Investigating Code Smell Co-Occurrences using Association Rule Learning: A Replicated Study

Fabio Palomba[1,2], Rocco Oliveto[3], Andrea De Lucia[1]
[1]University of Salerno, Italy — [2]Delft University of Technology, The Netherlands — [3]University of Molise, Italy
fpalomba@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it

***Abstract*—Previous research demonstrated how code smells (*i.e.,* symptoms of the presence of poor design or implementation choices) threat software maintainability. Moreover, some studies showed that their interaction has a stronger negative impact on the ability of developers to comprehend and enhance the source code when compared to cases when a single code smell instance affects a code element (*i.e.,* a class or a method). While such studies analyzed the effect of the co-presence of more smells from the developers' perspective, a little knowledge regarding which code smell types tend to co-occur in the source code is currently available. Indeed, previous papers on smell co-occurrence have been conducted on a small number of code smell types or on small datasets, thus possibly missing important relationships. To corroborate and possibly enlarge the knowledge on the phenomenon, in this paper we provide a large-scale replication of previous studies, taking into account 13 code smell types on a dataset composed of 395 releases of 30 software systems. Code smell co-occurrences have been captured by using association rule mining, an unsupervised learning technique able to discover frequent relationships in a dataset. The results highlighted some expected relationships, but also shed light on co-occurrences missed by previous research in the field.**

***Index Terms*—Code Smells; Empirical Studies; Association Rule Mining;**

I. Introduction

During software maintenance and evolution, a software system experience several changes aimed at enhancing existing features or fixing important bugs [1]. Developers usually perform such changes while they are in a hurry [2], increasing the risk to worsen the internal quality of the system by introducing the so-called *technical debts* [3], *i.e., not-quite-right* code possibly written in a rush to meet a deadline or to deliver the software to the market in the shortest time possible [3], [4], [5], [6]. *Code smells*, *i.e.,* symptoms of poor design and implementation choices [7], represent one important factor contributing to technical debt, and possibly affecting the maintainability of a software system [5]. In the past, and most notably in the recent years, the research community devoted a lot of effort in developing approaches and tool for their automatic detection in source code [8], [9], [10], [11], [12], [13], [14], [15], [16]. Moreover, a consistent research has been conducted to study the impact of code smells on non-functional attributes of source code. In particular, empirical studies have been conducted to investigate (i) the relevance of smells from the developers' perspective [17], [18], (ii) their longevity [19], [20], [21], [22], [23], and (iii) their impact on non-functional attributes of source code, such as program comprehension [24], change- and fault-proneness [25], [26], and, more in general, on maintainability [27], [28], [29], [30]. Recently, Yamashita and Moonen [29] demonstrated that the interaction of code smells consistently inhibits the ability of developers to maintain the source code. A possible reason behind this phenomenon has been provided by Abbes *et al.* [24], which showed how the interaction of more smells have a strong effect of program comprehension. Although these studies empirically showed the harmfulness of the interaction of code smells, only a little knowledge about which smells co-occur together is available in literature. Indeed, while Bavota *et al.* [31] and Palomba *et al.* [32] investigated the co-occurences of test smells (*i.e.,* bad implementation practices occurring in test code [33]), only the works by Anubhuti *et al.* [34] and Arcelli Fontana *et al.* [35] systematically explored the relationships between the smells occurring in the production code. However, such studies have been conducted taking into account a relatively small set of code smells (*i.e.,* 7 smells considered by Anubhuti *et al.* [34] and 6 by Arcelli Fontana *et al.* [35]) or a small number of object systems (*i.e.,* Anubhuti *et al.* only analyzed 2 software projects). As a consequence, these previous works may potentially miss important relationships between code smells.

