Applying Software Metric Thresholds for Detection of Bad Smells

Priscila P. Souza
DCC/UFMG
Belo Horizonte, Brasil
priscilinhapsouza@gmail.com

Bruno L. Sousa
DCC/UFMG
Belo Horizonte, Brasil
bruno.luan.sousa@dcc.ufmg.br

Kecia A. M. Ferreira
DECOM/CEFET-MG
Belo Horizonte MG, Brasil
kecia@decom.cefetmg.br

Mariza A. S. Bigonha
DCC/UFMG
Belo Horizonte MG, Brasil
mariza@dcc.ufmg.br

ABSTRACT

Software metrics can be an effective measurement tool to assess the quality of software. In the literature, there are a lot of software metrics applicable to systems implemented in different paradigms like Objects Oriented Programming (OOP). To guide the use of these metrics in the evaluation of the quality of software systems, it is important to define their thresholds. The aim of this study is to investigate the effectiveness of the thresholds in the evaluation of the quality of object oriented software. To do that, we used a threshold catalog of 18 software metrics derived from 100 software systems to define detection strategies for five *bad smells*. They are: Large Class, Long Method, Data Class, Feature Envy and Refused Bequest. We investigate the effectiveness of the thresholds in detection analysis of 12 software systems using these strategies. The results obtained by the proposed strategies were compared with the results obtained by the tools JDeodorant and JSPiRIT, used to identify bad smells. This study shows that the metric thresholds were significantly effective in supporting the detection of *bad smells*.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; **Metrics**; *Measurement*; • **Social and professional topics** → **Quality assurance**;

KEYWORDS

Software metrics, software quality, thresholds, bad smells detection.

ACM Reference Format:
Priscila P. Souza, Bruno L. Sousa, Kecia A. M. Ferreira, and Mariza A. S. Bigonha. 2017. Applying Software Metric Thresholds for Detection of Bad Smells. In *Proceedings of 11th Brazilian Symposium on Software Components Architecture, and Reuse, Fortaleza, Ceará, BRA, September 2017 (SBCARS'17),* 10 pages.
https://doi.org/10.1145/3132498.3134268

SBCARS'17, September 2017, Fortaleza, Ceará, BRA

ACM ISBN 978-1-4503-5325-0/17/09…$15.00
https://doi.org/10.1145/3132498.3134268

1 INTRODUÇÃO

Métricas de software proporcionam medidas quantitativas para avaliar a qualidade dos projetos de software, permitindo aos engenheiros de software efetuarem alterações no decorrer do processo de desenvolvimento a fim de promover a melhoria da qualidade do produto final. Apesar da importância das métricas no gerenciamento da qualidade de software orientados por objeto (SOO), elas ainda não são efetivamente usadas na indústria de software [12, 25, 34]. Uma possível razão é que, para realizar uma medição efetiva do software e elevar o uso de métricas para a tomada de decisão, é essencial associar a elas a definição de valores referência (do inglês, *thresholds*) significativos [5, 12, 13].

A definição de métodos para derivação de valores referência para métricas de SOO é foco de diversos estudos [1, 5, 7, 10, 12, 13, 16, 22, 28, 37]. Embora haja dezenas de métricas e uma quantidade razoável de métodos definidos para derivação desses valores, há poucas métricas com valores referência conhecidos, o que dificulta e até mesmo inviabiliza o gerenciamento da qualidade de software por meios quantitativos. Ademais, embora os valores referência para métricas de software existentes tenham sido avaliados, tais avaliações não são abrangentes. O trabalho de Filó et al. [13] possui o maior número de métricas com valores referência identificados, 18. Porém, eles só analisaram a relação de algumas dessas métricas com os *bad smells: God Class* e *Long Method*.

Bad smells se refere às características encontradas nas estruturas de códigos que indicam que eles podem estar com problemas e precisam ser reestruturados de alguma forma. *Bad smells* não são a causa direta de falhas na aplicação, são apenas um conjunto de más práticas de projeto, mas que podem influenciar indiretamente para a inserção de erros responsáveis por futuras falhas [15].

Dado esse cenário, é preciso um estudo empírico mais abrangente para verificar a aplicabilidade de valores referência. É importante, dentre outros aspectos, investigar se os valores referência são úteis na identificação de *bad smells* e na predição de falhas, itens que impactam diretamente na qualidade de SOO. Neste artigo não foram consideradas falhas. Para auxiliar na identificação dos *bad smells* definidos por Fowler [15] usamos nesse estudo os valores referência de métrica de SOO propostos por Filó et al. [13].

Para direcionar a pesquisa proposta foram definidas duas questões de pesquisa: *QP1: os valores referência de métricas de software orientados por objetos auxiliam a identificar bad smells*? *QP1.1: qual é a eficácia da detecção de bad smells utilizando-se estratégias baseadas*

em valores referência e tomando-se como base os resultados gerados por ferramentas de detecção de bad smells? Para respondê-las propusemos: (i) uma estratégia de detecção de *bad smells* baseada em métricas e seus valores referência, uma vez que a pesquisa realizada anteriormente por Souza [33] revela poucos estudos que usam valores referência para identificar *bad smells* [18, 21, 23] (Seção 2); (ii) a fim de aferir a qualidade dos valores referência derivados, foi conduzido um estudo de caso para avaliar a eficácia dos valores referência na detecção de cinco *bad smells* em 12 sistemas do Qualitas.class Corpus [35]. Nesse estudo de caso usamos as estratégias de detecção apresentadas neste artigo e as ferramentas `JSpIRIT` [39] [38] e `JDeodorant` [36], muito usadas para identificar *bad smells* automaticamente. As estratégias de detecção propostas são descritas na Seção 2. O estudo de caso e seus resultados são exibidos na Seção 3. Seção 4 trata das ameaças à validade. Seção 5 mostra alguns trabalhos relacionados ao nosso. Seção 6 conclui este artigo.

2 ESTRATÉGIAS PROPOSTAS PARA DETECÇÃO DE *BAD SMELLS*

Estratégia de detecção é definida como uma expressão quantificável de uma regra para verificar se fragmentos do código-fonte estão em conformidade com essa regra [8, 21]. Os componentes para a definição de estratégias de detecção de *bad smells* são: mecanismos de filtragem de métricas e de composição. O mecanismo de filtragem permite a redução do conjunto inicial de dados, para que se possa verificar uma característica especial de fragmentos que têm suas métricas capturadas. Os mecanismos de composição permitem combinar diferentes filtros, baseados em métricas de software distintas, por meio de conectivos lógicos, permitindo, assim, a mescla de métricas para obter informações relevantes. Para definir um filtro de dados é preciso definir valores para os limites inferior e superior do subconjunto filtrado. Filtros podem ser estatísticos, com base em valores referência absolutos ou relativos [18].

Neste trabalho, definimos estratégias de detecção para cinco dos 22 *bad smells* de Fowler [15]: *Data Class, Feature Envy, Large Class, Long Method, e Refused Bequest*. Para definir os mecanismos de filtragem e composição e propor as estratégias de detecção, as métricas de software foram escolhidas a partir de sua disponibilidade no catálogo de Filó et al. [13], Tabela 1. Os cinco *bad smells* são aqueles que podem ser avaliados com as métricas desse catálogo.

2.1 Métricas de Software Usadas nas Estratégias de Detecção

As métricas de software aplicadas na definição das estratégias de detecção propostas possuem valores referências definidos por Filó et al. e estão disponíveis no Qualitas.class Corpus [35]. São elas:

- DIT (*Depth in Inheritance Tree*): quantidade de classes que estão acima de determinada classe na hierarquia de heranças.
- LCOM (*Lack of Cohesion between Methods*): fornece uma medida da falta de coesão de uma classe.
- MLOC (*Method Lines of Code*): número de linhas de um método.
- NBD (*Nested Blocks Depth*): número máximo de blocos aninhados de um método.
- NSC (*Number of Children*): número de classes filhas em relação a uma classe.
- NOF (*Number of Fields*): quantidade de atributos de uma classe.

