Ten Years of JDeodorant: Lessons Learned from the Hunt for Smells

Nikolaos Tsantalis
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
nikolaos.tsantalis@concordia.ca

Theodoros Chaikalis
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
chaikalis@uom.gr

Alexander Chatzigeorgiou
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
achat@uom.gr

***Abstract*—Deodorants are different from perfumes, because they are applied directly on body and by killing bacteria they reduce odours and offer a refreshing fragrance. That was our goal when we first thought about "bad smells" in code: to develop techniques for effectively identifying and removing (i.e., deodorizing) code smells from object-oriented software. JDeodorant encompasses a number of techniques for suggesting and automatically applying refactoring opportunities on Java source code, in a way that requires limited effort on behalf of the developer. In contrast to other approaches that rely on generic strategies that can be adapted to various smells, JDeodorant adopts ad-hoc strategies for each smell considering the particular characteristics of the underlying design or code problem. In this retrospective paper, we discuss the impact of JDeodorant over the last ten years and a number of tools and techniques that have been developed for a similar purpose which either compare their results with JDeodorant or have built on top of JDeodorant. Finally, we discuss the empirical findings from a number of studies that employed JDeodorant to extract their datasets.**

***Index Terms*—Code Smells, Refactoring, Object-Oriented Software**

I. Introduction

Over the last decade, JDeodorant contributed refactoring recommendation techniques for a variety of code smells [1], including Feature Envy [2], [3], State Checking and Type Checking [4], [5] Long Method [6], [7], God Class [8]–[10], Duplicated Code [11]–[13], and Refused Bequest [14]. Our main philosophy was to create a tool that can provide a holistic solution for the practice of refactoring by supporting all refactoring-related activities as defined by Mens and Tourwé [15], namely:

1) Identify where the software should be refactored.
2) Determine which refactoring(s) should be applied to the identified places.
3) Guarantee that the applied refactoring preserves behavior.
4) Apply the refactoring.
5) Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort).
6) Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests, etc.).

To this end, we gave great emphasis not only in the implementation of code smell detection techniques (to support activity #1 and #2), but also in the implementation of refactoring mechanics (to support activity #4), the implementation of refactoring preconditions, which ensure that all recommended refactorings will preserve program behavior (to support activity #3), the implementation of software metrics to preassess the impact of all recommended refactorings on design quality and rank them accordingly (to support activity #5), and finally the automatic update of Javadoc comments whenever possible (to support activity #6). At the same time, we tried to provide an easy-to-use and minimal user interface, an easy installation process with minimal configuration (JDeodorant is available as an Eclipse plug-in through Eclipse Marketplace[1], since November 2007), code smell visualizations, and video tutorials[2], as a means to make the tool more attractive to practitioners, researchers, and educators. JDeodorant was open-sourced in August 2014[3] (before that date its source code was available only for research groups and educators through an academic license agreement), and in June 2015 a command-line version[4] was released allowing to execute JDeodorant in headless mode without opening the Eclipse IDE.

Along the way, several research groups proposed and developed competitive code smell detection and refactoring recommendation techniques. Bavota et al. [16] provide a comprehensive coverage of the state-of-the-art in refactoring recommendation systems until 2013. Al Dallal [17] conducted a Systematic Literature Review including 47 primary studies identifying refactoring opportunities in object-oriented code, which were published before the end of 2013. In many cases, *JDeodorant was used as the state-of-the-art baseline to evaluate the accuracy of newly proposed techniques*.

Moreover, several empirical researchers studied the evolution trends of code smells, and the relation of code smells with software quality attributes, such as change-proneness, error-proneness, and maintainability. Al Dallal and Abdin [18] conducted a Systematic Literature Review including 76 pri-

[1]https://marketplace.eclipse.org/content/jdeodorant
[2]https://www.youtube.com/channel/UCp-NaYVqKOERLreXxwCgWzg
[3]https://github.com/tsantalis/JDeodorant
[4]https://github.com/tsantalis/jdeodorant-commandline

978-1-5386-4969-5/18/$31.00 © 2018 IEEE

mary studies investigating the impact of refactoring on several internal and external quality attributes, which were published before the end of 2015. In many cases, *JDeodorant was used to extract code smell datasets from open-source projects*.

Finally, *several research groups built their tools on top of JDeodorant, or used a static source code analysis feature offered by JDeodorant*. For instance, [19]–[21] built their tools on top of JDeodorant to implement automated refactoring to the Strategy, Null Object, and Template Method design patterns [22], respectively. The MUSE (Method USage Examples) tool [23], an approach for mining and ranking actual code examples that show how to use a specific method, used the static slicer provided by JDeodorant to compute intra-procedural, backward slices.

The public availability of JDeodorant and its continuous support and maintenance over the years made possible its comparison with competitive tools, as well as its use in various empirical studies. It is worth noting that in the meantime, the Java Language Specifications[5] were updated 3 times to add new language features, such as the varags parameter in method declarations (JLS7), type inference for generic instance creation (JLS7), try statement with resource management (JLS7), catching multiple exception types with a Union type (JLS7), strings in switch statement (JLS7), lambda expressions (JLS8), and module declarations (JLS9), for which JDeodorant was updated to support them.

In this paper, we examine the impact of JDeodorant on research and practice over the last ten years. In Section II, we investigate the evolution in the number of citations and applied refactorings by JDeodorant users. In Section III, we present competitive code smell detection and refactoring recommendation techniques that include a comparison with JDeodorant in their evaluation. In Section IV, we present the findings of empirical studies that used JDeodorant to extract code smells and refactoring opportunities from software systems. Finally, in Section V, we present a retrospective of our experience from the development and maintenance of JDeodorant, which can serve as a guide for graduate students and research teams interested in investing on the development of new tools.

II. Impact in Numbers

To demonstrate the impact of JDeodorant on research and practice, we use two proxies, namely the number of *citations*, as counted by Google Scholar, received by 12 papers related to JDeodorant features [2]–[12], [24], which were published between 2007 and 2015, and received at least 20 citations between 2011 and 2017, and the number of unique refactorings recommended by JDeodorant and actually applied by its users (*applications*). Whenever a user applies a refactoring suggested by JDeodorant, a key is created based on the signature of the code element that was refactored combined with a unique machine ID. If the same user re-applies the same refactoring (e.g., by undoing and redoing it), only the application timestamp will be updated for the previously recorded refactoring application.

