A Textual-based Technique for Smell Detection

Fabio Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]
[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University of Molise, Italy

***Abstract*—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 10 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.**

I. Introduction

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working erodes the original design of the systems and introduces *technical debt* [1]. The erosion of the original design is generally represented by "*poor design or implementation choices*" [2], usually referred to as bad code smells (also named "code smells" or simply "smells"). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developers' perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change- and fault-proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [17], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, *i.e.*, the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For instance, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as Blob, are generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. For instance, Palomba *et al.* [24] recently proposed the use of historical information for detecting several bad smells, including Blob. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (i.e., terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we aim at investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of smells rather than recommending refactoring solutions for a specific smell. To this aim, we define TACO (**T**extual **A**nalysis for **C**ode smell detecti**O**n), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, *i.e.*, *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Misplaced Class*. We conducted an empirical study involving 10 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision 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 that better performance can be achieved by combining the two sources of information.

II. Background and Related Work

Starting from the definition of design defects proposed in [2], [35], [36], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to visualize code smells or refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key *symptoms* that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., if Lines Of Code $\geqslant k$); (ii) conflating the identified symptoms, leading to the final rule for detecting the smells [17], [18], [19], [20], [21], [22], [23]. These detection techniques mainly differ in the set of used structural metrics, which depends on the type of code smells to detect, and how the identified key *symptoms* are combined. For example, such a combination can be performed using AND/OR operators [17], [18], [19], Bayesian belief networks [20], and B-Splines [21]. In this context, Moha *et al.* [22] introduced DECOR, a method for specifying and detecting code and design smells using a Domain-Specific Language (DSL). Four design smells are identified by DECOR, namely *Blob*, *Swiss Army Knife*, *Functional Decomposition*, and *Spaghetti Code*. Tsantalis *et al.* [23] present JDeodorant, a tool for detecting *Feature Envy* bad smells and suggesting move method refactoring opportunities. In its current version JDeodorant also implements other detection strategies for detecting three other code smells (i.e., *State Checking*, *Long Method*, and *Blob*) [41], [42], [43]. In the field of refactoring, Bavota *et al.* proposed the use of *Relational Topic Modeling* to suggest *Move Method* [32] and *Move Class* [31] refactoring opportunities. Furthermore, they also proposed a combination of both structural and conceptual analysis for *Extract Class* [30] and *Extract Package* refactoring [44]. Unlike these works, our technique is able to (i) identify actual problems occurring to code components, i.e. code smells, rather than recommending refactoring solutions, and (ii) detect smells at different levels of granularity (i.e., method, class, package) by only exploiting textual information.

Code smell detection can also be formulated as an optimization problem, as pointed out by Kessentini *et al.* [45], leading to the usage of search algorithms to solve it [45], [46], [47], [48]. Kessentini *et al.* [45] used genetic algorithms to detect code smells following the assumption that what significantly diverges from good design practices is likely to represent a design problem. Later works further investigated this idea using different search algorithms, such as parallel collaborative search algorithms [46], competitive co-evolutionary search [47], and bi-level search algorithms based on genetic programming [48]. Besides structural information, historical data can be exploited for the detection of code smells. Ratiu *et al.* [9] propose to use the historical information of the suspected flawed structure to increase the accuracy of the automatic problem detection. Palomba *et al.* [24] provide evidence that historical data can be successfully exploited to identify not only smells that are intrinsically characterized by their evolution across the program history – such as *Divergent Change*, *Parallel Inheritance*, and *Shotgun Surgery* – but also smells such as *Blob* and *Feature Envy* [24].

Our paper. Most previous work on code smell detection relies on structural metrics, such as size and complexity metrics. Different smells require different sets of structural metrics as well as different rules for detection purposes. In this paper, we introduce a textual-based technique that (i) exploits the textual information contained in source code elements and (ii) relies on textual similarity between code elements characterizing a code component. The preliminary idea has been introduced in [49] and applied to the identification of *Long Method*. Here, we extend the approach to the identification of four more code smells (i.e. *Feature Envy*, *Blob*, *Promiscous Package*, and *Misplaced Class*). We also evaluate the approach on a much larger case study.