Tabela 1: Catálogo de valores referência para métricas de softwares orientados por objetos. Fonte: Filó et al. [13]

Métrica	Bom/Frequente	Regular/Ocasional	Ruim/Raro
CA	CA <= 7	7 < CA <= 39	CA > 39
CE	CE <= 6	6 < CE <= 16	CE > 16
MLOC	MLOC <= 10	10 < MLOC <= 30	MLOC > 30
NOC	NOC <= 11	11 < NOC <= 28	NOC > 28
NOF	NOF <= 3	3 < NOF <= 8	NOF > 8
NOM	NOM <= 8	8 < NOM <= 14	NOM > 14
NORM	NORM <= 2	2 < NORM <= 4	NORM > 4
NSC	NSC <= 1	1 < NSC <= 3	NSC > 3
NSF	NSF <= 1	1 < NSF <= 5	NSF > 5
NSM	NSM <= 1	1 < NSM <= 3	NSM > 3
PAR	PAR <= 2	2 < PAR <= 4	PAR > 4
SIX	SIX <= 0,019	0,019 < SIX <= 1,333	SIX > 1,333
VG	VG <= 2	2 < VG <= 4	VG > 4
WMC	WMC <= 11	11 < WMC <= 34	WMC > 34
DIT	DIT <= 2	2 < DIT <= 4	DIT > 4
LCOM	LCOM <= 0,167	0,167 < LCOM <= 0,725	LCOM > 0,725
NBD	NBD <= 1	1 < NBD <= 3	NBD > 3
RMD	RMD <= 0,467	0,467 < RMD <= 0,750	RMD > 0,750

- NOM (*Number of Methods*): quantidade de métodos em uma classe.
- VG (*McCabe's Complexity*): complexidade ciclomática de método.
- WMC (*Weighted Methods per Class*): somatório da complexidade dos métodos da classe. Refere-se ao peso total de uma classe em relação aos pesos de cada método da classe.
- SIX (*Specialization Index*): avalia o quanto uma subclasse sobrescreve a superclasse.

2.2 Faixas e Valores Referência Utilizados nas Estratégias de Detecção

Para filtrar o universo de entidades presentes em um sistema de software e, assim, possibilitar identificação de instâncias de *bad smell*, cada métrica de software usada por uma estratégia de detecção deve estar relacionada a um valor limiar. Esse valor, denominado valor referência, define a faixa de valores de interesse em relação a uma métrica de software específica. Assim, é possível usar tal métrica como apoio na identificação de algum *bad smell*. Por exemplo, para encontrar classes de um sistema que sejam muito grandes em termos de linhas de código, pode-se usar LOC como métrica de apoio em um filtro da estratégia de detecção. Porém, é preciso estabelecer um valor referência para LOC de forma que as classes que obdeçam a faixa de valores determinada pelo valor referência sejam identificadas corretamente. Um exemplo de valor referência poderia ser 1000, onde o filtro LOC > 1000 determinaria um limiar mínimo para que essa métrica indicasse classes grandes.

As estratégias de detecção propostas para os *bad smells* consideram três faixas de valores referência definidas em Filó et al. [13]: Bom/Frequente, Regular/Ocasional e Ruim/Raro. Bom/Frequente corresponde a valores com alta frequência, caracterizando os valores mais comuns da métrica, na prática. Ruim/Raro identifica valores com baixa frequência, e Regular/Ocasional é intermediária, corresponde a valores que não são nem muito frequentes nem raros. Para Ferreira et al. [12], valores considerados frequentes indicam que eles correspondem à prática comum no desenvolvimento de software de alta qualidade, e serve como um parâmetro de comparação de um software com os demais. Valores pouco frequentes indicam situações não usuais na prática, portanto, pontos a serem

considerados como críticos, e podem ser indicativos de problemas estruturais. Nas estratégias propostas, considera-se que: Bom: limite superior da faixa Bom/Frequente. Regular: limite inferior da faixa Regular/Ocasional. Ruim: limite inferior da faixa Ruim/Raro.

2.3 Estratégias de Detecção de *Bad Smells*

Essa seção mostra para cada estratégia uma breve descrição do *bad smell* a qual ela se refere, as métricas usadas e a estratégia em si.

Data Class. Refere-se a classes que contém somente atributos e métodos get() e set(). São classes que funcionam como um agregador de dados, geralmente sem processamento complexo. As métricas usdas para compor a estratégia de detecção desse *bad smell* são:

- `NSC`: dado que uma *Data Class* emula um tipo primitivo de dados, assume-se que esse tipo de classe tende a não ter subclasses. Espera-se que uma *Data Class* ofereça somente acesso aos seus atributos, por meio de `getters` e `setters`. Assim, considera-se que valores baixos de `NSC`, junto com outras características, possam auxiliar a identificar *Data Class*.
- `DIT`: uma classe de dados é auto-contida, provê somente acesso aos seus atributos. Não precisa herdar responsabilidade de uma superclasse. Em consonância com as considerações feitas para `NSC`, assume-se também que uma *Data Class*, em geral, não herda responsabilidades de superclasses. Assim, considera-se que um `DIT` baixo, associado a outras características, possa auxiliar a identificar *Data Class*.
- `NOF`: como a classe somente armazena dados, assume-se que uma quantidade Regular ou Ruim de atributos (NOF) indica a ocorrência de *Data Class*.

A estratégia de detecção para *Data Class* é exibida na Figura 1: `NSC` e `DIT` devem ter valor dentro do respectivo intervalo `Bom` da métrica, incluindo o limite superior desse intervalo; `NOF` deve ter valor dentro dos intervalos `Regular` ou `Ruim`, incluindo o limite inferior do intervalo `Regular` da métrica.

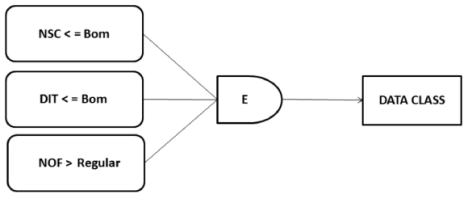

Figura 1: Estratégia proposta para detecção de *Data Class*

Feature Envy. Refere-se a um trecho de código pertencente a uma classe que está mais interessado nos dados e/ou processamentos de outra classe. A métrica usada para compor a estratégia de detecção desse *bad smell* é `LCOM`. *LCOM* é a única métrica do catálogo de Filó et al. [13] capaz de detectar, de alguma forma, o interesse de uma classe por dados de outra classe. Mesmo que indiretamente, a baixa coesão de uma classe pode indicar que ela está interessada em dados e/ou processamentos de outras classes, pelo fato de ela implementar vários interesses divergentes. Figura 2 exibe essa estratégia.

Large Class. Refere-se a classes que possuem muitas responsabilidades, processamentos e atributos. Elas tendem a centralizar a inteligência do sistema, realizando uma quantidade de trabalho excessivo e se tornando uma agregação de diferentes abstrações.

Figura 2: Estratégia proposta para detecção de *Feature Envy*

Isso está em desacordo com um dos princípios básicos de SOO, que diz que uma classe deve ter uma única responsabilidade. Algumas características desse tipo de classe é que elas são grandes, complexas e possuem baixa coesão. Figura 3 exibe a estratégia proposta para *Large Class*. As métricas usadas para compô-la são:

- `NOF`, `NOM`: número alto de atributos (`NOF`) e de métodos (`NOM`) aponta excessivos atributos e métodos da classe.
- `WMC`: considera-se que quanto maior o peso (*weight*) da classe, maior a sobrecarga de operações da mesma. `WMC` pode indicar muitas responsabilidades da classe.
- `LCOM`: falta de coesão entre os métodos da classe indica sobrecarga de responsabilidades. Portanto, `LCOM` alto sugere que a classe seja uma instância de *Large Class*.

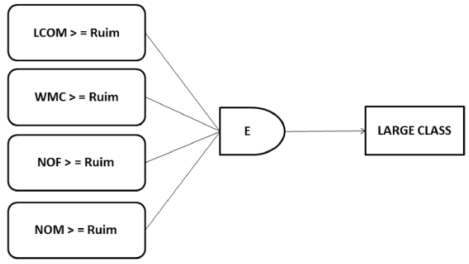

Figura 3: Estratégia proposta para detecção de *Large Class*

Long Method. Refere-se a métodos complexos, extensos e/ou com várias responsabilidades. São métodos nos quais, no processo natural da evolução de software, lhes são adicionados mais e mais funcionalidades, tornando-os de difícil compreensão e manutenção. A seguir, as métricas usadas para compor a estratégia de detecção deste *bad smell*, e a Figura 4 que exibe a estratégia proposta.