[5]https://docs.oracle.com/javase/specs/

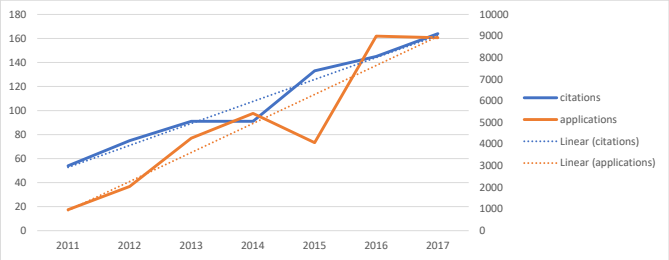

Fig. 1. Impact of JDeodorant.

Figure 1 shows the evolution of citations and applications over the period 2011-2017 (y-axis has two different scales). We are focusing on this particular period of time, because we started collecting usage information for JDeodorant in November 2010, and thus we have complete yearly information for both citations and applications only for the period 2011-2017.

We can observe from Figure 1 that there is an almost linear growth in the number of citations and refactoring applications. There is only a small discrepancy in the number of refactoring applications for year 2015, which is lower than the trendline. This was due to networking issues with the machine hosting the JDeodorant usage statistics database, and thus we believe a significant number of applied refactorings was not recorded due to server downtime.

The number of applied refactorings increased 9 times within a period of seven years (starting from 1,000 refactorings in 2011, and ending with 9,000 refactorings in 2017, i.e., 25 unique refactoring applications per day). It should be noted that the refactoring applications count in Figure 1 includes 4 different types of refactorings, namely Move Method (for Feature Envy), Extract Method (for Long Method), Extract Class (for God Class) and Replace Conditional Logic with Polymorphism (for State-checking and Type-checking), which were all offered by JDeodorant before 2011. Figure 2 shows the number of applied refactorings per refactoring type for the period 2011-2017.

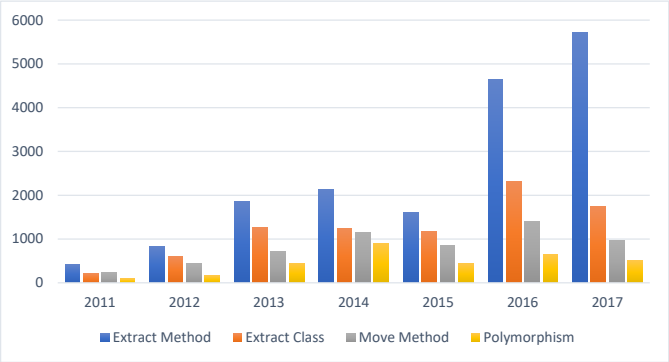

Fig. 2. Number of applied refactorings per refactoring type.

Out of the 35,000 refactorings applied in the period 2011-2017, 50% correspond to Extract Method, 25% correspond to Extract Class, 16% correspond to Move Method, and only 9% correspond to Replace Conditional Logic with Polymorphism refactorings. Despite the fact that Extract Method and Extract Class refactorings were introduced last in JDeodorant, we can see that they had a much greater user adoption than the

other two refactoring types that were introduced much earlier. There are two possible explanations for this phenomenon. First, there are significantly more opportunities for Extract Method and Extract Class refactorings compared to the other two refactoring types. Second, users might be more interested in decomposing long methods and large classes, than moving methods to other classes and replacing conditional logic with polymorphism. This finding is critical for researchers who are interested in developing new refactoring recommendation systems, because it can help them to focus their attention and effort on refactoring types that are considered more important by developers.

By performing an analysis on the package prefixes of the refactored code entities, we can observe that 22% of the applied refactorings take place in projects with `org.` prefix, 17.5% take place in projects with `com.` prefix, 5.5% take place in projects with `net.` prefix, and 3% take place in projects with `edu.` prefix. The remaining 50% of the applied refactorings correspond to projects, which have a country-based prefix, such as `de.` (2.2%), `ca.` (2%), `br.` (1.2%), or a project-specific prefix. These projects are a mix of commercial and educational projects. More detailed analysis about JDeodorant usage trends can be found at our statistics webpage[6].

With respect to the number of citations received by 12 papers related to JDeodorant, we can observe an almost perfect linear growth. The number of citations increased 3 times within a period of seven years (starting from 54 citations in 2011, and ending with 164 citations in 2017). This growth can be explained from the new JDeodorant articles that were published in or after 2011 (three were published in 2011, and three were published after 2011). Typically, it takes 1-2 years for a paper to receive its first citations. The most cited paper is [3] with 200 citations in total, which steadily receives around 30 citations per year, since 2013 (4 years after its publication). It should be noted that the tool demonstration papers [2], [4], [9] receive a gradually increasing number of citations over the last 3 years (ranging from 10 to 25 citations per year). This perhaps shows a preference of researchers to cite a tool demonstration paper as representative publication for a research work, when using or referring to a tool. Another important note is that as a tool gets more established, there is an increasing number of citations to the tool's website either in the form of footnotes or references in the citing papers. According to Google Scholar, there are 60 documents citing www.jdeodorant.com, 12 documents citing www.jdeodorant.org, and 18 documents citing marketplace.eclipse.org/content/jdeodorant. Although it is difficult to assess if these documents cite one of the published JDeodorant papers as well, we can expect that some of them cite only the tool's website. This phenomenon demonstrates the difficulties in assessing a tool's impact from the academic point of view (i.e., citation count), because several works, which are clearly impacted by a tool, but do not cite a related paper, cannot be tracked by citation analysis engines.

[6]https://users.encs.concordia.ca/~nikolaos/stats.html

III. Competitive Works

In this section, we present the tools and techniques that were published concurrently with, or after JDeodorant, up to date. The majority of these works, include a comparison with JDeodorant in their evaluation. Figure 3 shows the most important milestones in the evolution of code smell detection and refactoring recommendation systems, up to date. These milestones refer to novel sources of information and novel methods/algorithms that have not been used before in the context of code smell detection and refactoring recommendation. It should be noted that JDeodorant relies solely on structural information for the detection of refactoring opportunities.

