Uma análise da associação de co-ocorrência de anomalias de código com métricas estruturais

Carlos E. C. Dantas, Marcelo de A. Maia

Faculdade de Computação - FACOM
Universidade Federal de Uberlândia (UFU)

carloseduardodantas@iftm.edu.br,marcelo.maia@ufu.br

Abstract. *Source code anomalies (code bad smells) were characterized as symptoms of poor programming practices that should be either avoided or eliminated by refactoring. Several recent studies have recognized that the impact caused by these source code anomalies does not have the same severity. Although these studies have been conducted in order to understand the dynamics of the life cycle of the source code anomalies, there is little knowledge about how the source code anomalies evolve in the affected entities, especially regarding the question of how a code anomaly in an entity may induce the co-occurrence of other code anomaly during its evolution. This work studied the evolution of 5 systems, analyzing 12 types of anomalies that were introduced during the life cycle of their respective developments. We observed that 6 inter-relations anomalies between the anomalies analyzed were more prominent, and Long Method has great influence in all founded anomalies. With this anomaly, classes have highter coupling and lower cohesion, besides being more likely to introduce new types of anomalies.*

Resumo. *Anomalias de código (code bad smells) foram caracterizadas como sintomas de más práticas de programação que deveriam ser evitadas ou então eliminadas por meio de refatoração. Vários estudos recentes já reconhecem que os danos causados pelas anomalias de código não têm a mesma severidade. Apesar de esses estudos terem sido conduzidos com o intuito de entender a dinâmica do ciclo de vida das anomalias de código, ainda existe pouco conhecimento a respeito de como tais anomalias evoluem nas entidades afetadas, especialmente em relação à questão de como uma anomalia de código em uma entidade pode induzir co-ocorrência de outras anomalias durante sua evolução. Este trabalho estudou a evolução de código de 5 sistemas, analisando 12 tipos de anomalias que foram introduzidas no decorrer das suas respectivas evoluções. Observou-se que 6 inter-relações entre as anomalias analisadas foram mais proeminentes, com a anomalia Long Method exercendo influência em todas as inter-relações encontradas. Com a presença desta anomalia, foi constatado que classes possuem maior acoplamento e menor coesão, além de estarem mais propensos a introduzir novos tipos de anomalias.*

1. Introdução

Para atender os requisitos solicitados no menor tempo e custo possíveis, sistemas que adotam orientação a objetos (OO) deveriam seguir princípios básicos de *design*, como

polimorfismo, encapsulamento, abstração de dados e modularidade. Contudo, os desenvolvedores podem violar alguns desses princípios por vários motivos, seja pela urgência na entrega do sistema [Lavallée and Robillard 2015], pela adoção de uma solução deficiente no projeto das classes do sistema, ou simplesmente pelo desconhecimento quanto às boas práticas em desenvolvimento de sistemas OO [Palomba et al. 2014]. Estas violações resultam no surgimento de algumas anomalias nas classes desses sistemas. Várias dessas anomalias foram catalogadas por [Fowler et al. 1999], onde foram apelidadas de *code bad smells* (ou *code smells*). Em linhas gerais, a presença de anomalias nas classes de um sistema pode apresentar danos como o decremento da compreensibilidade [Abbes et al. 2011] e manutenibilidade [Yamashita and Moonen 2012]. Além disso, tais sistemas podem se tornar mais propensos a falhas, e sofrer mais mudanças no decorrer do tempo [Khomh et al. 2012].

Enquanto várias pesquisas têm analisado o comportamento das anomalias de maneira uniforme, alguns estudos têm buscado o discernimento sobre a severidade que cada tipo de anomalia exerce sobre as classes. Por exemplo, [Sjøberg et al. 2013] mostraram que nem toda anomalia é igualmente maléfica, pois algumas não são propensas a afetar diretamente a manutebilidade das classes de um sistema. Já [Lozano et al. 2007] apresentaram que certas anomalias podem não apenas dificultar a manutenibilidade, mas também influenciar no surgimento de novas anomalias, além de permanecerem por um longo período de tempo nas classes. Por fim, [Yamashita and Moonen 2013] mostraram que grupos específicos de anomalias possuem a tendência de estarem presentes nas mesmas classes, e que juntas podem multiplicar os danos sobre as mesmas. Com isso, cada grupo representa um conjunto de danos para as classes de um sistema.

