Alternative Sources of Information for Code Smell Detection: Postcards From Far Away

Fabio Palomba
University of Salerno, Italy
fpalomba@unisa.it

***Abstract*—Code smells have been defined as symptoms of poor design and implementation choices. Previous studies showed the negative impact of code smells on the comprehensibility and maintainability of code. For this reasons, several detection techniques have been proposed. Most of them rely on the analysis of the properties extractable from the source code. In the context of this work, we highlight several aspects that can possibly contribute to the improvement of the current state of the art and propose our solutions, based on the analysis on how code smells are actually introduced as well as the usefulness of historical and textual information to realize more reliable code smell detectors. Finally, we present an overview of the open issues and challenges related to code smell detection and management that the research community should focus on in the next future.**

I. Context

In 1993, Ward Cunningham coined the methaphor of *technical debt* [1] to explain the compromise between delivering the most appropriate but still immature product in the shortest time possible [1]. One of the key factors contributing to technical debts are the so called *bad code smells* (a.k.a. *code smells* or simply *smells*), namely symptoms of the presence of poor design or implementation choices applied by programmers during the development of a software system [2]. For instance, a *Long Method* represents a method that implements a main functionality together with auxiliary functions that should be managed in different methods. Such methods can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs or to add new features.

In the past and, most notably, in recent years, several studies have been carried out in order to study the relevance of code smells from developers' perspective [3], [4], their evolution and longevity in real software systems [5], [6], and, more importantly, their impact on non-functional properties of source code [7]–[10]. Specifically, previous work empirically demonstrated that classes involved in design flaws are more change- and fault-prone [7], other than less comprehensible [8] and less maintainable than *non-smelly* classes [9], [10].

For all the aforementioned motivations, the research community has been particularly active in the definition of approaches able to promptly detect portion of source code affected by a smell [11]–[13]. Most of them proposed the identification of design flaws by (i) characterizing their key *symptoms* using a set of thresholds based on the measurement of structural metrics (e.g., if lines of code $\geqslant k$), and (ii) combining the identified symptoms, leading to the final rule for detecting the smells [11]–[13]. All these techniques mainly differ in the set of structural metrics used for the detection — which depends on the type of smell to identify — and by how the *symptoms* identified are combined. For example, a combination can be performed using AND/OR operators, such in the case of the detection strategies proposed by Marinescu [11]. In this context, Moha *et al.* [12] introduced DECOR, a method for specifying and detecting smells using a Domain-Specific Language (DSL). Specifically, DECOR detects four types of smells, i.e., *Blob*, *Swiss Army Knife*, *Functional Decomposition*, and *Spaghetti Code*.

Code smells can be also identified through the analysis of the portions of code that need to be refactored. As an example, Tsantalis *et al.* [13] defined JDeodorant, a tool able to detect instances of the *Feature Envy* smell by analyzing the source code in order to suggest where to apply operations of *Move Method* refactorings. Following a similar phylosophy, Bavota *et al.* proposed the use of *Relational Topic Modeling* to suggest operations of *Move Method* [14] refactoring able to remove the *Feature Envy* smell. The problem of smell detection has been also formulated as an optimization problem, leading to the usage of search algorithms to solve it [15]. The basic idea is to use genetic algorithms to detect smells following the assumption that what significantly diverges from good design practices is likely to represent a design problem.

II. Research Statement

Although existing approaches exhibit good detection accuracy, during our research we highlighted several aspects that can be improved, summarizable as follow:

1) *Previous research is mainly based on common wisdom rather than be guided by the state-of-the-practice*. Indeed, the smell detectors proposed in literature do not consider the circumstances that could have caused the smell introduction, thus providing a detection only based on structural properties characterizing the current version of a system. The misalignment between theory and practice can cause an additional problem: smells detectable by using current approaches might be far from what developers actually perceive as design flaws [3], thereby leading developers to not refactor smells [16], [17]. In our opinion, to better support developers in planning actions to improve source code quality, it is imperative to have a contextualized understanding of the circumstances under which particular smells occur.