- `MLOC`: métodos longos geralmente possuem uma quantidade de linhas de código alta.
- `VG`: a complexidade ciclomática do método alta indica alta complexidade do método.
- `NBD`: um valor de aninhamento de blocos alto em um método é um indício de processamento complexo/excessivo.

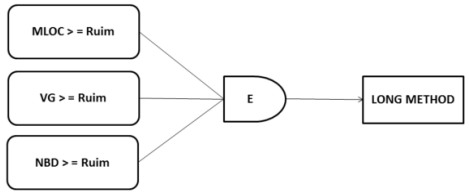

Figura 4: Estratégia proposta para detecção de *Long Method*

Refused Bequest. Refere-se a subclasses que possuem herança de métodos e dados mas que os usam pouco. A classe filha despreza os

atributos e métodos da superclasse. A métrica usada para compor a estratégia de detecção deste *bad smell* é a SIX. Nessa métrica, quanto maior o grau de sobrescrita que a classe filha possui em relação à superclasse, maiores as chances da classe filha estar recusando as responsabilidades providas por meio da herança. SIX foi usada como medida de uma característica particular e mensurável de *Refused Bequest*. Figura 5 exibe sua estratégia de detecção.

Para os demais *bad smells* de Fowler [15]: *Duplicated Code, Message Chains, Parallel Inheritance Hierarchies, Speculative Generality, Swith Statements, Temporary Field, Primitive Obsession, Inappropriate Intimacy* e *Comments* não foi possível definir estratégias visto que as métricas usadas por Filó et al. [13] não apoiam sua detecção. No caso dos *bad smells Alternative Classes with Different Interfaces* e *Incomplete Library Class*, embora as métricas usadas por Filó et al. [13] permitam a definição de estratégias de detecção, eles não foram incluídos nesse estudo. O motivo principal é que não foram encontradas ferramentas que avaliem tais *bad smells* [11].

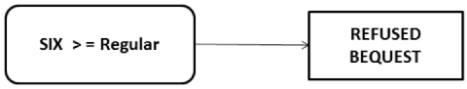

Figura 5: Estratégia proposta para detecção de *Refused Bequest*

Os outros *bad smells* de Fowler [15] não foram incluídos porque não foi possível usar as ferramentas que os detectam [11]. São eles:
1. *Lazy Class*: foram encontradas quatro ferramentas para esse *bad smell* na literatura mas estão indisponíveis para *download*. Uma delas, True Refactor, foi citada por Fernandes et al. [11] como a única ferramenta disponível para *download*, mas não foi encontrada.
2. *Middle Man*: Code Bad Smell Detector é a única ferramenta disponível para detectar esse *bad smell*. Contudo não foi possível executá-la devido a um problema no arquivo de configuração.
3. *Data Clumps*: das cinco ferramentas disponíveis para esse *bad smell*, três precisam de licença para uso: InCode, InFusion e *IntelliJ IDEA*. Code Bad Smell Detector apresentou problema no arquivo de configuração, e Stench Blossom apresentou problemas de incompatibilidade com a versão do Eclipse usada nesse estudo.
4. *Long Paramenter List*: existem nove ferramentas de detecção para esse *bad smell*. Porém, cinco estão indisponíveis para *download*. Das quatro restantes, IntelliJ IDEA precisa de licença; Gendarme não realiza detecção em sistemas Java; Checkstyle não retornou nesse estudo instâncias desse *bad smell* para os sistemas selecionados; e por fim, a PMD apresentou falhas ao tentar exportar os resultados de alguns sistemas, segundo Fernandes et al. [11].
5. *Shotgun Surgery*: há três ferramentas indisponíveis para *download*. Dentre as disponíveis, IntelliJ IDEA precisa de licença e iPlasma apresentou problemas de exportação dos resultados.

3 VALORES REFERÊNCIA E DETECÇÃO AUTOMATIZADA DE *BAD SMELLS*

A identificação de classes e métodos com problemas estruturais reflete o panorama de qualidade deteriorada do software. Esta seção descreve um estudo cujo objetivo é avaliar a capacidade de os valores referência de métricas de SOO propostos por Filó et al. [13] auxiliarem na identificação dos *bad smells* definidos por Fowler [15] e assim, garantir a qualidade dos softwares. Para isso, são investigadas duas questões de pesquisa, *QP1: os valores referência de métricas de SOO auxiliam a identificar bad smells*? *QP1.1: qual é a eficácia da detecção de bad smells utilizando-se estratégias baseadas nos valores referência e tomando-se como base os resultados gerados por ferramentas de detecção de bad smells*?

Neste estudo os resultados das estratégias propostas na Seção 2.3 foram comparados com os resultados providos pelas ferramentas de detecção de *bad smells*: JSpIRIT e JDeodorant [11, 24]. JSpIRIT [39] é um *plugin* para a Eclipse IDE que identifica *bad smells* em software Java por meio de uma abordagem baseada em métricas. JSpIRIT foi usada neste experimento para identificar a presença dos *bad smells*: *Large Class, Long Method, Data Class, Feature Envy* e *Refused Bequest*. JDeodorant [36] é um plugin de código-fonte aberto para Eclipse que identifica quatro *bad smells* em software Java: *God Class* (similar a *Large Class*), *God Method* (similar a *Long Method*), *Feature Envy* e *Switch Statement*. As técnicas de detecção aplicadas por JDeodorant baseiam-se em oportunidades de refatoração para *God Class* e *Feature Envy* e técnicas *slicing*, para detectar *God Method* [14]. Nesse experimento, a JDeodorant foi usada para detectar *Large Class, Long Method* e *Feature Envy*.

Para implementar os *scripts* com as estratégias de detecção descritas na Seção 2.3, criamos FindSmells [32], uma ferramenta de detecção de *bad smells* que permite ao usuário selecionar um ou mais arquivos em formato XML contendo medições de um software. Ela processa esses arquivos e insere as medições em um banco de dados. Após o usuário selecionar o sistema e o *bad smell* que deseja identificar, FindSmells executa a estratégia de detecção que foi implementada para aquele *bad smell*. Nesse processo, ela faz a filtragem de métodos, classes e pacotes que possuem medições anômalas de métricas de SOO no contexto do processo de medição de software. Medições anômalas são aquelas que se afastam significativamente do que é comum, podendo indicar problemas de qualidade no artefato medido [30]. Ademais, FindSmells permite a edição de estratégias de detecção já implementadas; e, permite a implementação de novas estratégias. Considerando a disponibilidade de FindSmells, os custos da solução proposta é baixo.

As ferramentas de detecção e as estratégias propostas foram executadas em 12 sistemas (vide Tabela 2) de pequeno a médio porte selecionados aleatoriamente do Qualitas.class Corpus 2013. Os resultados da JSpIRIT e JDeodorant não são tidos como corretos neste estudo. O motivo é que os resultados reportados são diferentes entre si e não há estudos prévios indicando a acurácia de tais ferramentas. Por outro lado, elas são usadas em estudos prévios a respeito de *bad smells* [11, 13]. A análise feita fornece uma visão de como os resultados das estratégias propostas se comparam aos dessas ferramentas, mas, elas não são consideradas como oráculo.

3.1 Análise com a Ferramenta JSpIRIT

Os resultados das estratégias propostas foram comparados com os resultados obtidos pela JSPiRIT em termos de Recall, Precisão e F-measure. No caso do *bad smell Data Class* os resultados foram computados com base em somente 8 dos 12 sistemas pois, para os demais sistemas, a JSpIRIT não encontrou instâncias de *Data Class*, ou houve limite de memória excedido pela ferramenta.