Diante dos trabalhos citados, necessita-se de estudos sobre como os diferentes tipos de anomalias podem evoluir no decorrer dos *commits* de uma classe, e com isso, influenciar no surgimento de outras anomalias. Além disso, esta inter-relação entre as anomalias podem multiplicar os efeitos danosos sobre as classes. Assim, alguns tipos de anomalias poderiam ser priorizados para remoção em detrimento de outros, visando evitar a multiplicação dos efeitos danosos sobre as classes de um sistema, como mencionado por [Yamashita and Moonen 2012]. O cenário ideal seria que, ao serem detectadas, todas as anomalias fossem removidas. Contudo, aplicar operações de refatoração possuem um certo custo, além dos riscos de novos defeitos serem introduzidos no sistema. Além disso, outros trabalhos já mostraram que nem sempre o desenvolvedor está interessado em remover anomalias, o que eleva a importância da priorização mencionada [Lavallée and Robillard 2015].

O restante deste trabalho está organizado como segue. A metodologia do estudo, com os objetivos, a descrição sobre a coleta e análise dos dados são descritas na Seção 2. Os resultados do estudo são apresentados na Seção 3. Os trabalhos relacionados são citados na Seção 4. Na Seção 5 são apresentadas as limitações deste trabalho, e as conclusões e os trabalhos futuros são expostos na Seção 6.

2. Metodologia do Estudo

Os objetivos deste trabalho são: 1) avaliar a evolução dos tipos de anomalias sobre as classes de sistemas com código-fonte aberto, descobrindo possíveis inter-relações entre as mesmas sobre um conjunto de classes, e 2) analisar os resultados de medições envolvendo

indicadores de coesão e acoplamento para as classes afetadas por tais inter-relações, e compará-los com classes não afetadas por tais inter-relações. Com isso, será possível avaliar se determinadas inter-relações precisam de priorização para remoção, evitando a multiplicação dos seus efeitos danosos.

2.1. Coleta de Dados

Os sistemas utilizados neste estudo foram: *Apache Ant* [1], *JHotDraw* [2], *Apache Log4j* [3], *Lucene-Solr* [4] e *Apache Tomcat* [5]. Todos estes foram extraídos do *Github* [6]. O *download* dos sistemas e a extração dos *commits* de cada classe foram realizados através de comandos próprios do controle de versão *git*, como *git clone*, *git log* e *git show* . A **Tabela 1** apresenta as características de cada sistema utilizado neste trabalho, onde são mostrados a quantidade de classes analisadas de cada sistema (foram desconsideradas classes que são destinadas a exemplos e testes unitários, pois estas não possuem funcionalidades aderentes ao propósito de cada sistema). Além disso, também é apresentado a quantidade de classes onde algum um tipo de anomalia tenha sido detectado em pelo menos um *commit*. Por fim, são apresentados a quantidade de *commits* que cada sistema possui, e o intervalo de tempo entre o primeiro e o último *commit* de cada sistema. O critério para escolha dos sistemas ocorreu em função da relevância de cada um para a comunidade de desenvolvedores Java, e pela variação quanto ao propósito de cada sistema (arquiteturas distintas), quantidade de classes, anomalias e *commits*.

Tabela 1. Sistemas utilizados para análise das anomalias de código

Sistema	Classes	Classes c/ Anomalias	*Commits*	Intervalo dos *Commits*
Ant	2.778	1.315	13.342	Jan/2000 a Jun/2016
JHotDraw	1.691	882	747	Out/2000 a Dez/2015
Log4j	1.032	595	3.275	Nov/2000 a Jun/2015
Lucene-Solr	7.163	4.102	24.958	Set/2001 a Abr/2016
Tomcat	3.320	1.933	17.246	Mar/2006 a Jun/2016
Total	15.984	8.827	59.568	-