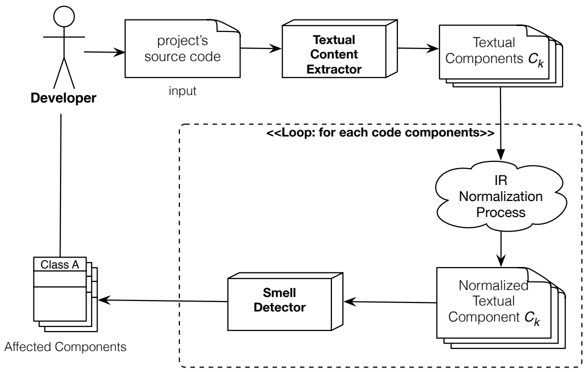

Fig. 1. TACO: The proposed code smell detection process.

III. The Proposed Code Smell Detection Process

Figure 1 depicts the main steps used by TACO in order to compute the probability of a code component being affected by a smell, which are (i) *Textual Content Extractor*, (ii) *IR Normalization Process*, and (iii) *Smell Detector*.

Textual Content Extractor. Starting from the set of code artifacts composing the software project under analysis, the first step is responsible for the extraction of the textual content characterizing each code component by selecting only the textual elements actually needed for the textual analysis process, i.e., source code identifiers and comments.

IR Normalization Process. Identifiers and comments of each component are firstly normalized by using a typical Information Retrieval (IR) normalization process. Thus, the terms contained in the source code are transformed by applying the following steps [50]: (i) separating composed identifiers using the camel case splitting which splits words based on underscores, capital letters and numerical digits; (ii) reducing to lower case letters of extracted words; (iii) removing special characters, programming keywords and common English stop words; (iv) stemming words to their original roots via Porter's stemmer [51]. Finally, the normalized words are weighted using the *term frequency - inverse document frequency* (*tf-idf*) schema [50], which reduces the relevance of too generic words that are contained in most source components.

Smell Detector. The normalized textual content of each code component is then individually analyzed by the *Smell Detector*, which applies different heuristics to identify target smells. The detector relies on Latent Semantic Indexing (LSI) [52], an extension of the Vector Space Model (VSM) [50], which models code components as vectors of terms occurring in a given software system. LSI uses Singular Value Decomposition (SVD) [53] to cluster code components according to the relationships among words and among code components (co-occurrences). Then, the original vectors (code components) are projected into a reduced k space of concepts to limit the effect of textual noise. For the choice of size of the reduced space (k) we used the heuristic proposed by Kuhn *et al.* [54] that provided good results in many software engineering applications, i.e., $k = (m \times n)^{0.2}$ where m denotes the vocabulary size and n denotes the number of documents (code components in our case). Finally, the textual similarity among software components is measured as the cosine of the angle between the corresponding vectors. The similarity values are then combined in different ways, according to the type of smell we are interested in, to obtain a probability that a code component is actually smelly. For detection purpose, we convert such a probability in a truth value in the set {`true, false`} to denote whether a given code component is affect or not by a specific smell. In the context of this paper, we instantiated TACO for detecting five code smells, namely (i) *Long Method*, (ii) *Feature Envy*, (iii) *Blob*, (iv) *Promiscuous Package*, and (v) *Misplaced Class*. We have instantiated TACO on these code smells in order to demonstrate how textual analysis can be useful for detecting smells at different levels of granularity (i.e., method-level, class-level, and package-level). Moreover, all the selected smells violate, in different ways, the OOP single responsibility principle [55], [56]. For instance, a Blob class implements more than one responsibility, while a Feature Envy is a method which has a different responsibility with respect to the one implemented in the class it is actually in. This peculiar characteristic makes them particularly suitable for a textual-based technique, since *the higher the number of the responsibilities implemented in a code component, the higher the probability that such a component contains heterogeneous identifiers and/or comments*. In the following subsections, we detail how the general process depicted in Figure 1 has been applied for detecting the smells described above.