3.1.1 Análise de Recall. Nesse experimento, Recall mede a capacidade das estratégias de detecção encontrarem instâncias de

Tabela 2: Sistemas extraídos do Qualitas.class Corpus 2013

#	Sistemas	Tamanho	No de Classes	No de Métodos
1	aoi-2.8.1	9.4 MB	841	6915
2	checkstyle-5.6	53 MB	560	2896
3	cobertura-1.9.4.1	11 MB	174	3520
4	displaytag-1.2	10 MB	336	1728
5	itext-5.0.3	7.8 MB	587	5982
6	jedit-4.3.2	15 MB	1183	7315
7	joggplayer-1.1.4s	7.2M	363	2299
8	jsXe-04_beta	17 MB	270	1445
9	pmd-4.2.5	12 MB	914	5958
10	quartz-1.8.3	12 MB	316	2923
11	squirrel_sql-3.1.2	13 MB	169	689
12	webmail-0.7.10	12 MB	129	1091

bad smells considerando o sistema de software sob análise. Quanto maior o valor de `Recall` para uma estratégia, maior é a sua capacidade de encontrar instâncias do *bad smell* que se propõe a achar.

A Figura 6 apresenta os resultados da distribuição de `Recall` para os *bad smells*: *Large Class* (LC), *Long Method* (LM), *Data Class* (DC), *Feature Envy* (FE) e *Refused Bequest* (RB), respectivamente. Analisando o gráfico da Figura 6, nota-se que as medianas estão acima de 40% de `Recall` para 4 dos 5 *bad smells* analisados: *Large Class, Long Method, Data Class* e *Refused Bequest*. Isso aconteceu porque para 6 dos 12 sistemas analisados nesse experimento, o `Recall` foi maior ou igual a 40% para cada *bad smell*. Este resultado é razoável se for considerado o contexto do estudo: detecção de *bad smells* em código-fonte, pois essa não é uma tarefa trivial. A literatura reporta que, mesmo em sistemas de pequeno-porte, várias instâncias de *bad smells* podem ser encontradas [20]. Assim, um `Recall` de ao menos 40% significa que, em um universo extenso de possíveis instâncias, as estratégias de detecção foram capazes de encontrar uma quantidade significativa delas.

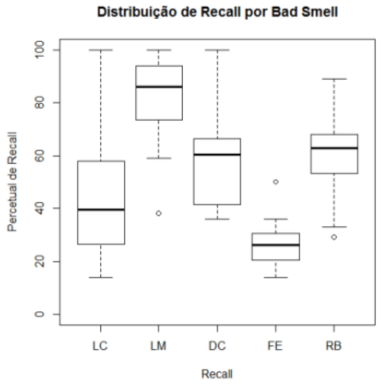

Figura 6: `JSpIRIT` **- distribuição de** `Recall` **por** *bad smell*

Observa-se também que a distribuição de `Recall` para *Feature Envy* tem mediana de 26%, e que o terceiro quartil (75%) não está muito distante disso, com valor próximo de 30%. Uma justificativa para isso é a carência de métricas no catálogo de Filó et al. [13] adequadas à detecção desse *bad smell*. Por exemplo, usou-se somente LCOM na estratégia proposta, por falta de métricas que meçam acoplamento entre classes. Outro dado importante é que foi obtido `Recall` de 100%, para pelo menos um sistema, considerando os *bad smells*: *Large Class, Long Method* e *Data Class*. *Refused Bequest*, obteve `Recall` de 90% para pelo menos um dos sistemas analisados.

3.1.2 Análise de `Precisão`. Diferentemente do `Recall`, a `Precisão` mede a capacidade das estratégias de detecção propostas encontrarem instâncias verdadeiras de *bad smells*. Enquanto o `Recall` mede a quantidade de resultados, a `Precisão` mede a qualidade dos resultados. Instâncias verdadeiras neste estudo referem-se àquelas identificadas por `JSpIRIT`. Todavia, conforme descrito, não se tem como premissa, nesse estudo, que os dados reportados pelas ferramentas são de fato verdadeiros. Esses resultados são usados somente para fins de comparação com as estratégias propostas.

A Figura 7 apresenta os resultados da distribuição de `Precisão` por *bad smell*. São eles: *Large Class* (LC), *Long Method* (LM), *Data Class* (DC), *Feature Envy* (FE) e *Refused Bequest* (RB). Observando-se a Figura 7, percebe-se que as medianas de percentual de `Precisão` estão acima de 40% para os *bad smells*: *Large Class* e *Feature Envy*. Ao verificar os dados dos sistemas para tais *bad smells* nota-se que 6 dos 12 possuem `Precisão` maior ou igual a 40%. E, para *Long Method*, a `Precisão` é pouco maior que 30%.

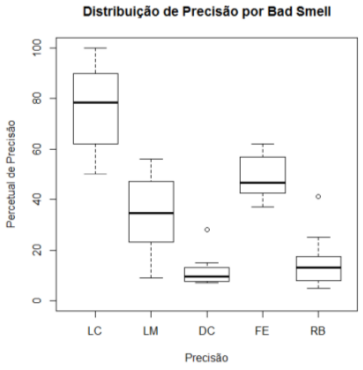

Figura 7: `JSpIRIT` **- distribuição de** `Precisão` **por** *Bad Smell*

3.1.3 Análise de `F-measure`. A `F-measure` foi usada para verificar a exatidão dos resultados gerados. Ela considera os resultados obtidos com as fórmulas de `Recall` e `Precisão`, podendo ser interpretada como uma média ponderada entre essas duas medidas.

Figura 8 exibe os resultados da distribuição de `F-measure` por *bad smell*. São eles: *Large Class* (LC), *Long Method* (LM), *Data Class* (DC), *Feature Envy* (FE) e *Refused Bequest* (RB). Pode-se observar nessa figura, que somente para os *bad smells*, *Large Class* e *Long Method*, a mediana para `F-measure` está próxima de 50%. Isso significa que, balanceando-se as medidas de `Recall` e `Precisão`, tem-se uma acurácia significativa e portanto, as estratégias de detecção propostas para esses *bad smells* se mostraram compatíveis com os resultados da `JSpIRIT`. Nota-se que o valor para a mediana no caso de *Feature Envy* não foi tão abaixo dos valores para *Large Class* e *Long Method*. *Data Class* e *Refused Bequest* tiveram resultados menores ainda tendo como base o conjunto de sistemas selecionados.

Pela Figura 8 foram alcançados coeficientes de até 78% e 72% para *Large Class* e *Long Method*, respectivamente. Ao balancear `Recall` e `Precisão`, os resultados mantiveram-se expressivos para esses *bad smells*. Para os demais *bad smells*, observa-se que *F-measure* foi baixo. Nesses casos, o `Recall` foi, em geral, baixo, mas a `Precisão` foi

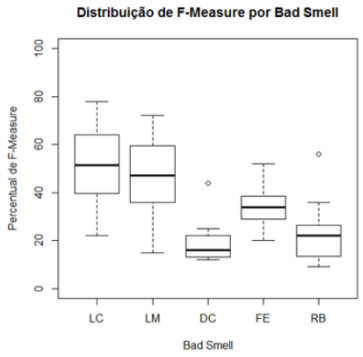

Figura 8: JSpIRIT: distribuição de F-mesure por *Bad Smell*

significativa. Isso indica que os resultados gerados pelas estratégias propostas para esses *bad smells* tendem a divergir em relação aos resultados da JSpIRIT. Ressalta-se que a análise *F-measure* pode ter sido impactada pelo fato de que somente 8 dos 12 sistemas tiveram instâncias de *Data Class* encontradas pela JSpIRIT.

3.1.4 Inspeção Manual dos Resultados da Ferramenta JSpIRIT. Trabalhos anteriores reportam ineficiências em ferramentas para detecção automatizada de *bad smells* em código-fonte [6, 11, 17]. Isso pode ser causado, entre outros fatores, pela forma como os valores referência e métricas usadas pelas ferramentas na composição de estratégias de detecção são definidos. Essas definições podem variar de acordo com o contexto dos sistemas analisados, e isso pode impactar na qualidade dos resultados.

A fim de minimizar uma das principais ameaças desse experimento, o uso dos resultados da JSpIRIT como lista de referência de *bad smells*, foi feita uma validação dos resultados por um especialista que possui conhecimentos em programação OO e que conhece as definições dos *bad smells* de Fowler [15]. O especialista fez a validação dos resultados providos pela JSpIRIT em 2 dos 12 sistemas de software avaliados nesse experimento: Squirrel SQL e Webmail. Esses sistemas foram escolhidos por serem os menores dentre o conjunto analisado, para assim, viabilizar a análise manual das instâncias de *bad smells* reportadas por JSpIRIT, com base no código-fonte dos sistema, sob o ponto vista do especialista.