Para detectar as anomalias sobre cada *commit* de cada classe, foi utilizada a ferramenta *DECOR* [Moha et al. 2010], com suas métricas já estabelecidas por padrão. A **Tabela 2** apresenta a quantidade de classes em que pelo menos um *commit* apresentou algum dos diferentes tipos de anomalias detectados pelo *DECOR*. Também é mostrado o percentual de ocorrências em que cada tipo de anomalia incide sobre o total de classes de cada sistema. Observa-se que as anomalias mais frequentes sobre as classes foram *Long Method, Lazy Class, Long Parameter List* e *Complex Class*. Além disso, em virtude das diferenças arquiteturais entre os sistemas, algumas anomalias são mais frequentes em alguns sistemas em detrimento de outros. Por exemplo, a anomalia *Anti Singleton* ocorre em 11,3% das classes do sistema *Log4j*, enquanto que em apenas 0,8% no sistema *Ant*.

[1] https://github.com/apache/ant.git
[2] https://github.com/wumpz/jhotdraw.git
[3] https://github.com/apache/log4j.git
[4] https://github.com/apache/lucene-solr.git
[5] https://github.com/apache/tomcat.git
[6] http://www.github.com

Em contrapartida, a anomalia *Long Method* ocorre com proporções semelhantes em todos os sistemas, indicando que, independente do propósito do sistema, é comum ocorrer de métodos possuírem uma quantidade elevada de linhas de código.

Tabela 2. Quantidade de classes que apresentaram cada tipo de anomalia detectada pelo DECOR [Moha et al. 2010]

Anomalia	*Ant*	*JHotDraw*	*Log4j*	*Lucene-Solr*	*Tomcat*
Anti Singleton(AS)	23(0,8%)	35(2,0%)	117(11,3%)	311(4,3%)	279(8,40%)
Base Class S. Be Abstract(BC)	13(0,4%)	5(0,2%)	0(0%)	23(0,3%)	8(0,2%)
Class Data S. Be Private(CD)	31(1,1%)	33(1,9%)	24(2,3%)	418(5,8%)	103(3,1%)
Complex Class(CC)	298(10,7%)	299(17,6%)	143(13,8%)	1.267(17,6%)	500(15,0%)
Large Class(LGC)	4(0,1%)	0(0%)	4(0,3%)	24(0,33%)	39(1,1%)
Lazy Class(LZC)	377(13,5%)	168(9,9%)	162(15,6%)	1.046(14,6%)	968(29,1%)
Long Method(LM)	916(32,9%)	564(33,3%)	387(37,5%)	2.788(38,7%)	1.108(33,3%)
Long Parameter List(LPL)	341(12,2%)	251(14,8%)	137(13,2%)	997(13,9%)	318(9,5%)
Many Field Attributes(MFA)	0(0%)	0(0%)	0(0%)	14(0,1%)	4(0,1%)
Refused Parent Bequest(RPB)	15(0,5%)	0(0%)	9(0,8%)	51(0,7%)	11(0,3%)
Spaghetti Code(SC)	2(0,0%)	10(0,5%)	15(1,4%)	46(0,6%)	37(1,1%)
Speculative Generality(SG)	0(0%)	1(0,0%)	0(0%)	20(0,2%)	0(0%)

2.2. Análise de Dados

Para descobrir quais relações são mais frequentes entre os diferentes tipos de anomalias, foram utilizadas regras de associação aplicando o algoritmo *APRIORI* [Agrawal and Srikant 1994]. Os limiares empregados foram 3% de suporte sobre as classes que apresentaram algum tipo de anomalia, e 80% de confiança sobre cada regras encontrada. O critério para a escolha do percentual de suporte ocorreu em virtude da busca pelo equilíbrio entre obter a maior quantidade possível de tipos de anomalias distribuídos entre as regras encontradas, desde que cada regra possua frequência de pelo menos algumas dezenas de classes, em função de maximizar a quantidade de casos para cada regra. Por fim, o critério para o percentual de confiança se deve pelo interesse em descobrir inter-relações entre tipos de anomalias que ocorrem constantemente a partir das regras encontradas.

Após obter o conjunto de regras, foram aplicadas quatro métricas estruturais sobre as classes envolvidas em cada relação. Segue uma breve descrição de cada métrica.