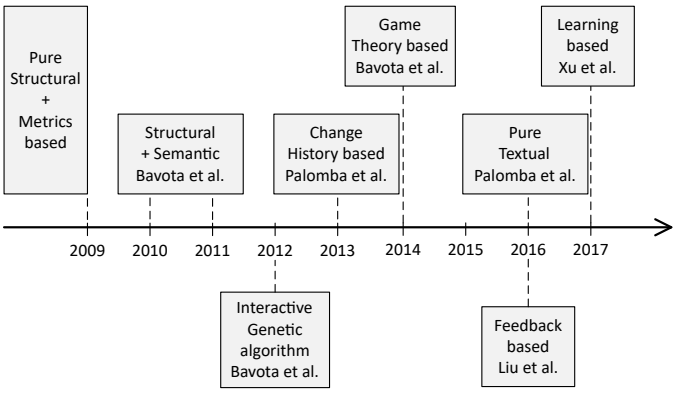

Fig. 3. Important milestones in the evolution of code smell detection and refactoring recommendation systems.

A. Move Method Refactoring

MethodBook [25] is the first work to combine structural and semantic (i.e., textual) information to identify Move Method refactoring opportunities and remove the Feature Envy bad smell from source code. In addition, it was also the first work based on *Relational Topic Models* (RTM), a hierarchical probabilistic model for representing and modeling topics, documents, and known relationships among these. The advantage of RTM is that it considers both document context and links among the documents, while other topic modeling techniques, such as *Latent Dirichlet Allocation* (LDA) or *Latent Semantic Indexing* (LSI), only consider textual information from the documents to model. The evaluation on 6 open-source systems showed that MethodBook always outperforms JDeodorant on the improvement of semantic metrics, while it is able to perform better than JDeodorant only on three systems regarding the improvement of structural metrics. Moreover, the evaluation performed with developers on their own systems, revealed that developers are more willing to apply the refactorings recommended by MethodBook compared to those recommended by JDeodorant.

HIST [26], [27] (Historical Information for Smell deTection) is the first code smell detection approach to employ change history information to detect instances of five different code smells, namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. HIST relies on association rule discovery to find classes/methods frequently

co-changing in the commit history of software systems. HIST was evaluated on 9 projects belonging to the Apache ecosystem, 5 projects belonging to the Android APIs, and 6 other open-source systems, including Eclipse Core, Google Guava, jEdit, and MongoDB. With respect to Feature Envy code smell, HIST achieved a precision of 78% and a recall of 77%, while JDeodorant achieved a precision of 65% and a recall of 71%. Moreover, there was a 54% overlap between the Feature Envy code smells detected by HIST and JDeodorant.

JMove [28], [29] compares the similarity of the dependencies established by a method with the dependencies established by the methods in possible target classes, in order to detect Move Method refactoring opportunities. JMove is the first approach that uses type reference information (i.e., declaring class of method call, accessed field type, local variable declaration type, instantiated object type, exception type, method return type, and annotation type) to determine the similarity of two method declarations. The precision and recall was evaluated on 10 synthesized versions of open-source systems, in which random methods were moved to random classes. A recommendation system is supposed to suggest the inverse move of these randomly moved methods. JMove precision ranges from 21% (small methods) to 32% (large methods) and its median recall ranges from 21% (small methods) to 60% (large methods), while JDeodorant has a maximum precision of 15% (large methods) and a maximum median recall of 40% (small methods).

TACO [30], [31] is the first pure textual analysis approach for code smell detection. It is able to detect a family of smells of different nature and different levels of granularity, such as Long Method, Feature Envy, Blob, Promiscuous Package and Misplaced Class. TACO relies on *Latent Semantic Indexing* (LSI), which models code components as vectors of terms occurring in a given software system. The textual similarity among software components is measured as the cosine of the angle between the corresponding vectors. The precision and recall of TACO was evaluated on 10 open-source projects. For Feature Envy code smell, TACO achieved a precision of 67% and a recall of 72%, while JDeodorant achieved a precision of 57% and a recall of 69%. Moreover, there was a 48% overlap between the Feature Envy code smells detected by TACO and JDeodorant.

RefactoringNavigator [32] is an interactive recommendation system for aiding architectural refactoring. It takes a given implementation as the starting point, a desired high-level design as the target, and iteratively recommends a series of refactoring steps. Moreover, it allows the user to accept, reject, or ignore a recommended refactoring step, and uses the users feedback in further refactoring recommendations. To evaluate RefactoringNavigator, Lin et al. designed a controlled experiment in which the experimental group (EG) used all the features of RefactoringNavigator, while the control group (CG) used a customized version without the recommendation function and JDeodorant. The goal of the experiment was to assess whether RefactoringNavigator can help developers perform architectural refactoring faster and more correctly. On average, the experimental group accomplished the refactoring task much faster (13.270m for EG vs. 58.619m for CG), more accurately (all 50 test cases passed for EG vs. 36.44 for CG), and with much less effort (6.0 LOC for EG vs. 354.1 LOC for CG).

DominoEffect [33] is a new approach to identify moving method opportunities based on conducted move method refactorings. Whenever a method is moved from source class to another destination class, the approach looks for methods within the source class that may also need to be moved to the destination class. The rationale behind DominoEffect is that if two methods are strongly coupled and closely related in business logic, when one of them is moved the other may need to be moved as well. The evaluation has shown that DominoEffect is accurate in recommending methods to be moved (average precision 76%) and in recommending destinations for such methods (average precision 83%), and surpasses the precision of JDeodorant.

Liu et al. [34] proposed the first feedback-based code smell detection technique. It extends the traditional metric-based code smell detection with a *threshold adaption* phase, which collects feedback from the developers based on the code smells they decide to refactor or skip, and adjusts automatically the threshold setting of the metric-based rule used for detecting code smells. To evaluate the proposed technique Liu et al. created an oracle of code smells by asking three engineers to manually examine the methods of five open-source systems exceeding a conservative threshold (in order to avoid examining all methods), and assess whether these methods are involved in code smells. They concluded that the proposed approach outperforms the *tuning machine* approach, which infers the optimal threshold setting from a fixed set of reference applications.

c-JRefRec [35] identifies Move Method refactoring opportunities based on four heuristics using static and semantic program analysis. c-JRefRec employs a semantic analysis to identify move method refactoring candidates by extracting all code identifiers including names of packages, classes, methods, attributes, and parameters for each class as well as the method to be moved. Then, it computes the cosine similarity between a method and a class, using tf-idf vectors, where methods and classes are regarded as individual documents. The evaluation on 3 open-source systems showed that c-JRefRec achieves an average precision for detecting Feature Envy code smells of 0.48 and an average recall of 0.73, while JDeodorant achieves an average precision of 0.38 and an average recall of 0.25. Regarding the accuracy of Move Method refactoring recommendations, c-JRefRec achieves an average precision of 0.42 and an average recall of 0.68, while JDeodorant achieves an average precision of 0.38 and an average recall of 0.25.