In this paper, we aim at corroborating and possibly improving the current knowledge about the phenomenon of code smell co-occurrences, by conducting a replication study which considers 13 code smell types having different characteristics and different granularity on a large dataset composed of 30 open-source software systems. To derive the relationships between the considered smells, we exploited the association rule learning [36], a technique able to discover local patterns highlighting attribute value conditions that occur together in a given dataset [36]. The results of the study indicate the presence of six pairs of code smells that frequently co-occur together. Some relationships are quite expected (*e.g., Long Method* and *Spaghetti Code*), while others are less obvious and more tricky to understand, as in the case of the frequent co-occurrence between *Message Chains* and *Refused Bequest*, that is due to the higher probability that classes implementing several methods have to be involved in a long chain of method calls.

Structure of the paper. Section II discusses the related literature about code smell co-occurrence. Section III describes the empirical setup, while Section IV reports the results of the

study. Section V discusses the threats possibly affecting the validity of our experiment. Finally, Section VI concludes the paper.

II. Related Work

Code smells have been widely investigated under different perspectives. Due to space limitation, in this paper we focus our attention only on the works analyzing the relationships between code smells, while a detailed analysis of the detection techniques proposed in literature is available in [37].

A number of studies have been carried out to investigate the relationships between test smells, *i.e.,* poor design choices (similarly to code smells) applied by programmers when developing test cases [33]. Bavota *et al.* [31] conducted an empirical investigation in order to study (i) the diffusion of test smells in 18 software projects, (ii) the relationships between them, and (iii) their effects on software maintenance. The results of the study demonstrated that 82% of JUnit classes in their dataset are affected by at least one test smell, and that the *Assertion Roulette* smell [33] is highly present together with the other smells. Moreover, the presence of design flaws has a strong negative impact on the maintainability of the affected classes. The same experimental design has been used by Palomba *et al.* [32] in the context of test cases automatically generated by EvoSuite [38], observing that three test smells, *i.e., Assertion Roulette*, *Eager Test*, and *Test Code Duplication* [33], frequently occur in test classes automatically generated by the tool.

Finally, Tufano *et al.* [39] performed an empirical analysis of the co-occurrences between test and code smells in the context of a more general investigation into the nature of test smells. The results showed that some test and production smells are generally related, as in the case of *Assertion Roulette* and *Spaghetti Code* [39]. In a closely related field, Cardoso and Figueiredo [40] explored the relationships between design patterns [41] and code smells on five open source systems. They discovered some relationships, as in the case of *Command* with *God Class* and *Template Method* with *Duplicated Code*.

As for studies investigating the smells occurring in the production code, Anubhuti *et al.* [34] studied the co-occurrences of 7 code smell types in two open source projects, *i.e.,* Chromium and Mozilla. In particular, they evaluated the percentage of smells co-occurring over the change history of such projects, finding that often traditional code smells (*i.e.,* Feature Envy and Data Clumps) co-exist together with code duplication. Unlike this work, we setup an experiment with a larger number of systems (*i.e.,* 30 vs 2) and a larger number of code smells (*i.e.,* 13 vs 7).

Yamashita *et al.* [42] presented a replicated study where they analyzed the problem of code smell interaction in both open and industrial systems, finding that the relation between smells vary depending on the type of system taken into account. Our study is complementary to the work by Yamashita *et al.* since it considers different smells with respect to previous research. Moreover, the code smells have been detected manually, rather than using automatic tools. Finally, we considered a larger number of systems (*i.e.,* 30 vs 3).

Also Arcelli Fontana *et al.* [35] studied the phenomenon of code smell co-occurrence by counting the percentage of smells present in the same class during the history of the software projects coming from the Qualitas Corpus [43]. The authors found that only in a small percentage of cases (*i.e.,* 3% of average) a *Brain Method* co-occur with other smells as *Dispersed Coupling* and *Message Chains*. Our study has been carried out considering a larger number of smells, and highlight several other co-occurrences between code smells.

III. Empirical Study Definition and Design

The *goal* of the empirical study is to determine which code smells co-occur together, with the *purpose* of identifying the relationships between them and possibly improve code smell detection and prioritization. The *perspective* is of researchers interested in building better recommenders for developers based on the interaction between design flaws. Formally, the research question investigated in this paper is:

RQ: *Which code smells co-occur together?*

The following sections detail the context of the study and the data extraction/analysis process.