- *CBO (Coupling Between Object Classes)* - quantidade de classes que estão acopladas com a classe analisada.
- *RFC (Response for Class)* - quantidade de métodos que são executados pela classe internamente quando alguma mensagem é recebida de outra classe.
- *LCOM (Lack of Cohesion)* - efetua um cálculo sobre a quantidade de métodos que não utilizam determinado atributo da classe.
- *Ca (Afferent couplings)* - quantidade de classes que utilizam alguma funcionalidade da classe analisada.

Para calcular tais métricas, foi utilizada a ferramenta *COPE* [Kakarontzas et al. 2013]. Para todas as métricas, quanto maior for o valor calculado, mais danosos são os efeitos que determinado conjunto de anomalias empregam sobre a classe analisada.

3. Resultados

A **Tabela 3** apresenta os resultados da aplicação do algoritmo *APRIORI* sobre as classes dos sistemas empregados neste trabalho. São mostradas as regras extraídas para cada sistema, com a quantidade de classes envolvidas. Além disso, também é apresentado percentual de confiança para cada regra. No total, foram encontradas 14 regras, sendo 6 regras distintas, envolvendo de 6 tipos de anomalias. Observa-se ainda que todas as regras encontradas implicam no tipo de anomalia *Long Method*. Isso indica que este tipo de anomalia exerce um papel centralizador sobre as demais anomalias com o qual esta se relaciona. A seguir, serão apresentados exemplos sobre 4 das 6 regras encontradas.

Tabela 3. Resultado do algoritmo APRIORI sobre as classes afetadas por diferentes tipos de anomalias

Sistema	Regra	Confiança
Ant	(CC,LPL)(124) → LM(113)	91%
Ant	CC(298) → LM(259)	87%
JHotDraw	(CC,LPL)(71) → LM(57)	80%
Log4j	(AS,LPL)(21) → LM(19)	90%
Log4j	(AS,CC)(25) → LM(22)	88%
Log4j	(CC,LPL)(44) → LM(36)	82%
Lucene-Solr	(CC,LPL)(438) → LM(386)	88%
Lucene-Solr	CC(1.267) → LM(1061)	84%
Lucene-Solr	(CD,CC)(160) → LM(130)	81%
Tomcat	(AS,CC)(122) → LM(111)	91%
Tomcat	(CC,LC)(77) → LM(67)	87%
Tomcat	(CC,LPL)(157) → LM(135)	86%
Tomcat	(AS,LPL)(72) → LM(61)	85%
Tomcat	CC(500) → LM(402)	80%

Para exemplificar a regra *(Complex Class → Long Method)* encontrada no sistema *Ant*, a classe `org.apache.ant.core.execution.ExecutionFrame.java` possui a anomalia *Complex Class* desde a sua criação, em virtude dos diversos fluxos anternativos existentes nos seus métodos, como por exemplo o método `parsePropertyString()`. Em virtude da complexidade da classe, alguns métodos estão propensos a possuírem várias linhas, apresentando a anomalia *Long Method*, como o método `fillinDependencyOrder()`, que possui 60 linhas de código no *commit 362903a*.

Com relação à regra *(Complex Class,Lazy Class → Long Method)* encontrada no sistema *Tomcat*, a classe `javax.servlet.http.HttpUtils.java` foi concebida para possuir métodos estáticos sem atributos, por isso foi detectada a presença da anomalia *Lazy Class*. Contudo, o método estático `parseName()` apresenta vários caminhos de execução distintos, apresentando a anomalia *Complex Class*, como consta no *commit 50a1d0b*.

Por fim, a última regra a ser exemplificada é a *(Class Data Should Be Private,Complex Class → Long Method)* encontrada no sistema *Lucene-Solr*. Esta apresenta uma situação diferente das demais regras. Como exemplo, a classe

`org.apache.lucene.queryParser.ParseException.java` apresenta atributos públicos desde a sua criação, possibilitando edições por parte das classes que estão acopladas a esta. Contudo, esta classe possui métodos complexos que fazem uso desses atributos públicos. O método `getMessage()`, apresenta a anomalia *Complex Class*, quebrando o encapsulamento da classe. E como esta classe foi crescendo a cada commit, surgiu a anomalia *Long Method*.