B. Extract Method Refactoring

JExtract [36] determines the statements to be extracted from a method based on the similarity of their dependency sets. In particular, JExtract considers variable, type, and package dependency sets extracted from each individual statement.

JExtract limits the code fragments suggested for extraction to complete blocks of code and continuous statements. To evaluate the accuracy of JExtract, Silva et al. created an oracle by randomly inlining methods in two open-source projects. A recommendation system is supposed to suggest the extraction of these randomly inlined methods. JExtract achieved a precision of 48% for project JUnit and 38% for project JHotDraw.

SEMI [37] is an approach that aims at identifying source code chunks that collaborate to provide a specific functionality by calculating the cohesion between pairs of statements. Two statements are characterized as coherent, if they access the same variable, if they call a method for the same object, and if they call the same method for a different object of the same type. Based on this definition, SEMI identifies all possible sets of successive statements that are coherent to each other, regardless of their size. SEMI was evaluated against JDeodorant and JExtract using as benchmarks the original studies in which JDeodorant and JExtract have been evaluated. The results showed that SEMI presents the most accurate approach in terms of F-measure, whereas JDeodorant in terms of precision and JExtract in terms of recall, when taking into account all methods in the benchmarks. By focusing only on methods with more than 30 lines of code, SEMI presents the best precision, recall, and F-measure.

GEMS [38] is the first learning-based approach for recommending Extract Method refactorings. GEMS extracts structural and functional features, which encode the concepts of complexity, cohesion and coupling in a probabilistic model. Then, this model is trained to extract suitable code fragments from a given source of a method. To evaluate GEMS, Xu et al. created two training sets. The first set includes 267 Extract Method refactorings from open-source projects that were confirmed from the developers who applied them [39]. The second set includes 5598 inlined methods that were invoked only once in the source code of the projects. Next, they compared the accuracy of GEMS against JExtract, SEMI, and JDeodorant. With respect to recall, GEMS achieves slightly better results compared to JExtract. With respect to precision, GEMS achieves a 5-7% improvement over the tool with the second highest precision, namely JDeodorant. When focusing only on methods with more than 30 lines of code, SEMI has the best precision and recall at 1% and 2% tolerance levels, while GEMS has the best precision and recall at 3% tolerance level. Tolerance is a permissible limit of variation in lines of code when evaluating the correctness of an Extract Method candidate.

C. Extract Class Refactoring

ARIES [40]–[43] was among the first recommendation systems for Extract Class refactoring, and was developed concurrently with JDeodorant.

The approach presented in [41] represents a candidate class for refactoring as a graph, where the nodes correspond to the methods of the class, and the edges connect all the pairs of methods of the class. The edges have weights representing the structural and semantic (i.e., textual) similarity of the connected methods. Then, a MaxFlow-MinCut algorithm is applied on the graph in order to split the class in two classes ensuring that the number of dependencies between the two extracted classes is low (due to the min cut).

The approach presented in [43] constructs a $n \times n$ matrix, called method-by-method matrix, where n is the number of methods in the class to be refactored. A generic entry $c_{i,j}$ of the *method-by-method* matrix represents the likelihood that method m_i and method m_j should be in the same class. This likelihood is obtained combining three different (structural and semantic) measures, i.e., Structural Similarity between Methods (SSM), Call-based Interaction between Methods (CIM), and Conceptual Similarity between Methods (CSM). The extraction of chains of methods is obtained computing the transitive closure of the *method-by-method* matrix.

The approach was evaluated with two empirical studies. In the first study, Bavota et al. asked 50 Master's students to rate the refactoring solutions suggested by ARIES on existing Blobs identified in two open-source systems. They also evaluated the impact of the refactoring operations proposed by ARIES on the cohesion and coupling of the object systems. In the second study, they identified and selected 11 classes in different versions of open source systems that actually underwent extract class refactoring by their original developers. Then, they asked 15 Master's students to refactor these classes and compared the refactorings proposed by ARIES and the refactorings performed by the students with the refactorings performed by the original developers. The results show that the refactoring solutions proposed by ARIES (i) strongly increase the cohesion of the refactored classes without leading to significant increases in terms of coupling; (ii) are considered useful by developers performing extract class refactoring; and (iii) are able to approximate manually performed refactorings at 91%, on average. In addition, a comparison with the former approach [41] showed that ARIES outperforms the MaxFlow-MinCut algorithm.

Bavota et al. [44] proposed the first approach based on game theory to identify refactoring solutions that provide a compromise between the desired increment in cohesion and the undesired increment in coupling. The Extract Class Refactoring (ECR) problem is modelled as an iterative multi-round game between n players. Each player represents a class to be extracted from a given class and seeks to maximize its cohesion while maintaining its coupling as low as possible. The players begin the refactoring process with one seed method each, taken from the original class to be refactored; then, they contend for the remaining methods to create new classes. The chosen methods correspond to the Nash equilibrium computed on a payoff matrix that uses similarity measures between methods to assess the effect of the players' choice on the cohesion and coupling of their classes. The candidate number of players and a candidate seed method for each player are determined by a heuristic based on the analysis of the topics captured in the source code of the class to be refactored, using *Latent Dirichlet Allocation* (LDA). The evaluation of

the approach was conducted on two cases studies. In the first case study, classes with high cohesion from three open-source systems have been merged to create artificial Blobs and then refactored with the aim of reconstructing the original classes. In the second case study, seven Blobs of an open-source system, namely GanttProject, were refactored to evaluate the usefulness of the proposed approach for actual developers. The case study results showed that the proposed approach can refactor Blob classes into new meaningful classes with higher cohesion and marginal increment of coupling.