A. Context Selection

The *context* of the study consists of (i) code smell types, and (ii) object systems. As for the types of design flaws, Table I reports the name of the 13 code smells studied along with a short definition. As better described later in this Section, we needed to limit our analysis to 13 code smells from the catalogues by Fowler [7] and Brown *et al.* [44] because of the manual detection we applied to discover instances of such smells. However, we carefully selected code smells having different characteristics (*e.g.,* classes characterized by long/complex code as well as violation of Object-Oriented principles) and different granularity (*i.e.,* class-level and method-level smells).

As for the object systems, our analysis is carried out on 395 releases of 30 open source systems hosted on the `Git` repository. Table II reports the analyzed systems, the number of releases considered for each of them, and their size ranges in terms of number of classes, number of methods, and KLOCs.

B. Data Extraction

To answer our research question we firstly needed the source code of each release taken into account. To this aim, one of the authors identified the dates in which the major releases of the 30 considered systems were issued. This was done by exploiting the `Git` tags (often used to tag releases). Note that we just considered major releases since those are the ones generally representing a real deadline for developers, while minor releases are sometimes issued just due to bug fixes or minor changes. Once having the release dates, the source code corresponding to the snapshots committed in that dates has been downloaded using the `git clone` command.

TABLE I: The Code Smells considered in our Study

Name	Description
Class Data Should Be Private (CDSBP)	A class exposing its fields, violating the principle of data hiding.
Complex Class	A class having at least one method having a high cyclomatic complexity.
Feature Envy	A method is more interested in a class other than the one it actually is in.
God Class	A large class implementing different responsibilities and centralizing most of the system processing.
Inappropriate Intimacy	Two classes exhibiting a very high coupling between them.
Lazy Class	A class having very small dimension, few methods and low complexity.
Long Method	A method that is unduly long in terms of lines of code.
Long Parameter List (LPL)	A method having a long list of parameters, some of which avoidable.
Message Chain	A long chain of method invocations is performed to implement a class functionality.
Middle Man	A class delegates to other classes most of the methods it implements.
Refused Bequest	A class redefining most of the inherited methods, thus signaling a wrong hierarchy.
Spaghetti Code	A class implementing complex methods interacting between them, with no parameters, using global variables.
Speculative Generality	A class declared as abstract having very few children classes using its methods.

TABLE II: Systems involved in the study

System	#Releases	Classes	Methods	KLOCs
ArgoUML	16	777-1,415	6,618-10,450	147-249
Ant	22	83-813	769-8,540	20-204
aTunes	31	141-655	1,175-5,109	20-106
Cassandra	13	305-586	1,857-5,730	70-111
Derby	9	1,440-1,929	20,517-28,119	558-734
Eclipse Core	29	744-1,181	9,006-18,234	167-441
Elastic Search	8	1,651-2,265	10,944-17,095	192-316
FreeMind	16	25-509	341-4,499	4-103
Hadoop	9	129-278	1,089-2,595	23-57
HSQLDB	17	54-444	876-8,808	26-260
Hbase	8	160-699	1,523-8148	49-271
Hibernate	11	5-5	15-18	0.4-0.5
Hive	8	407-1,115	3,725-9,572	64-204
Incubating	6	249-317	2,529-3,312	117-136
Ivy	11	278-349	2,816-3,775	43-58
Lucene	6	1,762-2,246	13,487-17,021	333-466
JEdit	23	228-520	1,073-5,411	39-166
JHotDraw	16	159-679	1,473-6,687	18-135
JFreeChart	23	86-775	703-8,746	15-231
JBoss	18	2,313-4,809	19,901-37,835	434-868
JVlt	15	164-221	1,358-1,714	18-29
jSL	15	5-10	26-43	0.5-1
Karaf	5	247-470	1,371-2,678	30-56
Nutch	7	183-259	1,131-1,937	33-51
Pig	8	258-922	1,755-7,619	34-184
Qpid	5	966-922	9,048-9,777	89-193
Sax	6	19-38	119-374	3-11
Struts	7	619-1,002	4,059-7,506	69-152
Wicket	9	794-825	6,693-6,900	174-179
Xerces	16	162-736	1,790-7,342	62-201
Total	**395**	**5-4,809**	**15-37,835**	**0.4-868**