2) *The proposed techniques still might not be adequate for detecting many of the smells described by Fowler* [2]. In particular, while there are smells where the use of structural analysis is suitable (*e.g.,* the *Complex Class* smell, a class having a high cyclomatic complexity), there are also several other design flaws not characterized by structurally visible problems. For example, a *Divergent Change* occurs when a class *is changed in different ways for different reasons* [2]. In this case, the smell is intrinsically characterized by *how source code changes over time* and, therefore, the use of historical information may help in its identification. At the same time, existing approaches do not take into account the vocabulary of the source code, that can be more effective in the identification of poorly cohesive or more complex classes, as already pointed out in previous research related to other software engineering tasks [18], [19]. For instance, a *Long Method* might be effectively identified by analyzing the textual scattering in the source code of a method.

Based on these observations, our research has the goal to address the following research questions:

- **RQ$_1$:** *When and why code smells are actually introduced by developers?*
- **RQ$_2$:** *How do approaches based on alternative sources of information, such as the historical and the textual one, perform in detecting code smells?*
- **RQ$_3$:** *Are code smells detectable using alternative sources of information more close to developers' perception of design problems?*

The final goal is to provide developers with more usable detectors, able to (i) accurately detect design flaws taking into account the way smells are generally introduced in the source code, and (ii) propose recommendations that are closer to the developers' perspective.

III. Research Results

The research conducted so far achieved the results reported in the following.

A. When and Why Code Smells are Introduced?

We investigated how code smells are introduced by setting up a large-scale empirical study conducted on the change history of 200 open source projects belonging to three software ecosystems, namely Android, Apache and Eclipse. The study aimed at investigating (i) *when* smells are introduced in software projects, and (ii) *why* they are introduced (*i.e.* under what circumstances smell introductions occur and who are the developers responsible for introducing smells) [20]. To address these research questions, we developed a metric-based methodology for analyzing the evolution of code entities in change histories of software projects to determine when smells start manifesting themselves and whether this happens suddenly (*i.e.*, because of a pressure to quickly introduce a change), or gradually (*i.e.*, because of medium-to-long range design decisions). We mined over 0.5M commits and we manually analyzed 9,164 of those that were classified as *smell-introducing*. The results achieved allowed us to report quantitative and qualitative evidence on when and why smells are introduced in software projects as well as implications of these results, often contradicting common wisdom.

In particular, we found that most of the smells are introduced when the (smelly) code artifact is created in the first place, *and not as the result of maintenance and evolution activities performed on such an artifact*. Moreover, code artifacts becoming smelly as consequence of maintenance and evolution activities are characterized by *peculiar metrics' trends*, different from those of clean artifacts. These results encourage the development of techniques able to recommend smells by using information about the changes applied on code artifacts. As for the motivations behind smells appearence, we found that they are generally introduced by developers when they work on the implementation of new features or enhancing existing ones, even if *we found almost 400 cases in which refactoring operations introduced smells*. Finally, newcomers are not necessary responsible for introducing bad smells, while developers with high workloads and release pressure are more prone to introducing them.

B. Using Alternative Sources of Information for Smell Detection

To evaluate the suitability of alternative sources of information in the context of smell detection, in our research we developed two approaches relying on historical information and textual analysis, coined as **HIST** (**H**istorical **I**nformation for **S**mell de**T**ection) [21], [22] and **TACO** (**T**extual **A**nalysis for **C**ode smell detecti**O**n) [23], respectively. In the following subsections, we report a sketch of the features of the techniques, as well as of the results achieved in the empirical studies conducted to evaluate them.