A Tabela 3 exibe os resultados obtidos por meio da análise com validação do especialista. Para fins de comparação, são exibidos os valores de Recall e Precisão referentes aos dados não-validados (NV) e aos dados após validação feita pelo especialista (V). A tabela também exibe a diferença (DIF) entre os percentuais de Recall e Precisão validados e não-validados. O *bad smell Data Class* foi descartado dessa análise, porque a JSpIRIT não retornou instâncias desse *bad smell* para os dois sistemas selecionados para avaliação. Além disso, JSpIRIT não retornou instâncias de *Refused Bequest* para o sistema Squirrel SQL. Porém, considerando que JSpIRIT reportou resultados para o outro sistema avaliado, Webmail, esse *bad smell* não foi descartado da análise.

Nota-se que a *DIF* apresentou valores positivos para Recall no caso dos *bad smells*: *Large Class, Feature Envy* e *Refused Bequest*. Isso significa que a validação do especialista identificou falsos positivos nos resultados gerados por JSpIRIT. A remoção dessas instâncias

Tabela 3: Comparação dos Resultados do Especialista com as Estratégias de Detecção

Sistemas		Large Class			Long Method			Feature Envy			Refused Bequest		
		NV	V	DIF	NV	V	DIF	NV	V	DIF	NV	V	DIF
Squire	Recall	100%	100%	0%	100%	100%	0%	50%	60%	10%	-	-	-
	Precision	60%	40%	-20%	17%	17%	0%	55%	30%	-25%	-	-	-
Webmail	Recall	15%	50%	35%	100%	100%	0%	27%	33%	6%	57%	80%	23%
	Precision	100%	50%	-50%	56%	41%	-15%	47%	24%	-23%	14%	14%	0%

identificadas pelo especialista gerou aumento do Recall. Além disso, um *DIF* negativo também foi obtido para Precisão no caso dos *bad smells*: *Large Class, Long Method* e *Feature Envy*. Nesse caso, conclui-se que, em algumas situações, o especialista não identificou instâncias verdadeiras nos resultados gerados pela JSpIRIT. Esse fato impactou negativamente no resultado da Precisão. Considerando a Tabela 3 e os *bad smells Large Class* e *Feature Envy*, observa-se também um aumento nos percentuais de Recall acompanhado por uma diminuição de Precisão.

A Figura 9 apresenta a distribuição de Recall para todos os *bad smells* cujas instâncias foram validadas pelo especialista em relação aos dois sistemas avaliados. Observa-se que foi obtida uma mediana de 80% para Recall. Além disso, nota-se que 3/4 dos resultados de Recall são maiores ou iguais a 55%. Tais resultados são bastante expressivos considerando as dificuldades inerentes à detecção de *bad smells* em sistemas de software. Mesmo em sistemas pequenos, pode haver uma quantidade elevada de instâncias de *bad smells* e, consequentemente, ter uma taxa alta de recuperações de instâncias é desejável. Em geral, a concentração de valores de Recall foi alta. Conclui-se que a participação do especialista foi positiva no sentido de que ele ajudou a melhorar a qualidade das listas de referência providas pela ferramenta JSpIRIT.

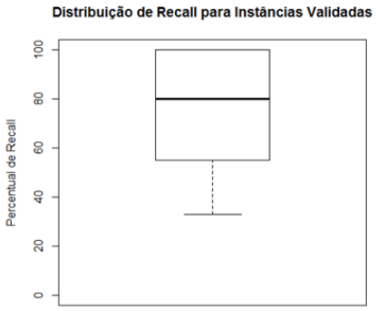

Figura 9: Distribuição de Recall para instâncias avaliadas pelo especialista

Figura 10 apresenta a distribuição de Precisão para todos os *bad smells* cujas instâncias foram validadas pelo especialista, em relação aos dois sistemas avaliados.

Contrário as observações feitas em relação a Recall, a Figura 10 indica uma mediana de Precisão de 30%, um valor baixo porém esperado se observada a concentração alta de Recall exibida na Figura 9. Nota-se que o maior valor de Precisão é 50%. Isso indica que, no melhor caso, dados os resultados de uma estratégia de detecção, metade dos resultados providos foram válidos. Esses resultados sugerem ou que o especialista, por vezes, não foi capaz de identificar

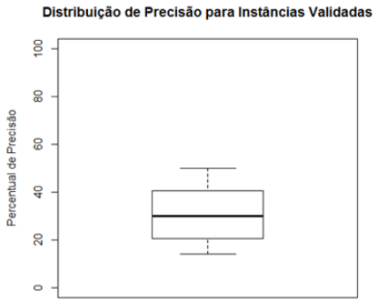

Figura 10: Distribuição de `Precisão` para instâncias avaliadas pelo especialista

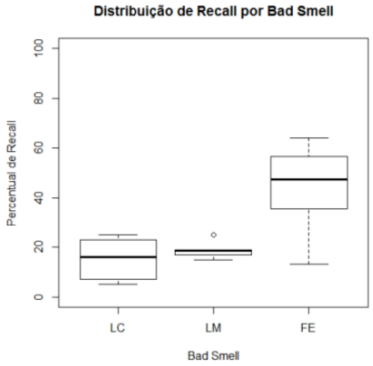

Figura 11: `JDeodorant`: distribuição de `Recall` por *Bad Smell*

manualmente algumas instâncias válidas de *bad smells* ou que as estratégias indicam mais instâncias de *bad smells* do que realmente existe. Avalia-se, aqui, que detectar *bad smells* manualmente pode ser uma tarefa complexa, e determinar se um trecho de código é anômalo é subjetivo. Por exemplo, o especialista reportou que a detecção de *Feature Envy* demandou uma examinação minuciosa do código-fonte, em relação a chamadas de métodos, manipulação de atributos e outros aspectos dos sistemas analisados.

3.2 Análise com a Ferramenta `JDeodorant`

Os resultados das estratégias propostas na Seção 2.3 também foram comparados com os resultados obtidos pela `JDeodorant` em termos de `Recall`, `Precisão` e `F-measure`.

3.2.1 Análise de `Recall`. Figura 11 exibe a distribuição de `Recall` por *bad smell*, considerando os resultados de detecção providos por `JDeodorant` e pelas estratégias de detecção apresentadas nesse estudo. É importante relembrar que `JDeodorant` não se propõe a identificar *Data Class* e *Refused Bequest*. Portanto, esses *bad smells* foram descartados nessa análise. Em relação aos *bad smells Large Class* e *Long Method*, observa-se uma concentração baixa dos valores de `Recall` para as estratégias de detecção propostas em relação à `JDeodorant`. Porém, ao verificar as listas de referência observou-se que a quantidade de resultados de detecção reportados por `JDeodorant` é, em geral, muito maior do que a quantidade identificada pelas estratégias. Por exemplo, tomando como base o sistema `Checkstyle`, `JDeodorant` identificou: (i) 81 instâncias de *Large Class*, contra 4 instâncias reportadas pela respectiva estratégia de detecção proposta, isto é, 95% mais instâncias; e (ii) 110 instâncias de *Long Method*, contra 40 instâncias reportadas pela estratégia de detecção desse *bad smell*, ou seja, 60% mais instâncias. Além disso, para o sistema `JsXe`: (i) 29 instâncias de *Large Class*, contra 4 instâncias reportadas pela respectiva estratégia de detecção, isto é, 86% mais instâncias; e (ii) 126 instâncias de *Long Method*, contra 52 instâncias reportadas pela estratégia de detecção desse *bad smell*, ou seja, 59% mais instâncias.

Em contrapartida, a concentração dos valores de `Recall` para *Feature Envy* são, em geral, moderados, com mediana próxima a 50%. O maior valor de `Recall` foi em torno de 65%. Além disso, o primeiro quartil está próximo de 40%, o que significa que, para 3 dos sistemas avaliados, o `Recall` foi significativo, menor ou igual 4 a 40%. Note-se que, ao contrário do que foi observado na análise com `JSpIRIT`, o `Recall` para esse *bad smell* foi, de certa forma, significativo. A `JDeodorant` possui uma abordagem híbrida de detecção de *bad smells*. Segundo Fernandes et al. [11], a detecção de *bad smells* implementada por `JDeodorant` é baseada em métricas de software e *Abstract Syntax Tree*. Tal abordagem híbrida pode ter provido resultados de detecção mais alinhados aos resultados reportados pela estratégia proposta quando comparado à `JSpIRIT`.

