICSM (Industrial and Tool Volume*, pp. 77--80. Society Press.

Marinescu, R. (2001). Detecting design flaws via metrics in object-oriented systems. Em *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*, pp. 173–182.

Moha, N.; Gueheneuc, Y.-G.; Duchien, L. & Le Meur, A.-F. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.*, 36(1):20--36. ISSN 0098-5589.

Oizumi, W.; Garcia, A.; da Silva Sousa, L.; Cafeo, B. & Zhao, Y. (2016a). Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. Em *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 440--451, New York, NY, USA. ACM.

Oizumi, W.; Garcia, A.; da Silva Sousa, L.; Cafeo, B. & Zhao, Y. (2016b). Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. Em *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 440--451, New York, NY, USA. ACM.

Oizumi, W. N.; Garcia, A. F.; Colanzi, T. E.; Ferreira, M. & v. Staa, A. (2014). When code-anomaly agglomerations represent architectural problems? an exploratory study. Em *2014 Brazilian Symposium on Software Engineering*, pp. 91–100. ISSN .

Olbrich, S.; Cruzes, D. S.; Basili, V. & Zazworka, N. (2009). The evolution and impact of code smells: A case study of two open source systems. Em *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 390–400. ISSN 1949-3770.

Ouni, A.; Kessentini, M.; Sahraoui, H.; Inoue, K. & Deb, K. (2016a). Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Trans. Softw. Eng. Methodol.*, 25(3):23:1--23:53. ISSN 1049-331X.

Ouni, A.; Kessentini, M.; Sahraoui, H.; Inoue, K. & Deb, K. (2016b). Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Trans. Softw. Eng. Methodol.*, 25(3):23:1--23:53. ISSN 1049-331X.

Ouni, A.; Kessentini, M.; Sahraoui, H.; Inoue, K. & Hamdi, M. S. (2015a). Improving multi-objective code-smells correction using development history. *Journal of Systems and Software*, 105:18 – 39. ISSN 0164-1212.

Ouni, A.; Kessentini, M.; Sahraoui, H.; Inoue, K. & Hamdi, M. S. (2015b). Improving multi-objective code-smells correction using development history. *Journal of Systems and Software*, 105:18 – 39. ISSN 0164-1212.

Paiva, T.; Damasceno, A.; Figueiredo, E. & Sant'Anna, C. (2017). On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5:7:1--7:28. ISSN 2195-1721.

Palomba, F. (2015). Textual analysis for code smell detection. Em *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pp. 769–771. ISSN 0270-5257.

Palomba, F.; Bavota, G.; Penta, M. D.; Fasano, F.; Oliveto, R. & Lucia, A. D. (2017). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*. ISSN 1573-7616.

Palomba, F.; Bavota, G.; Penta, M. D.; Fasano, F.; Oliveto, R. & Lucia, A. D. (2018). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188--1221. ISSN 1573-7616.

Palomba, F.; Bavota, G.; Penta, M. D.; Oliveto, R. & Lucia, A. D. (2014). Do they really smell bad? a study on developers' perception of bad code smells. Em *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 101–110. ISSN 1063-6773.

Palomba, F.; Bavota, G.; Penta, M. D.; Oliveto, R.; Lucia, A. D. & Poshyvanyk, D. (2013a). Detecting bad smells in source code using change history information. Em *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 268–278. ISSN .

Palomba, F.; Bavota, G.; Penta, M. D.; Oliveto, R.; Lucia, A. D. & Poshyvanyk, D. (2013b). Detecting bad smells in source code using change history information. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 268–278.

Palomba, F.; Bavota, G.; Penta, M. D.; Oliveto, R.; Poshyvanyk, D. & De Lucia, A. (2015). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489. ISSN 0098-5589.

Palomba, F.; Di Nucci, D.; Panichella, A.; Zaidman, A. & De Lucia, A. (2017). Lightweight detection of android-specific code smells: The adoctor project. Em *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 487–491.

Palomba, F.; Nucci, D. D.; Tufano, M.; Bavota, G.; Oliveto, R.; Poshyvanyk, D. & Lucia, A. D. (2015). Landfill: An open dataset of code smells with public evaluation. Em *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 482–485. ISSN 2160-1852.

Palomba, F.; Oliveto, R. & Lucia, A. D. (2017a). Investigating code smell co-occurrences using association rule learning: A replicated study. Em *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pp. 8–13. ISSN .

Palomba, F.; Oliveto, R. & Lucia, A. D. (2017b). Investigating code smell co-occurrences using association rule learning: A replicated study. *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pp. 8–13.

Palomba, F.; Panichella, A.; De Lucia, A.; Oliveto, R. & Zaidman, A. (2016). A textual-based technique for smell detection. Em *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–10. Exported from https://app.dimensions.ai on 2019/02/25.

Palomba, F.; Panichella, A.; Zaidman, A.; Oliveto, R. & De Lucia, A. (2018). The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*, 44(10):977–1000. ISSN 0098-5589.