As for the detection of the design flaws occurring in each release analyzed, our goal was to consider a set of smells as close as possible to the actual set. Indeed, imprecisions in the detection would have lead to imprecisions in the results of the study. For this reason, we preferred to manually detect the instances of the 13 code smells, rather than using any detector available in literature [8], [10], [9]. This choice has been done because the code smell detectors generally try to find a good compromise between the precision and the completeness of the recommendations, which lead to output a number of false positive instances as well as to miss some true negatives. A manual validation, instead, is supposed to be more accurate.

However, to facilitate the manual detection we developed a simple tool that discarded the classes/methods that surely do not contain a certain code smell. Specifically, given the type of code smell under consideration as input, the tool analyzes the metric profile of classes/methods and outputs a list of code elements to further analyze manually. For instance, when analyzing the *Class Data Should Be Private*, we filter out all the classes having no public attributes because they cannot be affected by the considered smell. The complete list of rules used to filter out code elements is available in our online appendix [45].

Once having the output of the tool for each of the 13 smells, we started a two-step manual validation phase. In the first step, two of the authors (*i.e.,* the inspectors) individually analyzed and classified the code elements of each system as true positive or false positive for a given smell. The output consisted of a list of smells identified by each inspector.

In the second step, the produced oracles were compared, and the involved authors discussed the differences, *i.e.,* smell instances present in the oracle produced by one inspector, but not in the oracle produced by the other. All the code elements positively classified by both the inspectors have been considered as actual smells. Regarding the other instances, the inspectors opened a discussion in order to resolve the disagreement and taking a shared decision. The final output consisted of a unique list of smells that we used to answer our research question. This list is publicly available in our online appendix [45]. The manual validation procedure took, overall, ≈300 man-hours.

C. Data Analysis

Once gathered the data about the presence of each code smell type in each release analyzed, we mined association rules [36] for detecting sets of code smells that often co-occur (*i.e.,* affect the same code element). Association rule discovery is an unsupervised learning technique used to detect local patterns highlighting attribute value conditions that occur together in a given dataset [36]. Formally, let $I = \{i_1, \ldots, i_n\}$ be a set of n binary attributes called *items* and indicating the presence of a certain property in the element under consideration, and let $T = \{t_1, \ldots, t_m\}$ a set of m *transactions* indicating the set of all the elements analyzed, an association rule is defined as an implication of the form $X \Rightarrow Y$, where $X, Y \subseteq I$ and $X \cap Y = \emptyset$.

In our work, the set T is composed by all the classes present in a specific system release object of our study, while each item in the set I indicates the presence of a given smell in

that release. Therefore, the association rule analysis has been performed at class-level, *i.e.,* two code smells are considered co-occurring if they affect the same class. Specifically, an association rule $CS_{left} \Rightarrow CS_{right}$, between two disjoint sets of code smells implies that, if a class is affected by each $cs_i \in CS_{left}$, then the same class should be affected by each $cs_j \in CS_{right}$. The strength of an association rule is determined by its support and confidence [36]:

$$\textit{Support} = \frac{|CS_{left} \cup CS_{right}|}{T}$$

$$\textit{Confidence} = \frac{|CS_{left} \cup CS_{right}|}{|CS_{left}|}$$

where T is the total number of classes in the system release under analysis. In this paper, we performed association rule mining using a well-known algorithm, namely *Apriori* [36]. Such algorithm has been implemented using the `arules`[1] library of the `R` toolkit [46]. Note that the minimum *Support* and *Confidence* to consider an association rule as valid can be set in the *Apriori* algorithm. Since we were only interested in identifying meaningful association rules having high confidence and support, in our study we set quite high values for these two parameters, and in particular *Support* = 0.7 and *Confidence* = 0.9.