3.2.2 Análise de `Precisão`. Figura 12 exibe a distribuição de `Precisão` por *bad smell*, dado os resultados de detecção providos por `JDeodorant` e pelas estratégias de detecção. Para *Large Class* e *Long Method*, obteve-se uma significativa distribuição dos valores de `Precisão` para as estratégias propostas, em relação à `JDeodorant`. As medianas estão em torno de 50% e 90%, respectivamente. Para *Large Class*, 50% dos sistemas avaliados possuem mais de 90% de `Precisão`, chegando a 100%. Para *Long Method*, observa-se uma coerência entre os valores obtidos, que estão entre 40% e 70%, ou seja, a `Precisão` mostrou pouca discrepância de valores entre os sistemas avaliados, no caso desse *bad smell*. Observa-se que, embora `Recall` tenha sido baixo para ambos os *bad smells*, a `Precisão` é grande. Embora `JDeodorant` tenha retornado uma quantidade baixa de instâncias em relação às estratégias, o resultado foi, em geral, preciso.

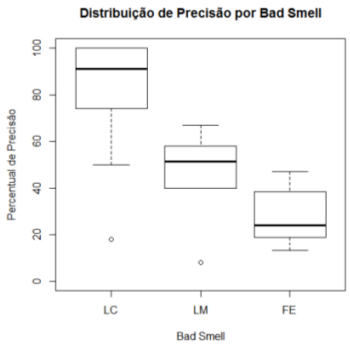

Figura 12: `JDeodorant`: distribuição de `Precisão` por *Bad Smell*

Para *Feature Envy*, observa-se uma distribuição baixa dos valores de `Precisão`, com mediana próxima a 20%. Isso vai de encontro aos resultados significativos de `Recall`. Uma possível justificativa

é que, quanto maior a quantidade de resultados recuperados por uma estratégia de detecção, maiores as chances de ocorrência de falsos positivos, conforme visto no experimento com JSpIRIT.

3.2.3 Análise de F-measure. Figura 13 exibe a distribuição de F-measure obtida por *bad smell*. Nela, são mostrados os resultados de distribuição para *Large Class* (LC), *Long Method* (LM) e *Feature Envy* (FE). Observando a Figura 13, tem-se que nenhum dos *bad smells* analisados por JDeodorant obteve F-measure maior que 50%. Ademais, as medianas obtidas não ultrapassaram 30%. Isso significa que, balanceando-se Recall e Precisão, obteve-se uma acurácia baixa. Assim, para nenhum dos *bad smells* avaliados: *Large Class, Long Method* e *Feature Envy*, as estratégias de detecção propostas identificaram resultados próximos aos da JDeodorant.

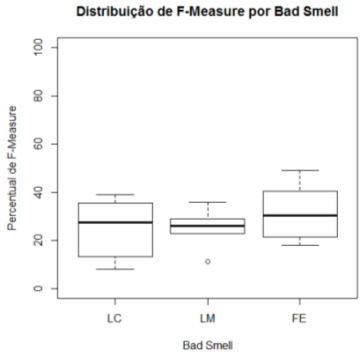

Figura 13: JDeodorant: **distribuição** F-measure **por** *Bad Smell*

Um dos fatores que pode justificar a discordância entre os resultados providos por JDeodorant e pelas estratégias de detecção é a abordagem híbrida usada por JDeodorant para detecção de *bad smells*. Outro fator que pode ter impactado a análise é a diferença entre a quantidade de resultados retornados por JDeodorant em relação às estratégias. Para o sistema JEdit, JDeodorant identificou 172 instâncias de *Large Class*, enquanto que a estratégia proposta retornou 34 instâncias, uma diferença de aproximadamente 80%. Para o sistema Squirrel SQL, obteve-se 21 instâncias de *Large Class* providas pela JDeodorant enquanto apenas 5 instâncias foram identificadas pela estratégia de detecção. Quarenta instâncias de *Long Method* foram retornadas por JDeodorant, ao passo que somente 12 instâncias foram providas pela estratégia.

4 AMEAÇAS À VALIDADE

Discutem-se os quatro tipos de ameaça à validade desse experimento: ameaças de construção, ameaças internas, ameaças externas e ameaças de conclusão, segundo as diretrizes de Wohlin et al. [40].

4.4.1 Ameaças de construção. Para esse experimento foram escolhidos de forma não-sistemática sistemas do Qualitas.class Corpus 2013. Isso pode ter impactado negativamente no cálculo de Recall, Precisão e, consequentemente, F-measure, pois quanto maior o tamanho da amostra, maior tende a ser sua representatividade e diversidade. Contudo, a escolha foi baseada na capacidade de analisar tais sistemas via JSpIRIT e JDeodorant, que mostraram-se não-escaláveis para sistemas acima de 60MB e, na disponibilidade do código-fonte dos sistemas para *download*. As estratégias de detecção desse estudo foram propostas de acordo com as métricas de software disponíveis no catálogo de Filó et al. [13] e no Qualitas.class Corpus 2013. Assim, a carência de métricas pode ter impactado negativamente nos resultados obtidos, uma vez que as estratégias podem não ter considerado alguns aspectos que caracterizam cada tipo de *bad smell*. A fim de amenizar esse problema, selecionou-se cuidadosamente cada métrica para composição das estratégias, buscando-se usar métricas apropriadas. O catálogo de valores referência selecionado também impacta diretamente nos resultados, pois tais valores são usados para composição das estratégias. Para mitigar tal impacto, as faixas de valores referência foram definidas cuidadosamente de acordo com as características de cada *bad smell*.

O estudo comparou os resultados das estratégias propostas com os resultados reportados pelas ferramentas, JDeodorant e JSPiRIT. Não foi realizada uma análise no presente estudo acerca da eficácia dessas ferramentas, de forma que seus resultados não foram tomados como corretos. Entretanto, a comparação realizada coloca em perspectiva os resultados das estratégias propostas em relação a duas das principais ferramentas disponíveis.

Em um estudo comparativo entre ferramentas de *bad smells*, Fernandes et al. [11] identificaram 84 ferramentas. Eles aplicaram os seguintes critérios para comparar as ferramentas empiricamente: ferramentas livres e disponíveis que analisam softwares em Java; quantidade de *bad smells* analisados; ferramentas que não apresentaram erros, dificuldades ou problemas na execução. Como resultado, eles encontraram quatro ferramentas: inFusion, JDeodorant, PMD e JSpIRIT. Os resultados deles indicam que inFusion e PMD apresentam maior Recall, 50% e 67%, respectivamente. JSpIRIT e PMD apresentam maior Precisão, 80% e 100%, respectivamente. JDeodorant apresentou Precisão de 17% e Recall de 33%. Entretanto, eles consideraram apenas dois *bad smells* nessa avaliação, *Large Class* e *Long Method*, e as análises foram realizadas com apenas um software, de pequeno porte. Eles também avaliaram as ferramentas quanto à facilidade de uso. Nessa análise, JDeodorant e inFusion foram as que apresentaram os melhores resultados. No presente trabalho, os estudos comparativos consideraram JSpIRIT e JDeodorant. Conforme os resultados apontados por Fernandes et al. [11], JSpIRIT tem eficácia maior do que JDeodorant na identificação de *bad smells*: JSpIRIT apresentou maior Recall e a segunda melhor Precisão e JDeodorant apresentou o maior Recall, tendo sido a ferramenta que mais identificou *bad smells* no software analisado. Considerando-se os três *bad smells* comuns analisados por essas ferramentas, as estratégias propostas no presente trabalho se comportaram da seguinte forma: quando comparadas com JSpIRIT, apresentaram Recall médio de 50% e Precisão média de 53%, observando-se as medianas; quando comparadas a JDeodorant, apresentaram Recall e Precisão médios de 44% e 57%.