Para avaliar o quão danoso é a presença do tipo de anomalia *Long Method* sobre as regras, foram calculadas as métricas estruturais *CBO, LCOM, RFC e Ca* sobre as classes envolvidas em cada regra, separando pelas classes que possuem e as que não possuem o tipo de anomalia *Long Method*. Por exemplo, como foi mostrado na **Tabela 3**, a regra *(Complex Class → Long Method)* encontrada no sistema *Ant* apresentou 259 casos onde a anomalia *Long Method* surgiu, e 39 casos onde tal anomalia não foi detectada. A **Tabela 4** apresenta os resultados sobre os itens citados, e também mostra o percentual de classes que apresentam outros tipos de anomalias além dos tipos que cada regra já apresenta.

Como pode ser observado, as classes que apresentaram o tipo de anomalia *Long Method* apresentaram valores consideravelmente maiores para as métricas estruturais, ou seja, tal anomalia contribui muito mais para a baixa coesão e o alto acoplamento entre as classes. Por exemplo, a classe *org.apache.catalina.core.StandardContext* do sistema *Tomcat* apresentou os valores 50724 para *LCOM*, 75 para *CBO*, 15 para *Ca* e 663 para *RFC*. Esta classe possui dezenas de atributos para finalidades distintas, com cada método utilizando um subconjunto desses atributos. Um detalhe importante é que além das anomalias mostradas na regra, esta classe também apresentou as anomalias *Spagheti Code* e *Large Class*. O mesmo aconteceu para diversas outras classes, como mostra o percentual na **Tabela 4**. Este percentual mostra que, as classes que apresentaram o tipo de anomalia *Long Method* estão mais propensas a serem infectadas com outras anomalias, se comparadas às classes que não apresentaram tal tipo de anomalia.

Tabela 4. Métricas de acoplamento e coesão sobre as regras encontradas

Regra	Classes com *Long Method*					Classes sem *Long Method*				
	CBO	RFC	LCOM	Ca	%	CBO	RFC	LCOM	Ca	%
(CC,LPL) → LM	15,17	71,10	344,38	9,60	30,49	7,40	32,34	64,13	2,01	23,47
CC → LM	12,23	60,33	256,23	8,23	53,40	5,56	31,02	57,13	4,55	44,32
(AS,LPL) → LM	19,24	127,89	973,24	8,16	30,49	3,00	27,00	18,66	3,33	23,47
(AS,CC) → LM	17,16	119,41	1.171,64	17,52	79,78	2,00	38,00	155,00	6,00	64,28
(CD,CC) → LM	14,44	92,60	709,41	11,30	80,95	4,75	18,61	30,18	4,31	46,29
(CC,LC) → LM	15,74	90,80	1.247,90	10,11	73,07	1,34	25,23	48,30	11,17	57,89

4. Trabalhos Relacionados

Diversos estudos empíricos foram realizados para explicar os comportamentos de anomalias que surgem nas classes dos sistemas. [Yamashita and Moonen 2012] fizeram um estudo sobre o impacto das anomalias sobre a manutenibilidade dos sistemas, onde concluíram que aspectos importantes em manutenibilidade, como simplicidade, encapsulamento e duplicidade de código-fonte são degradados em função das anomalias que surgem nas classes. Este se difere desse trabalho, que analisa a relação entre as anomalias, sem analisar os efeitos negativos que estas transmitem às classes. Além disso, o trabalho citado não destaca características individuais de cada tipo de anomalia.

[Lozano et al. 2007] fez um estudo empírico sobre a evolução da anomalia *Code Clones* em cima do histórico de versões do código-fonte. O estudo observou que uma anomalia possui tendência de promover o aparecimento de outras anomalias. O trabalho proposto busca mostrar uma relação semelhante, mas destacando quais anomalias se relacionam, e estendendo a várias anomalias e não apenas ao *Code Clone*.

[Sjøberg et al. 2013] conduziram uma pesquisa que buscava verificar se as anomalias exercem influência sobre o esforço de manutenção nas classes afetadas. Concluíram que a grande maioria dessas anomalias oferecem maior esforço, exceto a anomalia *Refused Bequest*. Isto indica que anomalias não possuem a mesma gravidade sobre as classes. Neste trabalho, métricas estruturais envolvendo acoplamento e coesão são avaliadas para verificar o impacto das anomalias em tais classes.