A. Computing Code Smell Probability

Long Method. This smell arises when a method implements a main functionality, together with auxiliary functions that should be managed in different methods. The refactoring associated with such a smell is clearly *Extract Method*[1], which allows the identification of portions of the method that should be treated separately, with the aim to create new methods for managing them [2]. It is worth noting that the definition of the smell strongly differs from its name, since this smell is only partially related to the size of a method. Rather, it is related to how much responsibilities a method manages, i.e. whether a method violates the single responsibility principle.

Textual Diagnosis. Given the definition above, our conjecture is that a method is affected by this smell *when it is composed of sets of statements semantically distant to each other*. In order to detect the different sets of statements composing the method, we re-implemented *SEGMENT*, the approach proposed by Wang *et al.* [57], which uses both structural analysis and naming information to automatically segment a method into a set of "*consecutive statements that logically implement a high level action*" [57]. Once we identified the sets of statements (i.e., segments) composing the method, we considered each of them as a single document. Then, for each pair of documents, we apply LSI [58] and the cosine similarity to have a similarity value. More formally, let M be the method under analysis, let $S = \{s_1, \ldots, s_n\}$ be the set of segments in M, we compute the textual cohesion of the method M as the average similarity between all pairs of its segments:

$$MethodCohesion(M) = \operatorname*{mean}_{i \neq j} \; sim(s_i, s_j) \qquad (1)$$

where n is the number of code segments in M, and $sim(s_i, s_j)$ denotes the cosine similarity between two segments s_i and s_j in M. Starting from our definition of textual cohesion of M, we compute the probability that M is affected by *Long Method* using the following formula:

$$P_{LM}(M) = 1 - MethodCohesion(M) \qquad (2)$$

It is worth noting that $P_{LM}(M)$ ranges in $[0; 1]$. The higher its value, the higher the probability that method M represents a Long Method instance.

Feature Envy. According to Fowler's definition [2], this smell occurs when "*a method is more interested in another class than the one it is actually in*". Thus, a method affected by Feature Envy is not correctly placed, since it exhibits high coupling with a class different than the one where it is located in. To remove this smell, a *Move Method* refactoring aimed at moving it to the more suitable class is needed.

Textual Diagnosis. When computing the probability that a method is affected by such a smell, we conjecture that a *method more interested in another class is characterized by a higher similarity with the concepts implemented in the envied class, with respect to the concepts implemented in the class it is actually in*. Let M be the method under analysis belonging to the class C_O, and let $C_{related} = \{C_1, \ldots, C_n\}$ be the set of classes in the system sharing at least one term with M. First, we derive the class ($C_{closest}$) having the highest textual similarity with M as follows:

$$C_{closest} = \operatorname*{arg\,max}_{C_i \in C_{related}} \; sim(M, C_i) \qquad (3)$$

Then, if $C_{closest}$ is not the class where M is actually placed in (i.e., $C_{closest} \neq C_O$), then, M should be moved to the class $C_{closest}$. Therefore, we compute the probability for M to be a *Feature Envy* instance as:

$$P_{FE}(M) = sim(M, C_{closest}) - sim(M, C_O) \qquad (4)$$

The formula above is equal to zero when $C_{closest} = C_O$, i.e., the method M is correctly placed. Otherwise, if $C_{closest} \neq C_O$, the probability is equal to the difference between the textual similarities of M and the two classes

[1] More details about refactoring operations defined in literature can be found in the refactoring catalog available at http://refactoring.com/catalog/

$C_{closest}$ and C_O.