Palomba, F. & Zaidman, A. (2019). The smell of fear: on the relation between test smells and flaky tests. *Empirical Software Engineering*, pp. 1–40.

Peters, R. & Zaidman, A. (2012). Evaluating the lifespan of code smells using software repository mining. Em *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 411–416. ISSN 1534-5351.

PMD (2012). Pmd. https://marketplace.eclipse.org/content/pmd-eclipse-plugin acessado dia 25/06/2020.

Rezende, D. A. (2005). *Engenharia de software e sistemas de informação*. Brasport.

Saboury, A.; Musavi, P.; Khomh, F. & Antoniol, G. (2017). An empirical study of code smells in javascript projects. Em *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 294–305. ISSN .

Sahin, D. (2016). A multi-level framework for the detection, prioritization and testing of software design defects.

Sharma, T. & Spinellis, D. (2018). A survey on software smells. *Journal of Systems and Software*, 138:158 – 173. ISSN 0164-1212.

Sirikul, K. & Soomlek, C. (2016). Automated detection of code smells caused by null checking conditions in java programs. Em *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 1–7. ISSN .

Sobrinho, E. V. P.; Lucia, A. & Maia, M. (2018). A systematic literature review on bad smells — 5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering*, pp. 1–1.

Sommerville, I. (2011). *Engenharia de Software*. Pearson Education.

Sousa, B. L.; Souza, P. P.; Fernandes, E. M.; Ferreira, K. A. M. & Bigonha, M. A. S. (2017). Findsmells: Flexible composition of bad smell detection strategies. Em *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 360–363.

Szőke, G.; Nagy, C.; Ferenc, R. & Gyimóthy, T. (2014). A case study of refactoring large-scale industrial systems to efficiently improve source code quality. Em Murgante, B.; Misra, S.; Rocha, A. M. A. C.; Torre, C.; Rocha, J. G.; Falcão, M. I.; Taniar, D.; Apduhan, B. O. & Gervasi, O., editores, *Computational Science and Its Applications – ICCSA 2014*, pp. 524--540, Cham. Springer International Publishing.

Technology, O. (2010). Java foundation classes (jfc).

Terra, R.; Valente, M. T.; Miranda, S. & Sales, V. (2018). Jmove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software*, 138:19 – 36. ISSN 0164-1212.

Vale, G.; Albuquerque, D.; Figueiredo, E. & Garcia, A. (2015). Defining metric thresholds for software product lines: A comparative study. Em *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pp. 176--185, New York, NY, USA. ACM.

Vale, G. A. D. & Figueiredo, E. M. L. (2015). A method to derive metric thresholds for software product lines. Em *2015 29th Brazilian Symposium on Software Engineering*, pp. 110–119. ISSN .

Vale, G. A. D. & Figueiredo, E. M. L. (2015). A method to derive metric thresholds for software product lines. Em *2015 29th Brazilian Symposium on Software Engineering*, pp. 110–119. ISSN .

Vidal, S.; Vazquez, H.; Diaz-Pace, J. A.; Marcos, C.; Garcia, A. & Oizumi, W. (2015). Jspirit: a flexible tool for the analysis of code smells. Em *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pp. 1–6.

Wagey, B. C.; Hendradjaya, B. & Mardiyanto, M. S. (2015). A proposal of software maintainability model using code smell measurement. Em *2015 International Conference on Data and Software Engineering (ICoDSE)*, pp. 25–30. ISSN .

Walton, L. (1999). Eclipse metrics plugin. http://eclipse-metrics.sourceforge.net/ acessado dia 25/06/2020.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. Em *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pp. 38:1--38:10, New York, NY, USA. ACM.

Wohlin, C.; Runeson, P.; Hst, M.; Ohlsson, M. C.; Regnell, B. & Wessln, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated. ISBN 3642290434.

Yamashita, A. (2014a). Assessing the capability of code smells to explain maintenance problems: An empirical study combining quantitative and qualitative data. *Empirical Softw. Engg.*, 19(4):1111--1143. ISSN 1382-3256.

Yamashita, A. (2014b). Assessing the capability of code smells to explain maintenance problems: An empirical study combining quantitative and qualitative data. *Empirical Softw. Engg.*, 19(4):1111--1143. ISSN 1382-3256.

Yamashita, A. & Moonen, L. (2013). To what extent can maintenance problems be predicted by code smell detection? – an empirical study. *Information and Software Technology*, 55(12):2223 – 2242. ISSN 0950-5849.

Yamashita, A.; Zanoni, M.; Fontana, F. A. & Walter, B. (2015). Inter-smell relations in industrial and open source systems: A replication and comparative analysis. Em *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 121–130. ISSN .

Zhao, X.; Xuan, X. & Li, S. (2015). An empirical study of long method and god method in industrial projects. Em *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pp. 109–114. ISSN .