HIST: The approach detects smells based on change history information mined from versioning systems, and, specifically, by analyzing co-changes occurring between source code artifacts [21], [22]. We instantiated HIST for detecting five types of smells. Three of them—*Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance*—are symptoms that can be intrinsically observed from the project's history. The remaining two—*Blob* and *Feature Envy*—are smells that are traditionally detected using structural information (e.g., [12], [13]). However, we conjectured that in these cases the historical information can help in capturing complementary, additionally useful properties with respect to structural analysis.

To gather historical information from a versioning system such as `Git`, HIST includes the `Change History Miner` developed in the context of the MARKOS European project[1], a tool that is able to extract *fine-grained* changes, i.e., the changes occurring to methods in every single commit of the software system under analysis. In particular, the tool (ii) downloads the change history log of a system, and (ii)

[1] http://markosproject.sourceforge.net

analyzes each pair of subsequent commits in order to identify which methods have been changed from a commit to another. The set of fine-grained changes computed by the `Change History Miner` is provided as an input to the `Code Smell Detector`, that identifies the list of code components (if any) affected by specific smells. While the exploited underlying information is the same for all target smells (i.e., the change history information), HIST uses custom detection heuristics for each smell. For sake of space limitation, in this paper we only provide indication about how our approach detects the previously mentioned *Divergent Change* smell.

Given the definition of this smell, our conjecture is that *classes affected by Divergent Change present different sets of methods each one containing methods changing together but independently from methods in the other sets*. The *Code Smell Detector* mines association rules [24] for detecting subsets of methods in the same class that often change together. Association rule discovery is an unsupervised learning technique used for local pattern detection highlighting attribute value conditions that occur together in a given dataset [24]. In HIST, the dataset is composed of a sequence of change sets—e.g., methods—that have been committed (changed) together in a version control repository. An association rule, $M_{left} \Rightarrow M_{right}$, between two disjoint method sets implies that, if a change occurs in each $m_i \in M_{left}$, then another change should happen in each $m_j \in M_{right}$ within the same change set. The strength of an association rule is determined by its support and confidence [24]:

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

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

where T is the total number of change sets extracted from the repository. In our approach, we perform association rule mining using a well-known algorithm, namely *Apriori* [24]. Note that, minimum *Support* and *Confidence* to consider an association rule as valid can be set in the *Apriori* algorithm. Once HIST detects these change rules between methods of the same class, it identifies classes affected by *Divergent Change* as those containing at least two sets of methods with the following characteristics:

1) The cardinality of the set is at least γ;
2) All methods in the set change together, as detected by the association rules; and
3) Each method in the set does not change with methods in other sets as detected by the association rules.

We have evaluated HIST in two empirical studies. In the first the goal was to evaluate HIST detection accuracy in terms of two well-known and widely-used Information Retrieval metrics, namely precision and recall [25], against a manually-produced oracle. Furthermore, whenever possible, we also compared HIST with results produced by approaches that detect smells by analyzing structural properties of the source code, such as JDeodorant [13] (for the *Feature Envy* smell) and our re-implementations of the DECOR's [12] detection rules (for the *Blob* smell) and of the approach by Rao *et al.* [26] (for Divergent Change and Shotgun Surgery). The results indicate that HIST's precision is between 72% and 86%, and its recall is between 58% and 100%. When comparing HIST to alternative approaches, we observed that HIST tends to provide better detection accuracy, especially in terms of recall, since it is able to identify smells that other approaches are not able to capture. Moreover, we observed a strong complementarity of the structural-based approaches with respect to HIST, suggesting that even better performances can be achieved by combining these two complementary sources of information.

In the second empirical study we evaluated whether our approach is actually able to provide recommendations about code design problems that are recognized as such by software developers. For this reason, we surveyed twelve developers in order to investigate to what extent the smells detected by HIST (and by the alternative structural techniques) reflect developers' perception of poor design and implementation choices. The results of this second study highlight that over 75% of the smell instances identified by HIST are considered as design/implementation problems by developers, that generally suggest refactoring actions to remove them.