D. Replication Package

The tool developed to filter out classes not affected by design flaws, as well as the data used in the empirical study are publicly available in our online appendix [45].

IV. Analysis of the Results

TABLE III: Co-occurrence between code smells: association rule results.

Item Set #1	Item Set #2	Support	Confidence
Complex Class	Message Chains	0.94	1.0
Long Method	Long Parameter List	0.94	0.99
Long Method	Feature Envy	0.91	0.99
Spaghetti Code	Long Method	0.90	0.98
Inappropriate Intimacy	Feature Envy	0.79	0.97
Refused Bequest	Message Chains	0.79	0.91

Table III reports the co-occurrences of code smells identified in our study as result of the association rule discovery. The identified co-occurrences are sorted based on the level of *Confidence* reported by the *Apriori* algorithm.

The *Message Chain* code smell often co-occurs with the *Complex Class* ($supp = 0.94$, $conf = 1.0$). Remember that a *Message Chain* smell arises when a long chain of method invocations is performed when one method of the affected class is invoked, while a *Complex Class* is a class including methods having a high cyclomatic complexity. We looked into these smell instances to understand the reasons behind this strong co-occurrence relationship. We found that often the complex method(s) contained in the *Complex Class* are also the one(s) responsible for triggering the long chain of method calls resulting in a *Message Chain* code smell. For example, in the version `1.8` of Apache Ant, we found that the class `FTP` of the package `org.apache.tools.ant.taskdefs.optional`, responsible for implementing a basic FTP client that can send, receive, list, delete files, and create directories, is affected by a *Complex Class* smell. At the same time, the method `accountForIncludedDir` exhibits a *Message Chain* smell since it recursively calls 4 different methods belonging to different classes, resulting in a long chain of calls.

Another interesting case concerns the co-occurrence between the *Inappropriate Intimacy* and the *Feature Envy* code smells. An *Inappropriate Intimacy* smell occurs when two classes are highly coupled, while a *Feature Envy* affects a method that is more interested in (*i.e.,* has more dependencies toward) a class other than the one it is actually in. Our analysis of the co-occurring instances highlighted the quite obvious reason behind this strong relationship between the two code smells: the presence of a *Feature Envy* in a method m of class C_i clearly results in an increase of coupling between C_i and the C_j class "envied" by m, thus promoting the introduction of an *Inappropriate Intimacy* code smell between C_i and C_j. For example, we found in *Apache Xerces* that the method `scanEntityDecl` of the class `XMLEntityHandler` exhibits high levels of coupling with the class `StringReader`, since it needs to scan the declarations of entities by using common functionalities provided by `StringReader`. Such coupling strongly increases the number of dependencies between the two classes, resulting in an *Inappropriate Intimacy* smell.

The co-occurrences between the *Spaghetti Code* and the *Long Method* code smells ($supp = 0.90$, $conf = 0.98$ - see Table III) is a quite expected result, dictated by the *Spaghetti Code*'s definition (*i.e.,* a class implementing complex methods interacting between them, with no parameters and using global variables). Given the well-known relationship between size (method length) and complexity, it is reasonable to think that the complex methods present in a class affected by the *Spaghetti Code* smell are *Long Methods*.

The co-occurrences between the *Message Chain* and the *Refused Bequest* code smells ($supp = 0.79$, $conf = 0.91$) are less obvious and not expected. Remember that a class affected by *Refused Bequest* overrides most of the methods it inherits from its superclass. Looking inside these cases we simply found that *Refused Bequest* are often classes implementing several methods (on average 14, against the 6 implemented by classes not affected by this smell) thus having a higher chance of containing methods taking part in a *Message Chain*.