Blob. These classes are usually characterized by a huge size, a large number of attributes and methods and a high number of dependencies with data classes [2]. This smell involves low cohesive classes that are responsible for the management of different functionalities. The *Extract Class* refactoring is the more suitable operation that can be applied for removing this smell type, since it allows to split the original class by creating new classes, re-distributing its responsibilities.

Textual Diagnosis. Our conjecture is that Blob classes *are characterized by a semantic scattering of contents*. More formally, let C be the class under analysis, let $M = \{M_1, \ldots, M_n\}$ be the set of methods in C, we compute the textual cohesion of the class C as defined by Marcus and Poshyvanyk [25]:

$$ClassCohesion(C) = \underset{i \neq j}{\text{mean}} \ sim(M_i, M_j) \tag{5}$$

where n is the number of methods in C, and $sim(M_i, M_j)$ denotes the cosine similarity between two methods M_i and M_j in C. Therefore, we compute the probability that C is affected by *Blob* using the following formula:

$$P_B(C) = 1 - ClassCohesion(C) \tag{6}$$

Also in this case, $P_B(C)$ ranges in $[0; 1]$.

Promiscuous Package. A package can be considered as promiscuous if it contains classes implementing too many features, making it too hard to understand and maintain [2]. As for Blob, this smell arises when the package has low cohesion, since it manages different responsibilities. In this case, to refactor the smell an *Extract Package* operation is needed for split the package in more sub-packages, re-organizing the responsibilities of the original promiscuous package.

Textual Diagnosis. We conjecture that packages affected by this smell *are characterized by subset of classes semantically distant from the other classes of the package*. Formally, let P be the package under analysis, and let $C = \{C_1, \ldots, C_n\}$ be the set of classes in P, the textual cohesion of the package P is defined as done by Poshyvanyk *et al.* [59]:

$$PackageCohesion(P) = \underset{i \neq j}{\text{mean}} \ sim(C_i, C_j) \tag{7}$$

where n is the number of classes in P, and $sim(C_i, C_j)$ is the cosine similarity between two classes C_i and C_j in P. Given such definition, we define the probability that P is a *Promiscuous Package* using the formula below:

$$P_{PP}(P) = 1 - PackageCohesion(P) \tag{8}$$

$P_{PP}(P)$ assumes values in the range $[0; 1]$.

Misplaced Class. A *Misplaced Class* smell suggests a class that is in a package that contains other classes not related to it [2]. The obvious way to remove such a smell is to apply a *Move Class* refactoring able to place the class in a more related package.

Textual Diagnosis. Our conjecture is that a class affected by this smell is *semantically more related to a different package with respect to the package it is actually in*. Let C be the class under analysis, contained in the package P_O, and let $P_{related} = \{P_1, \ldots, P_n\}$ be the set of packages that share at least one term with C. We firstly retrieve the package with the highest textual similarity with C, using the following formula:

$$P_{closest} = \underset{P_i \in P_{related}}{\arg\max} \ sim(C, P_i) \tag{9}$$

Then, if $P_{closest}$ is different from the package where the class C is actually placed in (i.e., $P_{closest} \neq P_O$), then C should be moved to the package $P_{closest}$. More formally, we compute the probability C is affected by a *Misplaced Class* as:

$$P_{MC}(C) = sim(C, P_{closest}) - sim(C, P_O) \tag{10}$$

As in the case of *Feature Envy*, the value is equal to zero if $P_{closest} = P_O$. Otherwise, if $P_{closest} \neq P_O$, the probability is equal to the difference between the textual similarities of C and the two packages $P_{closest}$ and P_O.

B. Applying TACO to Code Smell Detection

TACO assigns a smelliness probability to each code component according to the textual diagnosis metrics reported above. In the context of smell detection, we need to convert such probabilities in a truth value in the set {`true, false`}. Thus, we need to discriminate when a probability indicates the presence of a given smell with respect to the cases where such probability is not enough for considering a code component affected by a smell. After different experiments aimed at identifying the optimal cut-off point, we found that the best results are obtained when using as threshold the median of the non-null values of the probability distribution of the system under analysis. Interested readers can find the results of such a calibration analysis in our online appendix [60].