MethodSimilarity [45] is a novel approach for identifying and decomposing class responsibilities using method similarity based on both the internal and external class relationships. It measures method similarity for four types of the internal method relationships (internal attribute sharing, internal direct call dependency, internal indirect call dependency, and internal method sharing), and two types of the external method relationships (external indirect call dependency, and external call dependency). The candidate classes for refactoring are decomposed based on measured similarity using hierarchical agglomerative clustering, which is a widely used clustering method for its flexibility in determining the level of granularity and clustering basis (e.g., method similarity). Lee et al. evaluated MethodSimilarity on three open-source systems, namely JMeter, JHotDraw, and ArgoUML. First, they asked a group of experienced software engineers to evaluate the results of MethodSimilarity on classes haivng multiple responsibilities. Second, they introduced classes simulating multi-responsibility classes by merging single responsibility classes, and evaluated how precisely MethodSimilarity can decompose the merged classes into the original classes. The results showed that MethodSimilarity (with an average F-measure of 79.53%) outperforms the distance-based approach used by JDeodorant in [10] (with an average F-measure of 69.88%) and the text-based conceptual similarity approach used by ARIES in [43] (with an average F-measure of 47.89%).

ExtC [46] applies various clustering approaches for the detection of Extract Class refactoring opportunities. The first approach is Betweenness clustering, i.e., the clustering is based on the analysis of communication patterns within a class. The second approach is Dual clustering, i.e., a graph-based divisive clustering based on structural information, followed by an agglomerative clustering based on semantic information. The first clustering algorithm splits the members of the class into at least two clusters. Afterwards, the second clustering algorithm connects any additional small clusters with the two largest clusters. The evaluation has shown that dual clustering enabled the refactoring of more classes than betweenness clustering. On the other hand, based on the average cohesion scores of the modified and extracted classes, betweenness clustering generally produced more cohesive classes than dual clustering.

IV. Empirical Evidence from the Use of JDeodorant

Various empirical studies relying on code smell identification techniques and tools have been presented in the literature with a diverse set of research goals. Broadly speaking, such studies aimed at analyzing: a) the relation between smells and other software qualities (internal or external) and b) properties of the smells themselves (e.g. diffuseness, persistence, ability to be detected, etc.) including their perception by development teams. Next, we focus on studies that have used JDeodorant (either the tool as is, or its individual components) for the creation of their datasets.

An overview of the findings and the context in which empirical studies have been performed is provided in Table I. The landscape of empirical studies that have been performed with JDeodorant is visually depicted in Figure 4.

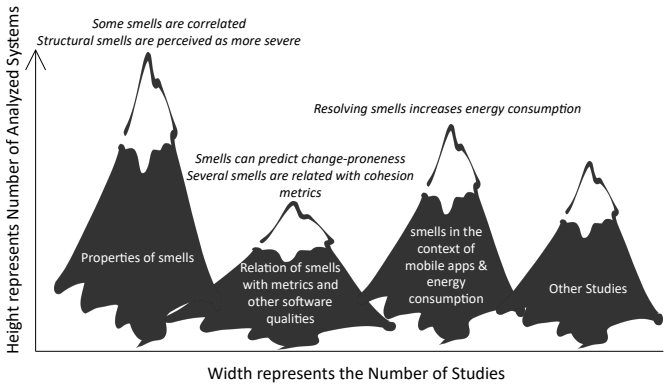

Fig. 4. Landscape of empirical studies on smells using JDeodorant (summits refer to representative findings).

Most techniques for code smell detection are based on the analysis of structural properties of source code. However, textual techniques that rely on the analysis of textual content of methods and statements can be very effective, especially for certain smells. For example, a Blob can be detected by computing the textual cohesion reflected in the lexical similarity among methods of a class. Palomba et al. [31] conducted a repository mining study and a user study to understand how code smells detected using textual analysis are perceived and refactored by developers. The former study revealed that the intensity of textual code smells tends to decrease over time in contrast to structural code smells whose intensity increases. The responses by industrial developers in the latter study revealed that textual code smells are perceived as actual design problems and therefore tend to be refactored over time. On the other hand, structural code smells are often perceived as more severe.

The co-existence of smells has also been the focus of research efforts since evidence on the correlation among smells can assist both their detection as well as their refactoring. Lozano et al. [47] employed JDeodorant for the identification of Feature Envy, God Class, Long Method and Type Checking bad smells in several releases of three open-source applications. The findings confirm that correlations exist: for example, almost all methods suffering from Feature Envy reside in Long Methods and Feature Envy methods were very often located in God Classes.

Some of the most prominent approaches for smell detection rely on the use of metrics. However, the empirical validation

TABLE I
EMPIRICAL EVIDENCE RELYING ON JDEODORANT

Finding	Context	Support
a. Textually detected smells are easier to identify and refactor b. Structural smells are often perceived as more severe	20 OSS systems 19 developers and 5 quality experts	[31]
a. Strong correlation between Feature Envy and Long Method b. Mild correlation between Long Method and God Class c. Mild correlation between Feature Envy and God Class	3 OSS systems	[47]
One size (LOC) and three cohesion metrics (LCOM1, LCOM2 and Coh) can accurately predict Long Method smells	4 OSS systems	[48]
a. Extract method refactoring has a positive impact on software metrics except for LCOM b. All refactorings have a positive impact on Tight Class Cohesion	1 OSS system	[49]
Code smells form better predictors of change-proneness compared to object-oriented software metrics.	1 OSS application	[50]
Refactoring application in mobile apps increases energy consumption from 8% to 70%	4 mobile apps	[51]
Long Method and Type Checking code smells are twice as likely to occur in mobile applications	14 mobile apps	[52]
Refactoring application in standalone apps increases energy consumption	3 OSS systems	[53]
Code smells (except for "Duplicated Code") are not influenced by the underlying application domain	118 Java applications from 6 domains	[54]
a. Extract Method refactoring has a negative effect on traceability b. Move Method refactoring does not affect requirements traceability	1 proprietary & 2 OS applications	[55]
Approximately 42-44% of Self-Admitted Technical Debt incurs positive interest	One Apache Project	[56]
Comments that imply design problems constitute the majority (42%-84%) of self-admitted technical debt	5 OSS projects	[57]
Code examples illustrating the use of specific methods can be automatically selected and ranked	6 libraries	[23]

of these metrics as reliable indicators for the existence of smells is rather limited. Charalampidou et al. [48] empirically explored the ability of size and cohesion metrics to predict the existence and the intensity of long method occurrences. JDeodorant was used to identify Extract Method refactoring opportunities in a study on 4 OSS systems. The results suggest that one size and three cohesion metrics are capable of predicting Long Method issues, confirming the intuitive connection among cohesion and the presence of long methods.