Finally, another interesting co-occurrence is the one between the *Feature Envy* and the *Long Method* code smells. Since *Long Methods* are composed of several code statements, including of course dependencies toward other classes, they are more likely to also being affected by the *Feature Envy* code smell. Besides the relationship with *Feature Envy*, the *Long Method* smell frequently co-occur with a *Long Parameter*

[1]https://cran.r-project.org/web/packages/arules/arules.pdf

List one. Again, one can expect that longer methods, likely implementing several class responsibilities, also require a higher number of parameters, increasing the chances of also being affected by a *Long Parameter List* smell.

When comparing our findings with the ones achieved in previous studies on code smell co-occurrence [34], [35], it is worth noting that the relationships we found have been not experienced previously. Therefore, our results can be considered as complementary to those reported by Anubhuti *et al.* [34] and Arcelli Fontana *et al.* [35].

In Summary. We identified six pairs of code smells that co-occur very often (see Table III). While some of these co-occurrences are quite expected (e.g., *Long Method* and *Spaghetti Code*), others are not (e.g., *Message Chains* and *Refused Bequest*), recalling the need for studying more deeply the reasons behind their appearance and their apparent relationships.

V. Threats to Validity

The main threat related to the relationship between theory and observation (*construct validity*) are due to imprecisions/errors in the measurements we performed. While detecting code smell instances manually, we exploited a tool in order to facilitate our validation. The tool had the goal to discard from the manual investigation the code elements that surely not contain any design flaw by using conservative heuristics to avoid the filtering of classes potentially affected by a smell. Moreover, the set of code smells subject of this study was built by two authors independently before discussing the the cases of disagreement and taking a shared decision. Still, we cannot exclude competely the presence of false positives/negatives in our dataset.

Threats related to the relationship between the treatment and the outcome (*conclusion validity*) are represented by the analysis methods exploited in our study. Unlike other papers on code smell co-occurrence [34], [35] we exploited a machine learning technique for discovering hidden patterns, *i.e.,* association rule mining [36] rather than considering the percentage of smells co-occurring during the change history. In addition, practical explainations of the reasons behind the relationships found have been proposed.

Finally, regarding the generalization of our findings (*external validity*) this is, to the best of our knowledge, the largest study—in terms of number of software releases (395), and considered code smell types (13)—concerning the analysis of code smell co-occurrence. However, we are aware that we limited our attention only to Java systems. Further studies aiming at replicating our work on systems written in other programming languages are desirable.

VI. Conclusion

The interaction of code smells in the same code element has a strong negative effect on program comprehension and ability of developers to maintain a software project, as highlighted by recent findings by Abbes *et al.* [24] and Yamashita and Moonen [29]. While previous research focused the attention on the analysis of the effect of the co-occurrence of more smells in the same source code, to date a little knowledge about which code smells generally co-occur together is available.

Indeed, previous studies on this field have been carried out on a small set of systems [34] or on few code smell types [35]. To enlarge the knowledge about the phenomenon, we conducted a large scale empirical investigation involving 13 code smell types and 395 releases of 30 software systems. Unlike previous works, the co-occurrences have been studied by exploiting association rule mining [36], a technique specialized to detect recurring patterns in a given dataset.

The results of the study highlight six pairs of code smells that frequently co-occur together. Some of such co-occurrences were quite expected because of the innate relationship between the involved smells, *e.g., Long Method* and *Spaghetti Code* or *Long Method* and *Long Parameter List*. Besides such cases, some relationships we found were less obvious. For instance, we found an unexpected relationship between *Message Chains* and *Refused Bequest*, which are due to the higher probability classes implementing many methods have to be involved in a long chain of method calls.

The findings of the study represent the main input for our future research agenda. We plan to extend our investigation to systems implemented in other programming languages, as well as studying the possibility to build a recommendation system able to prioritize code smell removal based on the information about co-occurrences of smells.