4.4.2 Ameaças internas. Parte da coleta dos dados para análise possui algumas ameaças relacionadas a fatores humanos, pois os resultados providos por JSpIRIT e JDeodorant foram transferidos manualmente para planilhas para viabilizar o cálculo de Recall, Precisão e *F-measure*. Ademais, a aplicação das estratégias propostas foram feitas por meio da ferramenta, FindSmells, que pode conter erros. A fim de mitigar esse problema, tanto o código-fonte de FindSmells quanto as planilhas para cálculo de Recall, Precisão e F-measure foram validados por um especialista em SOO. Também foi feita uma verificação manual dos resultados para identificar a interseção entre os resultados obtidos por JSpIRIT e JDeodorant

com os resultados identificados pelas estratégias de detecção propostas. Esses resultados foram conferidos por um especialista como forma de tratamento para essa ameaça.

Para a seleção dos 12 sistemas avaliados neste estudo, foi necessário importar o projeto de código-fonte de cada sistema na ferramenta de desenvolvimento `Eclipse IDE`, pois tanto a `JSpIRIT` quanto a `JDeodorant` são plugins do `Eclipse` e por isso requerem que os sistemas de entrada sejam importados para identificação de *bad smells*. Porém, nem sempre foi possível importar os sistemas, como por exemplo, nos casos de `ANTLR, ArgoUML` e `Tomcat` que apresentaram erro. Eventualmente, a falta de uma configuração de importação adequada pode ter ocasionado no descarte de sistemas que poderiam ter sido usados no estudo. Como tratamento dessa ameaça, vários sistemas foram submetidos para importação a fim de obter-se uma quantidade suficiente de sistemas para análise.

4.4.3 Ameaças externas. Baseado em trabalhos anteriores que investigam *bad smells* em código-fonte [6, 11, 13], foram escolhidas as métricas `Recall`, `Precisão` e `F-measure` para análise da efetividade dos valores referência usados por meio das estratégias de detecção. Ainda que outras medições possam ser interessantes neste contexto de estudo, as medições escolhidas mostraram-se eficazes para apoiar a discussão apresentada neste artigo. Além disso, para validação da análise feita, contou-se com um especialista em SOO e com conhecimento sobre os *bad smells* de Fowler [15].

4.4.4 Ameaças de conclusão. Neste estudo foi avaliado um conjunto de 12 sistemas de software distintos do Qualitas.class Corpus 2013. Dos 111 sistemas disponíveis no Corpus, tentou-se importar 58 sistemas, porém 46 foram descartados pelos seguintes motivos: (i) problemas na importação do código-fonte na `Eclipse IDE`; ou (ii) limite excedido de memória pela `Eclipse IDE`; ou (iii) existência de mais de um arquivo `XML` com métricas de software para o mesmo sistema. Por questões de escalabilidade de `JSpIRIT` e `JDeodorant`, o maior sistema avaliado, em termos de espaço em disco, possui no máximo 60 MB, 1 KNOC (classes) e 7 KNOM (métodos). Ainda que, sob o ponto de vista dos autores desse artigo, os 12 sistemas escolhidos foram suficientes para a análise proposta nesse estudo, esse conjunto de sistemas pode não refletir a realidade de desenvolvimento no contexto de sistemas de grande porte. Além disso, todos os sistemas disponibilizados no Qualitas.class Corpus 2013 são *open-source* e implementados na linguagem de programação Java. Isso também pode dificultar a generalização dos resultados obtidos para outros contextos de desenvolvimento, como por exemplo, para outras linguagens de programação.

Devido a limitações de tecnologia, como é o caso dos problemas de memória excedida que aconteceram ao tentar executar sistemas de grande porte, a análise de mais sistemas de software foi inviável. Como tratamento dessa ameaça, buscou-se selecionar sistemas conhecidos pela comunidade acadêmica, investigados na literatura e com quantidades significativas de instâncias de *bad smells*, como por exemplo, `AOI, JEdit` e `Webmail` [9, 27].

5 TRABALHOS RELACIONADOS

Esta seção mostra como pesquisadores tem investigado a aplicação de valores referência de métricas de SOO à detecção de *bad smells*. Nessas pesquisas, percebe-se várias abordagens, e quase todas propõem um conjunto de regras que são combinações de métricas de softwares com os seus valores referência, para a detecção de *bad smells* em um sistema. Há abordagens que usam a derivação de valores referência de métricas de software para identificar *bad smells* usando análise de riscos. Alguns estudos propõem a relação de que os valores referência derivados para detectar *bad smell* podem ser verificados por meio de predição de classes defeituosas. Outros trabalhos propõem a detecção de *bad smell* a partir de diagramas UML e diagramas de classes e a detecção de *bad smells* por meio de métricas de interesse [5, 12, 13, 16, 18, 21–23, 31, 33, 37].

Vale et al. [37] comparam três métodos para derivar valores referência para 33 Software Product Lines: Alves et al. [5], Ferreira et al. [12] e Oliveira et.al [22]. Os autores avaliam quais desses métodos derivam valores referência apropriados para quatro métricas usadas em linha de produto de software: *Lines of Code (LoC), Coupling between Objects classes (CBO), Weighted Method per Class (WMC),* e *Number of Constant Refinements (NCR).* Os valores referência foram usados para a identificação de *God Class*.

Sahin et al. [26] afirmam que *bad smells* são detectados, de um modo geral, via métricas de qualidade que representam alguns sintomas dos *bad smells*. No entanto, a seleção de métricas de qualidade é um desafio devido à ausência de consenso acerca da estratégia de identificação de alguns *bad smells* baseados em um conjunto de sintomas, e também devido ao elevado esforço em determinar manualmente o valor referência para cada métrica.

Singh and Kahlon [29] aborda esse problema usando análise de risco em cinco níveis diferentes. Três versões do Mozilla Firefox são usadas como conjunto de dados no estudo. Os autores se baseiam na ideia de que a propensão de um módulo ao erro está de alguma forma associada com um projeto ruim. Os valores referência derivados foram avaliados via identificação de classes defeituosas para o Firefox. Os resultados foram positivos, mas para validar a pesquisa, segundo os autores, mais experimentos precisam ser feitos.

Lanza and Marinescu [18] realizaram um dos primeiros trabalhos para detecção de *bad smells* via métricas de software. Eles usam dois meios estatísticos para encontrar valores referência. A média para determinar o valor referência mais típico do conjunto de dados e o desvio padrão para obter uma medida de quanto os valores referência nos dados são espalhados. Usando limiares baseados em estatísticas, os autores mediram 45 sistemas Java e 37 projetos em C++ em relação a três métricas: *Number of Methods* (`NOM`), *Lines of Code* (`LOC`) e *Cyclomatic Number* (`CYCLO`). Eles aplicam valores referência na proposta de estratégia de detecção de *bad smells*. Todavia, a derivação dos valores referência são baseados em médias e, em geral, as métricas de software não são modeladas por distribuições em que a média é representativa. Os valores referência aplicados no presente trabalho foram derivados considerando-se as características dessas distribuições.

Macia et al. [19] descrevem um estudo empírico sobre o impacto das anomalias de código na degradação do projeto de arquitetura. As métricas necessárias pelas estratégias de detecção são coletadas com as ferramentas: Together [3], MuLATo [2] e Understand [4]. Os resultados mostram que a maioria dos problemas de arquitetura no código fonte emerge de um código anômalo. Esses resultados sugerem que a remoção sistemática de anomalias de código pode ser usada para combater eficazmente sintomas de degradação arquitetônica. Segundo os autores, também revelou como certos tipos de anomalias precoce de código causam impacto adverso no projeto arquitetônico à medida que os sistemas evoluem.

6 CONCLUSÃO

Este artigo apresentou as estratégias de detecção para cinco *bad smells: Large Class, Long Method, Data Class, Feature Envy* e *Refused Bequest*, propostos por Fowler [15], com base nos valores referência do catálogo de Filó et al. [13]. Esse catálogo foi usado na condução desta pesquisa por ter a maior quantidade de métricas e ter sido parcialmente avaliado. Este artigo mostrou também um estudo de caso que investigou a eficácia dos valores referência de métricas de software na identificação de *bad smells* em código-fonte.