Apart from the use of metrics as a means of identifying smells, they can also be used to assess the impact of refactoring applications for smell removal on the design quality of a software system. This has been the goal of an empirical study by Fontana and Spinelli [49] where 6 cohesion, coupling and complexity metrics have been used to assess the impact of refactorings on an OSS system. JDeodorant was one of the three employed tools (JDeodorant, PMD, InFusion) to identify Feature Envy, Large Class, Long Method and Shotgun surgery smells, and one of the two tools (JDeodorant and IntelliJ IDEA) to perform automatic refactoring. The results showed that all aforementioned refactorings have a positive impact on cohesion as captured by the Tight Class Cohesion metric. Furthermore, applying the Extract Method refactoring to resolve Long Method smells appears to have a positive impact on all metrics except for LCOM.

Change proneness is another major indicator of design quality. A software system that demands extensive modifications during new feature additions, witnesses poor design quality. To this end, Kaur et al. [50], studied the relationship between code smells and metrics with change-prone classes in an attempt to find the optimal predictor of change proneness. They used 10 code smells as predictors, including 3 classic smells (Long Method, Feature Envy and God Class) that were identified by JDeodorant and 7 exception handling smells. The results suggest that code smells form better predictors of change-proneness compared to object-oriented software metrics.

Although code and design quality have been mostly studied in the context of desktop and enterprise applications, maintainability is a concern for software in all domains, including mobile applications. However, in such application areas, runtime qualities, such as energy consumption, matter most. The trade-off between object-oriented design quality, as reflected in the number of smells and energy consumption has been the focus of the study by Rodriguez et al. [51]. In particular, the authors studied on four mobile applications the energy impact of refactorings on Java source code and JDeodorant has been used to identify refactoring opportunities. The results confirm the findings of previous studies showing that assuring important features of object-oriented design through removing bad smells increases energy consumption, sometimes as much as 70%.

The tremendous growth of the mobile application industry and the extreme frequency of updates, call for increased attention to design and code quality. Moreover, mobile applications have shorter software delivery cycles in order to remain competitive and therefore maintainability issues form a crucial factor for their viability. In this context, Verloop [52] employed JDeodorant in order to identify possible differences in code smell density between mobile and non-mobile applications. The results reveal that Long Method and Type Checking code smells are twice as likely to occur in mobile applications, while God class and Feature Envy mobile software smells yield equal probability to those of non-mobile systems.

The impact of refactorings on energy consumption has also been studied with the help of JDeodorant in the context of standalone applications by Dhaka and Singh [53], where the results once again verify the fact that improvements of software architecture tend to increase energy consumption.

It is widely acknowledged that software smells are ubiquitous, but an interesting hypothesis is whether smell characteristics vary across different application domains. dos Reis et al. [54] investigated the hypothesis that application domain has a statistically significant impact on the presence of code smells. The conducted a quasi-experiment using 118 Java software systems, classified into 6 application domains. JDeodorant was used to identify Long Method, Large Class and Feature Envy smells, while four more smells were identified by another tool. The results show that the incidence of most code smells does not depend on the application domain, except for the Duplicated Code smell.

JDeodorant has also facilitated research in the field of Requirements Traceability, i.e. the process of tracking source code artifacts that implement specific functional requirements. Specialized software that usually employs Information Retrieval methods identifies the traces of requirements in source code for a given snapshot of a software system. However, software evolution acts detrimentally on the identified traces and causes distorted traceability tracks. According to Mahmoud and Niu [55], these "broken" traces could be restored by refactoring maintenance. To this end, JDeodorant has been employed for the identification and elimination of Feature Envy smells in 3 software systems. Other tools have also been used for the detection and elimination of code clones as well as the restoration of previous identifier names. Results indicate that restoring previous identifier names has positive impact in feature traceability effectiveness. On the contrary, Extract Method refactoring appeared to negatively affect traceability whereas Move Method yielded no significant impact.

Code smells are directly related to the Technical Debt metaphor which assumes that software liabilities set up a context that can make a future change more costly or impossible. A holy grail in the Technical Debt community is the quantification of interest, that is, the increased work effort incurred by the presence of Technical Debt. A case study on the Apache JMeter project [56] found that approximately 42 - 44% of the technical debt incurs positive interest, which indeed costs more to pay off in the future. In particular, the paper focuses on self-admitted technical debt (SATD) which refers to the situation where developers admit that the current implementation is not optimal and write comments alerting the inadequacy of the solution. For the identification of SATD JDeodorant was used to perform the parsing of source code in order to extract comments and map them to their corresponding method.

Self-admitted Technical Debt has also been the focus of a study examining 33K code comments [57]. It was found that self-admitted technical debt can be classified into five main types, namely design, defect, documentation, requirement and test debt, with design debt being the most common type. JDeodorant has been used to parse the source code of five OSS systems and extract comments through the extracted Abstract Syntax Tree of the target code.

Code examples are source code fragments whose purpose is to illustrate how a programming language element, a library, or a specific function should be used. Moreno et al. [23] proposed MUSE (Method USage Examples) as an approach for mining and ranking code examples that show how to use a specific method. MUSE combines static slicing (to simplify examples) with clone detection (to group similar examples), and eventually selects and ranks the best examples in terms of reusability, understandability, and popularity. JDeodorant's static slicer has been used to develop an Example Extractor that parses the source code of the client projects in order to search for invocations (and the associated backward slice) of a target method. The approach has been empirically evaluated using examples from six libraries. The results indicate that it is possible to automatically select and rank examples close to how humans do while most of the code examples (82%) are perceived as useful.