TACO: The approach [23] is able to identify five types of smells related to promiscuous responsibilities, i.e., *Long Method*, *Blob*, *Promiscuous Package*, *Feature Envy*, and *Misplaced Class*, using a three-step process, i.e., (i) textual content extraction, (ii) application of IR normalization process, and (iii) application of specific heuristics in order to detect the target smells. In the first step TACO extracts all textual elements needed for the textual analysis process of a software project, i.e., source code identifiers and comments. Then, the approach applies a standard IR normalization process [25] aimed at (i) separating composed identifiers, (ii) reducing to lower case letters the extracted words, (iii) removing special characters, programming keywords and common English stop words, and (iv) stemming words to their original roots via Porter's stemmer [27]. Thus, the smell detection process relies on Latent Semantic Indexing (LSI) [28], an extension of the Vector Space Model (VSM) [25], that models code components as vectors of terms occurring in a given software system. LSI uses Singular Value Decomposition (SVD) [29] to cluster code components according to the relationships among words and among code components (co-occurrences). The original vectors (code components) are then projected into a reduced *k* space of concepts to limit the effect of textual noise. To this aim, TACO uses the well-known heuristic proposed by Kuhn *et al.* [30], i.e., $k = (m \times n)^{0.2}$ where m denotes the vocabulary size and n denotes the number of documents (code components). Finally, smells are detected by measuring the lack of textual similarity among their constituent code components (e.g., vectors) using the cosine distance. For example, a *Blob* is detected (i) by computing the average similarity among the methods of the class, which correspond to the textual cohesion of a class defined by Marcus and

Poshyvanyk [18]; and (ii) by applying the following formula in order to calculate the probability P_B that a class is affected by the *Blob* smell:

$$P_B(C) = 1 - ClassCohesion(C) \quad (1)$$

where $ClassCohesion(C)$ represents the textual cohesion of the class C [18]. For all the types of smells, TACO outputs a value ranging in $[0; 1]$, which indicates the probability of a code component to be affected by a specific smell.

TACO has been evaluated in an empirical study involving 10 systems where our aim was to (i) evaluate the accuracy of TACO in the detection of smells, and (ii) compare our approach with state-of-the-art structural-based detectors, namely DECOR [12] for the *Blob* smell, JDeodorant [13] for the *Feature Envy* smell, and the approaches proposed in [31] and [32] for the *Promiscuous Package* and *Misplaced Class* smells, respectively. The results of our study indicate that the precision of TACO ranges between 67% and 77%, while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting, also in this case, that better performance can be achieved by combining the two sources of information.

IV. Open Issues and Challenges for the Future

The issues related to the management of code smells have attracted an ever increasing attention by the research community, interested in studying the dynamics behind code smell introduction and evolution as well as of their detection and removal. Although several steps ahead have been done in recent years, there are still a number of problems which preclude the transfer of such concepts in industry.

Open Issue #1: The techniques able to discover the presence of code smells are not perfect. As highlighted in our work, different techniques capture different smells using different types of information [22], [23]. Thus, to obtain a technique that significatively improves the current state-of-the-art smell detectors, a combination of different sources of information is needed. However, such a combination is not trivial. For example, during the development of TACO [23], we evaluated a simple combination between textual and structural information obtained using AND/OR operators for the detection of the *Feature Envy* smell: in the AND case we experienced a strong increase of the precision (i.e., +17% than TACO, +27% than JDeodorant), accompanied by a strong decreases of the recall (-34% than TACO, -31% than JDeodorant). Similarly, in the OR case the recall strongly increases (+24% than TACO, +34% than JDeodorant), while the precision decreases of almost 39% for TACO and 36% for JDeodorant). The construction of a hybrid technique is part of our future agenda.