IV. Empirical Study Definition and Design

The *goal* of the study is to evaluate TACO, with the *purpose* of investigating its effectiveness during the detection of code smells in software systems. The *quality focus* is on the detection accuracy and completeness when compared to the approaches based purely on structural analysis. The *perspective* is of researchers, who want to evaluate the effectiveness of textual analysis for detecting code smells for building better recommenders for developers.

The *context* of the study consists of ten open source software projects. Table I reports the characteristics of the analyzed systems[2], namely their size in terms of number of classes, number of methods, and KLOC. Among the analyzed projects, we have seven projects belonging to the Apache ecosystem[3], and three open source projects from elsewhere. Note that our choice of the subject systems is not random, but instigated by

[2] The list of repositories is available in our online appendix [60]

[3] http://www.apache.org/ verified on January 2016

TABLE I
CHARACTERISTICS OF THE SOFTWARE SYSTEMS IN OUR DATASET

System	Classes	Methods	KLOCs
Apache Ant 1.8.0	813	8,540	204
aTunes 2.0.0	655	5,109	106
Eclipse Core 3.6.1	1,181	18,234	441
Apache Hive 0.9	1,115	9,572	204
Apache Ivy 2.1.0	349	3,775	58
Apache Lucene 3.6	2,246	17,021	466
JVLT 1.3.2	221	1,714	29
Apache Pig 0.8	922	7,619	184
Apache Qpid 0.18	922	9,777	193
Apache Xerces 2.3.0	736	7,342	201

our aim to analyze projects belonging to different ecosystems, having different size and scope.

In this study, we investigate the following research questions:

- **RQ$_1$:** *What is the accuracy of TACO in detecting code smells?*
- **RQ$_2$:** *How does TACO perform when compared with state-of-the-art techniques purely based on structural analysis?*
- **RQ$_3$:** *To what extent is TACO complementary with respect to the structural-based code smell detectors?*

To answer **RQ$_1$** we run TACO on the selected software projects. To evaluate its accuracy, we needed an oracle reporting the actual code smells contained in the considered systems. For all of the code smells considered in this paper, an annotated set of such smells is publicly available in literature [61].

Once obtained the set of smells detected by TACO on each software project, we evaluated its performances by using two widely adopted Information Retrieval (IR) metrics, i.e. precision and recall [50]:

$$precision = \frac{|TP|}{|TP \cup FP|}\% \qquad recall = \frac{|TP|}{|TP \cup FN|}\%$$

where TP and FP represent the set of true and false positive smells detected by TACO, respectively, while FN (false negatives) represents the set of smell instances in the oracle missed by TACO. To have an aggregate indicator of precision and recall, we also report the F-measure, defined as the harmonic mean of precision and recall:

$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall}\%$$

To answer **RQ$_2$**, we run code smell detection techniques purely based on structural analysis on the same software projects on which we run TACO. For *Long Method* and *Blob* smells we compared TACO with DECOR, the structural detection approach proposed by Moha *et al.* [22]. Since a tool implementing the approach is not publicly available, we re-implemented the detection rules defined by DECOR, which are available online[4]. For the *Feature Envy* smell, we considered JDeodorant as the alternative approach [23]. JDeodorant is available as open source Eclipse plug-in[5]. The technique behind JDeodorant analyzes attributes and method calls of each method in the system under analysis with the aim to form a set of candidate target classes where the method should be moved. As for *Promiscuous Package*, we compared TACO with the clustering-based algorithm proposed in [33], where classes are grouped using the dependencies among them. Finally, for *Misplaced Class*, we used the approach proposed by Atkinson and King [34], which traverse the abstract syntax tree of a class in order to determine, for each feature, the set of classes referencing them. In this case, a class is affected by *Misplaced Class* if it has more dependencies with a different class with respect to the one it is actually in.