V. Lessons Learned

Through our experience with the development and maintenance of JDeodorant and the interaction with several research groups in the field of smell detection as well as with users of JDeodorant, we have reached a number of conclusions regarding the merits of a successful tool that aids software engineering practice and research. From the list of desirable properties listed below, not all were present in JDeodorant from the beginning; however, we acknowledged their importance and strove to address them:

1) *The design of a tool should be guided by empirical evidence about developers' practices, so that the tool is tailored to the actual needs of the developers*. For example, a recent study [39], investigated the motivations behind refactoring by asking developers the reasons they applied specific refactoring operations, right after pushing their commits in public repositories. The study revealed that refactorings are mainly applied to make easier the completion of a maintenance task (i.e., bug fix, or feature request), rather than eliminating code smells in the software. If we had this knowledge before designing JDeodorant, we would adopt a maintenance-task-oriented design and user interface, instead of a code-smell-oriented one.
2) *All source code analysis techniques implemented by a tool should depend on a higher-level abstraction of the source code, rather than the Abstract Syntax Tree (AST) representation typically created by programming language parsers*. The reason is that as the programming language evolves the AST representation has to be changed or updated to support new language features. If the implemented source code analysis techniques depend on the AST representation, these changes might have to be propagated to the tool's codebase. On the other hand, depending on a higher-level abstraction of the source code, provides shielding against the propagation of such changes. For example, JDeodorant went through 3 major updates in the Java Language Specification (JLS7, JLS8, JLS9) which, however, required minimal changes in its codebase to support the new language features.

3) *Tool usage statistics should be recorded as early as possible, to allow focusing on improvements targeting the most popular tool features*. For example, if we knew from early on that the most popular refactoring applied by JDeodorant users is Extract Method, we would focus our research and development on improving the performance of the algorithms related to the recommendation of this type of refactoring, so that the user experience and appreciation of the tool is better. Focused improvements based on user needs help in building a good reputation for the tool.
4) *A tool should be offered as an IDE plug-in instead of a standalone application*. Building JDeodorant as an Eclipse plug-in was perhaps the wisest decision we could have made. The Eclipse Update Site infrastructure not only makes the installation of plug-ins very easy through the Eclipse Marketplace Client, but it also makes the update of existing clients to the latest release of the plug-in very easy by receiving notifications for updates directly in the IDE. In addition, having a tool running in the same environment where developers work and perform their daily tasks, makes the use of the tool much more convenient.
5) *A tool should require minimum installation and configuration effort*. Installation and configuration are part of the first experience of a new user with a tool. If they are problematic, it is very likely that the user will abandon any further attempt to make the tool work. JDeodorant offers a smooth installation process through Eclipse Marketplace and does not require any configuration. In addition, JDeodorant has a very simple user interface. Essentially, the user has just to select the code element to be analyzed and click on a button to perform the detection of refactoring opportunities. If some refactoring opportunities are found, the user can apply the corresponding refactorings and preview the refactoring changes before they end up in the source code. These features helped a lot in the adoption of the tool by researchers for conducting experiments and empirical studies, by practitioners for finding and applying refactoring opportunities in their daily development and maintenance tasks, by educators for teaching refactoring in a practical way, and by students for using the tool in their projects and assignments.
6) *A tool should be tested in an industrial setting*. Performance and reliability are key characteristics for the adoption of a tool in the industry. Testing a tool only on open-source projects does not guarantee that the tool will be able to scale to the size and characteristics of industrial projects. In addition, developers working on industrial projects may provide very useful feedback about possible tool improvements, especially regarding the integration of the tool in their infrastructure automation process.
7) *A tool should be open-sourced as early as possible, even if it is not as mature and stable*. Unfortunately, JDeodorant was open-sourced in 2014, 7 years after its initial release in 2007. Between 2010 and 2014, JDeodorant source code was available only for research groups and educators through an academic license agreement. The reason for this late decision to open-source it, was the lack of consensus among its contributors about turning JDeodorant into a commercial product, or keeping it as an academic tool. The benefits we experienced after open-sourcing the tool are numerous, including an increased number of bug reports, an increased number of external contributions through pull requests, and over 30 forks of the project on GitHub.
8) *A tool should be accompanied with documentation, tutorials, and code snippets demonstrating the use of its API*. This is essential for the adoption of the tool, because the information provided in relevant publications might not be sufficient for particular needs. One of our mistakes was that we neglected the documentation of the project. The first public document describing the architecture of JDeodorant was released in 2010, three years after the first release of the tool. Moreover, it would have been much more convenient if we incorporated the tool's manual into the JDeodorant Eclipse plug-in, instead of having it available on the tool's website, because the information needed by the user to work with the tool would be less scattered.

VI. Conclusion

Software development practice has acknowledged the importance of code and design quality as a prerequisite for building maintainable and sustainable software projects. Among all methodologies for ensuring quality in code, the notion of smells and refactorings have been widely embraced by practitioners and researchers, possibly due to the fact that they refer to individual, identifiable and actionable issues. JDeodorant has contributed to the developers' arsenal through a set of techniques for the identification of code smells in Java code and the suggestion of refactoring opportunities that can be automatically applied. Our main philosophy was to create a tool that supports all refactoring-related activities, is easy-to-use and configure and provides sufficient accuracy by exploiting the particular characteristics of the target code smell/refactoring.

JDeodorant publications have received an increasing number of citations over the years but more importantly, the tool is being used by an increasing number of developers to refactor their code. A total of 35,000 refactorings has been recorded for the period 2011-2017, with the majority of them targeting long, complex and non-cohesive methods through the Extract Method refactoring. Along the way, several sophisticated smell detection and refactoring recommendation techniques have been proposed, outperforming the accuracy of JDeodorant in some of the cases. However, the reliability, straightforward installation and use, and integration within an IDE rendered JDeodorant a popular baseline for accuracy evaluation, as well as a convenient smell detector for performing empirical studies. Through our experience with the development and maintenance of JDeodorant, we have identified a number of desirable properties (several of which were missing in the early stages of JDeodorant) for tools aiming at supporting developers and researchers.

ACKNOWLEDGMENT

We would like to thank the research community for appreciating JDeodorant and the Most Influential Paper Award committee at SANER 2018 for the honour of selecting our paper. Work reported in this paper has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 780572 (project SDK4ED), as well as from the Natural Sciences and Engineering Research Council of Canada (NSERC grant 435480-2013).

REFERENCES

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[2] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, 2007, pp. 519–520.

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

[4] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, ser. CSMR '08, 2008, pp. 329–331.