O trabalho mais relacionado com este se trata de [Yamashita and Moonen 2013], que pesquisou sobre a inter-relação entre as anomalias, enfatizando nos efeitos que determinadas anomalias juntas podem afetar a manutenibilidade das classes. Para tal finalidade, foram analisadas 12 tipos de anomalias, com detecção de 3 problemas de manutenibilidade e 5 fatores que relacionam as anomalias. Com isso, obtém-se tendências de comportamento sobre as classes. Embora este trabalho também encontra inter-relação entre as anomalias, o objetivo consiste em explicar o comportamento de tais inter-relações sobre as classes, enquanto que este trabalho verifica o nível de deterioração das classes com tais inter-relações.

Por fim, [Tufano et al. 2015] fizeram uma análise em cerca de 500.000 *commits*, onde 9.164 desses possuíam classes afetadas com anomalias. Neste estudo, uma das conclusões foi que as classes geralmente são afetadas por anomalias logo na sua criação, ao invés das manutenções que a classe sofre no decorrer dos *commits*. Para coletar tais resultados, foi construída uma ferramenta chamada *History Miner* que utiliza a ferramenta *DECOR* [Moha et al. 2010] para detectar as anomalias. Esta metodologia possui algumas interseções com este trabalho, contudo são coletados outros indicadores como acoplamento e coesão das classes, além da utilização de regras de associação para detectar a inter-relação entre as anomalias.

5. Conclusões e Trabalhos Futuros

Em geral, as principais conclusões obtidas neste trabalho foram:

1. Existem tipos de anomalias que possuem fortes inter-relações, tendo em vista que o percentual de confiança empregado foi consideravelmente alto. Além disso, foi constatado que alguns tipos de anomalias ocorrem com maior frequência nas classes, dificultando a obtenção de regras dos demais tipos de anomalias, por não possuírem amostras suficientes para obter tais regras.
2. O tipo de anomalia *Long Method* surgiu em todas as regras encontradas, onde foi constatado que a presença deste tipo de anomalia nas classes é extremamente danoso, além de facilitar no surgimento de novos tipos de anomalias nas classes. Caso tal anomalia são seja detectada, mesmo que a classe apresente outros tipos de anomalias, os danos às classes são consideravelmente menores, reforçando que nem todas as anomalias são igualmente danosas.

Para trabalhos futuros, recomenda-se efetuar uma avaliação sobre todos os tipos de anomalias, obtendo quais destas são mais danosas. Além disso, obter um conjunto

maior de sistemas para coletar uma quantidade maior de amostras para anomalias que surgem menos frequentemente, buscando regras para estas.

Referências

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

Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *20th Int. Conference on Very Large Data Bases (VLDB)*, pages 487–499.

Fowler, M., Beck, K., Brant, T., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Kakarontzas, G., Constantinou, E., Ampatzoglou, A., and Stamelos, I. (2013). Layer assessment of object-oriented software: A metric facilitating white-box reuse. *Journal of Systems and Software*, 86(2):349 – 366.

Khomh, F., Penta, M. D., Guéhéneuc, Y.-G., and Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275.

Lavallée, M. and Robillard, P. N. (2015). Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In *37th Int. Conference on Software Engineering (ICSE)*, pages 677–687.

Lozano, A., Wermelinger, M., and Nuseibeh, B. (2007). Assessing the impact of bad smells using historical information. In *9th Int. Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, pages 31–34.

Moha, N., Guéhéneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., and Lucia, A. D. (2014). Do they really smell bad? A study on developers' perception of bad code smells. In *30th Int. Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110.

Sjøberg, D. I. K., Yamashita, A. F., Anda, B. C. D., Mockus, A., and Dybå, T. (2013). Quantifying the effect of code smells on maintenance effort. *Trans. Soft. Eng.*, 39(8).

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *37th Int. Conference on Software Engineering (ICSE)*, pages 403–414.

Yamashita, A. F. and Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *28th Int. Conference on Software Maintenance (ICSM)*, pages 306–315.

Yamashita, A. F. and Moonen, L. (2013). Exploring the impact of inter-smell relations on software maintainability: an empirical study. In *35th Int. Conference on Software Engineering (ICSE)*, pages 682–691.