Even if in literature several other approaches have been defined for smell detection, our choice of the alternative techniques has been guided by (i) the availability of a tool (e.g., JDeodorant), or (ii) the simplicity of a re-implementation, in order to avoid possible errors due to a wrong implementation. To compare the performance achieved by TACO with those of the alternative structural detection techniques, we used the same set of accuracy metrics used for measuring TACO's results, i.e., recall, precision, and F-measures.

Finally, to answer **RQ$_3$**, we compared the sets of smell instances correctly detected by TACO and by the alternative approaches by computing the following overlap metrics:

$$correct_{m_i \cap m_j} = \frac{|correct_{m_i} \cap correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\%$$

$$correct_{m_i \setminus m_j} = \frac{|correct_{m_i} \setminus correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\%$$

where $correct_{m_i}$ represents the set of correct code smells detected by the approach m_i, $correct_{m_i \cap m_j}$ measures the overlap between the set of true code smells detected by both approaches m_i and m_j, and $correct_{m_i \setminus m_j}$ appraises the true smells detected by m_i only and missed by m_j. The latter metric provides an indication of how a code smell detection technique contributes to enriching the set of correct code smells identified by another approach. This information can be used to analyze the complementarity of structural and textual information when performing code smell detection.

V. RESULTS

This section reports the results of our study, with the aim of addressing the research questions formulated in the previous section. To avoid redundancies, we report the results for all the three research questions together, discussing each smell separately. Tables II, III, IV, V, and VI show the results achieved by TACO and by the structural approaches on the ten subject systems for *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Misplaced Class*, respectively. Specifically, we report (i) the number of actual components

[4] http://www.ptidej.net/research/designsmells/grammar/

[5] http://www.jdeodorant.com/

TABLE II
LONG METHOD - TACO ACCURACY COMPARED TO DECOR.

	#Actual	TACO			DECOR		
Project	Smells	Precision	Recall	F-measure	Precision	Recall	F-measure
Apache Ant	52	68%	96%	80%	63%	56%	59%
aTunes	11	75%	81%	78%	13%	18%	15%
Eclipse Core	89	74%	78%	76%	24%	82%	37%
Apache Hive	53	70%	85%	77%	47%	66%	55%
Apache Ivy	18	76%	89%	82%	75%	50%	60%
Apache Lucene	82	65%	80%	72%	34%	71%	46%
JVLT	7	63%	71%	67%	60%	43%	50%
Apache Pig	33	74%	85%	79%	19%	64%	29%
Apache Qpid	39	70%	82%	75%	42%	56%	48%
Apache Xerces	27	77%	85%	81%	31%	85%	46%
Overall	**411**	**71%**	**83%**	**76%**	**32%**	**67%**	**43%**

TABLE III
FEATURE ENVY - TACO ACCURACY COMPARED TO JDEODORANT.

	#Actual	TACO			JDeodorant		
Project	Smells	Precision	Recall	F-measure	Precision	Recall	F-measure
Apache Ant	8	67%	75%	71%	15%	25%	19%
aTunes	8	60%	75%	67%	34%	75%	50%
Eclipse Core	3	50%	67%	57%	0%	0%	0%
Apache Hive	22	88%	68%	77%	89%	77%	83%
Apache Ivy	17	67%	59%	63%	77%	59%	67%
Apache Lucene	26	61%	73%	67%	51%	88%	65%
JVLT	1	50%	100%	67%	50%	100%	67%
Apache Pig	7	56%	71%	63%	57%	57%	57%
Apache Qpid	15	76%	87%	81%	73%	73%	73%
Apache Xerces	8	67%	75%	71%	63%	63%	63%
Overall	**115**	**67%**	**72%**	**70%**	**57%**	**69%**	**62%**