[5] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," *Journal of Systems and Software*, vol. 83, no. 3, pp. 391–404, Mar. 2010.

[6] ——, "Identification of extract method refactoring opportunities," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 119–128.

[7] ——, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011.

[8] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique," in *Proceedings of the 25th IEEE International Conference on Software Maintenance*, 2009, pp. 93–101.

[9] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: Identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 1037–1039.

[10] ——, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, Oct. 2012.

[11] G. P. Krishnan and N. Tsantalis, "Unification and refactoring of clones," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, 2014, pp. 104–113.

[12] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, Nov. 2015.

[13] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 60–70.

[14] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM '13, 2013, pp. 392–395.

[15] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[16] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, *Recommending Refactoring Operations in Large Software Systems*. Springer Berlin Heidelberg, 2014, pp. 387–419.

[17] J. A. Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, vol. 58, pp. 231 – 249, 2015.

[18] J. A. Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, Jan 2018.

[19] A. Christopoulou, E. Giakoumakis, V. E. Zafeiris, and V. Soukara, "Automated refactoring to the strategy design pattern," *Information and Software Technology*, vol. 54, no. 11, pp. 1202 – 1214, 2012.

[20] M. A. G. Gaitani, V. E. Zafeiris, N. Diamantidis, and E. Giakoumakis, "Automated refactoring to the null object design pattern," *Information and Software Technology*, vol. 59, pp. 33 – 52, 2015.

[21] V. E. Zafeiris, S. H. Poulias, N. Diamantidis, and E. Giakoumakis, "Automated refactoring of super-class method invocations to the template method design pattern," *Information and Software Technology*, vol. 82, pp. 19 – 35, 2017.

[22] 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.

[23] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, pp. 880–890.

[24] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 25–34.

[25] 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, Jul. 2014.

[26] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13, 2013, pp. 268–278.

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

[28] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending move method refactorings using dependency sets," in *Proceedings of the 2013 20th Working Conference on Reverse Engineering (WCRE)*, Oct 2013, pp. 232–241.

[29] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "JMove: A novel heuristic and tool to detect move method refactoring opportunities," *Journal of Systems and Software*, vol. 138, pp. 19 – 36, 2018.

[30] F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *Proceedings of the 2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[31] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. D. Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[32] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 535–546.

[33] H. Liu, Y. Wu, W. Liu, Q. Liu, and C. Li, "Domino effect: Move more methods once a method is moved," in *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 1–12.

[34] H. Liu, Q. Liu, Z. Niu, and Y. Liu, "Dynamic and automatic feedback-based threshold adaptation for code smell detection," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 544–558, June 2016.

[35] N. Ujihara, A. Ouni, T. Ishio, and K. Inoue, "c-JRefRec: Change-based identification of move method refactoring opportunities," in *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 482–486.

[36] D. Silva, R. Terra, and M. T. Valente, "Recommending automated extract method refactorings," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014, 2014, pp. 146–156.

[37] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou, "Identifying extract method refactoring opportunities based on functional relevance," *IEEE Transactions on Software Engineering*, vol. 43, no. 10, pp. 954–974, Oct 2017.

[38] S. Xu, A. Sivaraman, S. C. Khoo, and J. Xu, "Gems: An extract method refactoring recommender," in *Proceedings of the 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, Oct 2017, pp. 24–34.

[39] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 858–870.

[40] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "A two-step technique for extract class refactoring," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10, 2010, pp. 151–154.

[41] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, no. 3, pp. 397 – 414, 2011.

[42] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in eclipse: The ARIES project," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 1419–1422.

[43] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, Dec 2014.

[44] G. Bavota, R. Oliveto, A. D. Lucia, A. Marcus, Y. G. Guehnuc, and G. Antoniol, "In medio stat virtus: Extract class refactoring through nash equilibria," in *Proceedings of the 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, Feb 2014, pp. 214–223.

[45] J. Lee, D.-K. Kim, S. Kim, and S. Park, "Decomposing class responsibilities using distance-based method similarity," *Frontiers of Computer Science*, vol. 10, no. 4, pp. 612–630, Aug 2016.

[46] K. Cassell, "Using clustering techniques to guide refactoring of object-oriented classes," Ph.D. dissertation, Victoria University of Wellington, 2012.

[47] A. Lozano, K. Mens, and J. Portugal, "Analyzing code evolution to uncover relations between bad smells," in *Proceedings of the IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention*, ser. PPAP 2015, March 2015, pp. 1–4.

[48] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell: An empirical study," in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE '15, 2015, pp. 8:1–8:10.

[49] F. A. Fontana and S. Spinelli, "Impact of refactoring on quality code evaluation," in *Proceedings of the 4th Workshop on Refactoring Tools*, ser. WRT '11, 2011, pp. 37–40.

[50] A. Kaur, K. Kaur, and S. Jain, "Predicting software change-proneness with code smells and class imbalance learning," in *Proceedings of the 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Sept 2016, pp. 746–754.

[51] A. Rodriguez, M. Longo, and A. Zunino, "Using bad smell-driven code refactorings in mobile applications to reduce battery usage," in *Proceedings of the 16 Simposio Argentino de Ingeniera de Software (ASSE)*, Sep 2015, pp. 56–68–486.

[52] D. Verloop, "Code Smells in the Mobile Applications Domain," Master's thesis, Delft University of Technology, 2013.

[53] G. Dhaka and P. Singh, "An empirical investigation into code smell elimination sequences for energy efficient software," in *Proceedings of the 2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, Dec 2016, pp. 349–352.

[54] J. P. dos Reis, F. B. e Abreu, and G. de F. Carneiro, "Code smells incidence: Does it depend on the application domain?" in *Proceedings of the 2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, Sept 2016, pp. 172–177.

[55] A. Mahmoud and N. Niu, "Supporting requirements to code traceability through refactoring," *Requirements Engineering*, vol. 19, no. 3, pp. 309–329, Sep 2014.

[56] Y. Kamei, E. da S. Maldonado, E. Shihab, and N. Ubayashi, "Using analytics to quantify interest of self-admitted technical debt," in *Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2016) and 1st International Workshop on Technical Debt Analytics (TDA 2016)*, 2016, pp. 68–71.

[57] E. d. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *Proceedings of the 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, Oct 2015, pp. 9–15.