References

[1] M. M. Lehman and L. A. Belady, *Software Evolution - Processes of Software Change*. Academic Press London, 1985.

[2] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the International Conference on Software Engineering (ICSE) - Volume 1*. IEEE, 2015, pp. 403–414.

[3] W. Cunningham, "The WyCash portfolio management system," *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[4] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. L. Nord, I. Ozkaya, R. S. Sangwan, C. B. Seaman, K. J. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the Workshop on Future of Software Engineering Research, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2010, pp. 47–52.

[5] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[6] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman, *Perspectives on the Future of Software Engineering*. Springer, 2013, ch. Technical Debt: Showing the Way for Better Transfer of Empirical Results, pp. 179–190.

[7] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[8] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[9] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, May 2015.

[10] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "A textual-based technique for smell detection," in *Proceedings of the 24th International Conference on Program Comprehension (ICPC 2016)*. Austin, USA: IEEE, 2016, p. to appear.

[11] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2004, pp. 350–359.

[12] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the International Software Metrics Symposium (METRICS)*. IEEE, September 2005, p. 15.

[13] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[14] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the International Conference on Quality Software (QSIC)*. Hong Kong, China: IEEE, 2009, pp. 305–314.

[15] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on B-splines," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2010, pp. 248–251.

[16] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[17] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 101–110.

[18] A. F. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 242–251.

[19] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proceedings of the European Conference on Software Maintenance and ReEngineering (CSMR)*. IEEE, 2012, pp. 411–416.

[20] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *Proceedings of the International Workshop on Refactoring Tools*. ACM, 2011, pp. 33–36.

[21] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Int'l Conf. Quality of Information and Communications Technology (QUATIC)*. IEEE, 2010, pp. 106–115.

[22] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Proceedings of the International workshop on Principles of Software Evolution (IWPSE)*. ACM, 2007, pp. 31–34.

[23] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2004, pp. 223–232.

[24] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. IEEE Computer Society, 2011, pp. 181–190.

[25] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2009, pp. 75–84.

[26] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[27] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Software Eng.*, vol. 39, no. 8, pp. 1144–1156, 2013.

[28] A. F. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 306–315.

[29] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 682–691.

[30] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *Commun. ACM*, vol. 36, no. 11, pp. 81–94, Nov. 1993.

[31] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[32] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, ser. SBST '16. New York, NY, USA: ACM, 2016, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/2897010.2897016

[33] A. van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001, pp. 92–95.

[34] A. Garg, M. Gupta, G. Bansal, B. Mishra, and V. Bajpai, *Do Bad Smells Follow Some Pattern?* Singapore: Springer Singapore, 2016, pp. 39–46.

[35] F. A. Fontana, V. Ferme, and M. Zanoni, "Towards assessing software architecture quality by exploiting code smell relations," in *Software Architecture and Metrics (SAM), 2015 IEEE/ACM 2nd International Workshop on*, May 2015, pp. 1–7.

[36] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993, pp. 207–216.

[37] F. Palomba, A. D. Lucia, G. Bavota, and R. Oliveto, "Anti-pattern detection: Methods, challenges, and open issues," *Advances in Computers*, vol. 95, pp. 201–238, 2015.

[38] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025179

[39] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 4–15. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970340

[40] B. Cardoso and E. Figueiredo, "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: Information Systems: A Computer Socio-Technical Perspective - Volume 1*, ser. SBSI 2015. Porto Alegre, Brazil, Brazil: Brazilian Computer Society, 2015, pp. 46:347–46:354. [Online]. Available: http://dl.acm.org/citation.cfm?id=2814058.2814114

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[42] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 121–130. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332458

[43] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *Proc. 17th Asia Pacific Software Eng. Conf.* Sydney, Australia: IEEE, December 2010, pp. 336–345.

[44] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998.

[45] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study – replication package https://dibt.unimol.it/staff/fpalomba/reports/maltesque/," 2016.

[46] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2014. [Online]. Available: http://www.R-project.org/