affected by a smell (column *#Actual Smells*), (ii) accuracy metrics for each approach involved in a comparison, in terms of precision, recall, and f-measure. In our online appendix [60] we provide a technical report in which we also report the number of true and false positive instances found by both TACO and the alternative techniques. In addition, Table VII reports values concerning overlap and differences between TACO and the structural techniques: column "TACO $\cap$ ST" reports the percentage of correct smell instances detected by both TACO and the alternative structural approach; column "TACO $\setminus$ ST" reports the percentage of correct code smells correctly identified by TACO but missed by the structural technique; finally, column "ST $\setminus$ TACO" reports the percentage of correct code smells identified by the structural technique but not by TACO. In the following, we discuss the results for each smell type.

A. Long Method Discussion

In the set of subject systems, we found 411 instances of this smell. The analysis of the results indicates that a textual approach more accurately detects instances of the Long Method smell. Specifically, the F-measure on the overall dataset of TACO is 76% (83% of recall and 71% of precision) against 43% (67% of recall and 32% of precision) achieved by the alternative approach. The low accuracy achieved by DECOR is due to the fact that the number of lines of code (LOC) of a method only tells a part of the whole story. Indeed, there are several cases in which a method is cohesive even if the large size of the method can indicate the presence of such a smell. The use of textual analysis is able to better discriminate whether a method implements more than one functionality. Of particular interest is the case of the `Eclipse Core` project: Here we found several methods having more than 100 LOC, implementing intrinsically complex operations, but not characterized by the presence of a Long Method smell. For example, the method `createSingleAssistNameReference` of the class `CompletionParser` needs to parse the actual content of the IDE workspace in order to automatically suggest to developers the methods she can use in her context. Although the method has 113 LOC, it can not be considered a *Long Method* smell, since it has a focused responsibility implemented across multiple lines. However, the DECOR rule detects this method as affected by the smell. Conversely, TACO correctly discards it from the candidate smell set. Moreover, our approach is able to identify different types of *Long Method* instances with respect to the ones a structural technique can identify. As an example, the method `findTypesAndPackages` of the class `CompletionEngine`, allows to discover the classes and the packages of a given project. Clearly, this method manages different tasks, even if its size is not large (i.e., 58 lines of code). This means that the use of textual analysis is actually useful to avoid the identification of many false positive candidates, but also to detect instances of *Long Method* that the structural technique is not able to detect. This claim is supported by the results achieved when analyzing the overlap between TACO and DECOR (see Table VII). Indeed, we observed an overlap of 58%, i.e. 58% of the actual *Long Method* instances are correctly detected by both TACO and DECOR, while it is interesting to note that TACO is able to correctly detect 30% of instances that DECOR is not able to detect. Finally, 12% of instances are correctly identified by DECOR and missed by TACO. An example of a *Long Method* instance correctly identified by DECOR and missed by TACO can be found in the class `RetrieveEngine` of the `Apache Ivy` project, where the method `retrieve` is characterized by 165 LOC. This method implements the basic operation of finding the settings of the machine Ivy is working on, however it also has an auxiliary function checking whether or not the settings are up to date. The textual approach fails in the detection of this smell because of the consistent vocabulary of the method. The achieved results highlight a tangible potential of combining structural and textual information for detecting this type of smell. We are planning to tackle this combination as part of our future work.

B. Feature Envy Discussion

For the *Feature Envy* smell, we found a total of 115 affected methods in our dataset. TACO has been able to identify 83 of them (recall of 72%), against the 79 detected by JDeodorant (recall of 69%). On the other hand, the precision obtained by TACO is higher than the one achieved by JDeodorant (67% against 57%). Overall, TACO's F-Measure is higher than JDeodorant (70% against 62%), and our approach outperforms the alternative one on 7 out of 10 systems (70% of the times). It is important to note that *JDeodorant* is a *refactoring* tool and, as such, it identifies *Feature Envy* smells with the purpose of

TABLE IV
BLOB - TACO ACCURACY COMPARED TO DECOR.