Open Issue #2: The quality of the suggestions matters. As recently pointed out [16], [17], only a small percentage of code smells is actually removed by developers. In our opinion, the reason behind this data is twofold. First of all, since the removal of code smells is a time-consuming and error-prone task [33], it is important that such techniques find relevant suggestions about which parts of the source code a developer should care about. To this aim, the research community needs to focus its attention on how to rank code smells based on their importance for developers and/or the context a developer is working on. While some attempts in this direction have already carried out [34], [35], we believe that these aspects need to be still improved in next years. Secondly, as revealed in our research on how code smells are introduced [20], code components are generally affected by bad smells since their creation. This means that new recommenders implementing a *just-in-time* philosophy would be worthwhile. On the other hand, we found that there are also several cases in which code smells are introduced as consequence of several maintenance activities performed on a code artifact. In these cases, such code components are characterized by peculiar metrics' trends, different from those of clean artifacts. This implies the possibility to define a new generation of recommenders able to *predict* which classes will become smelly over time and, therefore, allow a more suitable way to manage code smells.

Open Issue #3: Improving the Usability of Code Smell Detectors. Detection tools might require the definition of several parameters. Thus, they might be hard to understand and to work with, making developers more reluctant to use such tools. In addition, it is necessary to define a good strategy for the visualization and the analysis of candidate smells. This issue is particular important since the smells identified by any detection tool need to be validated by the user. Thus, a good graphic metaphor is required to highlight problems to the developer's eye, allowing her to decide which of the code components suggested by the tool really represent design problems.

Open Issue #4: Smell detection is not just a problem for its own sake. Although part of the research community is still skeptical about the problem of code smell detection (even considering that smells are generally not removed), the information about the bad designed source code may be useful *not only* for detecting portion of code that should be refactored. Indeed, as demonstrated in our recent work, the quality of source code can be effectively used in different other software engineering tasks. For example, we demonstrated the usefulness of smell-related information for improving bug prediction performance [36], as well as in the context of automatic test case generation [37]. We believe that the research community should investigate these aspects more in depth.

Acknowledgements

The author is really grateful to his advisor Andrea De Lucia and his co-advisor Rocco Oliveto for all the help and support received during his PhD.

REFERENCES

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

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

[3] 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.

[4] 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.

[5] 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.

[6] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Proceedings of the International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 2010, pp. 106–115.

[7] F. Khomh, M. Di 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.

[8] 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.

[9] 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.

[10] 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.

[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] 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.

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

[14] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, July 2014.

[15] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. ACM, 2010, pp. 113–122.

[16] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *J. Syst. Softw.*, vol. 107, no. C, pp. 1–14, Sep. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2015.05.024

[17] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 24th International Symposium on the Foundations of Software Engineering*, 2016.

[18] A. Marcus and D. Poshyvanyk, "The conceptual cohesion of classes," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2005, pp. 133–142.

[19] R. Buse and W. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, July 2010.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[26] A. Rao and K. Raddy, "Detecting bad smells in object oriented design using design change propagation probability matrix," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, 2008, pp. 1001–1007.

[27] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[28] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, vol. 41, no. 6, pp. 391–407, 1990.

[29] J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*. Boston: Birkhauser, 1998, vol. 1, ch. Real rectangular matrices.

[30] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information & Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[31] M. Girvan and M. E. Newman, "Community structure in social and biological networks." *Proc Natl Acad Sci U S A*, vol. 99, no. 12, pp. 7821–7826, June 2002.

[32] D. Atkinson and T. King, "Lightweight detection of program refactorings," in *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, Dec 2005, pp. 8 pp.–.

[33] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, Sept 2012, pp. 104–113.

[34] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based code smells prioritization for prefactoring," in *Proceedings of the 24th International Conference on Program Comprehension (ICPC 2016)*. Austin, USA: IEEE, 2016, p. to appear.

[35] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers' context for automatic refactoring of software anti-patterns," *Journal of Systems and Software (JSS)*, 2016.

[36] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*. Raleigh, USA: IEEE, 2016, p. to appear.

[37] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 130–141. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931057