Com os resultados deste estudo, as questões de pesquisa investigadas são respondidas. *QP1: os valores referência de métricas de SOO auxiliam a identificar bad smells?* e *QP1.1: qual é a eficácia da detecção de bad smells utilizando-se estratégias baseadas nos valores referência e tomando-se como base os resultados gerados por ferramentas de detecção de bad smells*? A resposta de *QP1* é afirmativa. Os resultados do estudo feito mostram que os valores referência foram significativamente eficazes no apoio à detecção de *bad smells*, com percentuais em geral altos para `Recall` e moderados para `Precisão` e `F-measure`. Esse resultado positivo foi obtido tanto em relação a listas de referência providas pelas ferramentas de detecção de *bad smells*, `JSpIRIT` e `JDeodorant`, quanto em relação às listas geradas manualmente pelo especialista. A conclusão principal que esse estudo fornece é que os resultados das estratégias de detecção não são discrepantes em relação as ferramentas disponíveis para detecção de *bad smell*. Além disso, observa-se também que os resultados das estratégias de detecção são mais próximos dos resultados da `JSpIRIT` do que dos da `JDeodorant`.

Como sugestões de trabalhos futuros é importante que sejam realizadas: (i) avaliações das estratégias em outros sistemas; (ii) avaliação manual de cada um dos 12 sistemas considerados neste trabalho; (iii) avaliações adicionais do catálogo de Filó et. al [13] referentes a outras questões relacionadas a qualidade de software, como padrões de projeto [31], projeto arquitetural [19], dentre outros; (iv) extensão das análises feitas por Filó et.al [13] neste trabalho à sistemas implementados em outras linguagens de programação, como C#, JavaScript e PHP; (v) replicação dos estudos apresentados neste artigo para avaliação de outros catálogos de valores referência para se avaliar a aplicabilidade desses catálogos.

REFERÊNCIAS

[1] [n. d.]. ([n. d.]).
[2] [n. d.]. MuLATo tool, http://sourceforge.net/projects/mulato/(2009).
[3] [n. d.]. Together: http://www.borland.com/us/products/together/.
[4] [n. d.]. Understand: http://www.scitools.com/.
[5] T. L. Alves, C. Ypma, and J. Visser. 2010. Deriving metric thresholds from benchmark data. In *International Conference on Software Maintenance*. IEEE, 10.
[6] S. Bellon, R. Koschke, G. A., J. K., and E. M. 2007. Comparison and evaluation of clone detection tools. *Transactions on Software Engineering* 33, 9 (2007), 577–591.
[7] Saida Benlarbi, Khaled El Emam, Nishith Goel, and Shesh Rai. 2000. Thresholds for Object-Oriented Measures. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*. IEEE Computer Society, 24–38.
[8] I.M. Bertrán. 2009. *Avaliação da qualidade de software com base em modelos uml*. Ph.D. Dissertation. RJ - Brasil.
[9] B. Cardoso and E. Figueiredo. 2015. Co-Occurrence of Design Patterns and Bad Smells in Software Systems: An Exploratory Study. In *Proceedings of the Annual Conference on Brazilian Symposium on Information Systems*, Vol. 46. 347–354.
[10] C. Couto, C. Maffort, R. Garcia, and M. T. Valente. 2013. COMETS: A Dataset for Empirical Research on Software Evolution Using Source Code Metrics and Time Series Analysis. *ACM SIGSOFT Software Engineering Notes* 38, 1 (2013), 1–3.
[11] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 18.
[12] K. A. M. Ferreira, Mariza A.S. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida. 2012. Identifying Thresholds for Object-oriented Software Metrics. *Journal of Systems and Software* 85 (2012), 244–257.
[13] T. G. S. Filó, M. A. S. Bigonha, and K. A. M. Ferreira. 2015. A Catalogue of Thresholds for Object-Oriented Software Metrics. In *Proceedings of International Conference on Advances and Trends in Software Engineering (SOFTENG)*. 48–55.
[14] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita. 2015. Automatic metric thresholds derivation for code smell detection. In *Proceedings of the Sixth International Workshop on Emerging Trends in Software Metrics*. IEEE Press, 44–53.
[15] M. Fowler. 1999. *Refactoring:improving the design of existing code*. Pearson Ed.
[16] S. Kaur, S. Singh, and H. Kaur. 2013. A quantitative investigation of software metrics threshold values at acceptable risk level. *International Journal of Engineering Research and Technology* 2 (2013).
[17] H. L., Z. M, W. S., and Z. N. 2012. Schedule of bad smell detection and resolution:a new way to save effort. *Transactions on Software Engineering* 38, 1 (2012).
[18] M. Lanza and R. Marinescu. 2010. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems (1st ed.)*. Springer Publishing Company, Incorporated.
[19] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. 2012. On the Relevance of Code Anomalies for Identifying Architectural Degradation Symptoms. In *Proceedings of the 16th Europe Conference on Software Maintenance and Reengineering*. IEEE, 277–286.
[20] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa. 2012. Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM, 167–178.
[21] R. Marinescu. 2004. Detection strategies:metrics-based rules for detecting design flaws. In *Software Maintenance. 20th International Conference*. IEEE, 350–359.
[22] P. Oliveira, M. T. Valente, and F. P. Lima. 2014. Extracting relative thresholds for source code metrics. In *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 254–263.
[23] J. Padilha, E. Figueiredo, C. Sant'Anna, and A. Garcia. 2013. Detecting God Methods with Concern Metrics: An Exploratory Study. In *Proceedings of the 7th Latin-American Workshop on Aspect-Oriented Software Development(LA-WASP)*.
[24] T. Paiva, A. Damasceno, J. Padilha, E. Figueiredo, and C. Santanna. 2015. Experimental evaluation of code smell detection tools. In *III Workshop on Software Visualization, Evolution, and Maintenance (VEM)*.
[25] M. Riaz, E. Mendes, and E. Tempero. 2009. A Systematic Review of Software Maintainability Prediction and Metrics. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 367–377.
[26] D. Sahin, M. K., S. Bechikh, and K. Deb. 2014. Code-smell detection as a bilevel problem. *Transactions on Software Engineering and Methodology* 24, 1 (2014), 6.
[27] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente. 2013. Recommending move method refactorings using dependency sets. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 232–241.
[28] R. Shatnawi, W. Li, J. Swain, and T. Newman. 2010. Finding Software Metrics Threshold Values Using ROC Curves. *Journal of software maintenance and evolution: Research and practice* 22, 1 (2010), 1–16.
[29] S. Singh and K. Kahlon. 2014. Object oriented software metrics threshold values at quantitative acceptable risk level. *CSI transactions on ICT* 2, 3 (2014), 191–205.
[30] I. Sommervile. 2011. *Engenharia de Software*. Pearson Education Brazil.
[31] Bruno L. Sousa, Mariza A.S. Bigonha, and Kecia Ferreira. 2017. Evaluating Co-Ocorrence of GOF Design Patterns with God Class and Long Method Bad Smells. In *XIII Brazilian Symposium on Information Systems (SBSI)*. 396.
[32] Bruno L. Sousa, P.P. Souza, E. Fernandes, K.A.M. Ferreira, and M.A.S. Bigonha. 2017. FindSmells:Flexible Composition of Bad Smell Detection Strategies. In *25th International Conference on Program Comprehension (ICPC)*. 360–363.
[33] Priscila Souza. 2016. *A Utilidade dos Valores Referência de Métricas na Avaliação da Qualidade de Softwares Orientados por Objeto*. Master's thesis. DCC-UFMG.
[34] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Asia Pacific Software Engineering Conference*. 336–345.
[35] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha. 2013. Qualitas. class corpus: A compiled version of the qualitas corpus. 38, 5 (2013), 1–4.
[36] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. 2008. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference*. IEEE, 329–331.
[37] G. Vale, D. Albuquerque, E. Figueiredo, and A. Garcia. 2015. Defining metric thresholds for software product lines: a comparative study. In *Proceedings of the 19th International Conference on Software Product Line*. ACM, 176–185.
[38] S. Vidal, Vasquez H., A. Díaz-Pace, and W. Oizumi. 2015. JSpIRIT: a flexible tool for the analysis of code smells. In *34th International Conference of the Chilean Computer Science*. SCCC, 1–11.
[39] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace. 2014. An approach to prioritize code smells for refactoring. *Automated Software Engineering* (2014).
[40] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in software engineering*. Springer Science and Business Media.