Project	#Actual Smells	TACO Precision	TACO Recall	TACO F-measure	DECOR Precision	DECOR Recall	DECOR F-measure
Apache Ant	31	68%	81%	74%	81%	55%	65%
aTunes	9	64%	78%	70%	75%	34%	46%
Eclipse Core	43	63%	81%	71%	48%	72%	52%
Apache Hive	27	71%	63%	67%	72%	48%	58%
Apache Ivy	10	80%	80%	80%	100%	30%	46%
Apache Lucene	27	74%	81%	77%	67%	67%	67%
JVLT	3	50%	34%	40%	100%	34%	50%
Apache Pig	7	67%	57%	62%	42%	71%	53%
Apache Qpid	29	79%	93%	86%	80%	41%	55%
Apache Xerces	16	70%	88%	78%	59%	81%	68%
Overall	**202**	**70%**	**79%**	**74%**	**62%**	**57%**	**60%**

TABLE V
PROMISCUOUS PACKAGE - TACO ACCURACY COMPARED TO THE APPROACH PROPOSED IN [33].

Project	#Actual Smells	TACO Precision	TACO Recall	TACO F-measure	Approach in [33] Precision	Approach in [33] Recall	Approach in [33] F-measure
Apache Ant	10	89%	80%	84%	67%	40%	50%
aTunes	3	50%	34%	40%	50%	34%	40%
Eclipse Core	9	70%	78%	74%	50%	34%	40%
Apache Hive	7	83%	71%	77%	57%	57%	57%
Apache Ivy	4	100%	50%	67%	50%	25%	34%
Apache Lucene	26	71%	92%	80%	70%	73%	72%
JVLT	3	25%	34%	29%	0%	0%	0%
Apache Pig	10	89%	80%	84%	83%	50%	63%
Apache Qpid	15	65%	73%	69%	58%	47%	52%
Apache Xerces	4	50%	50%	50%	34%	25%	29%
Overall	**91**	**71%**	**76%**	**73%**	**62%**	**49%**	**55%**

suggesting opportunities of *Move Method refactoring*. Thus, the tool detects the smell only if the application of the refactoring is actually possible. To this aim, JDeodorant checks some preconditions to ensure that the program behavior does not change after the application of the refactoring [23]. For example, one of the preconditions considered is that *the envied class does not contain a method having the same signature as the moved method* [23]. In order to set a fair comparison with our approach, we filtered the *Feature Envy* instances found by our approach, using the same set of preconditions defined by JDeodorant [23]. During this step, we removed 1 correct instances and 3 false positives from the initial set. Once the filtering has been applied, TACO's precision increases to 69%, while its recall decreases to 70%. Moreover, it is interesting to note that the two approaches are highly complementary, as reported in Table VII. In fact, 48% of the correct smell instances have been detected by both approaches, while our textual technique identifies 28% of instances missed by JDeodorant. On the other side, the structural tool is able to capture 24% of correct *Feature Envy* instances missed by TACO. An example of an instance correctly captured by TACO is represented by the method `readSchema` of the class `IndexSchema` of `Apache Lucene`. Here the method, implementing the functionality able to read the schema of a database, has a concept more related to the class `ZkIndexSchemaReader` with respect to the class it is actually in. On the other hand, JDeodorant is the only technique able to correctly identify the smell affecting the method `isRebuildRequired` of the class `WebsphereDeploymentTool`, present in `Apache Ant` project. In this case, TACO is not able to identify the smell since it is characterized by a high number of dependencies with the envied class, even if the textual content of the method is more related to the actual class.

C. Blob Discussion

As for the detection of *Blobs*, TACO is able to achieve, overall, a precision of 70% and a recall of 79% (F-measure=74%), while DECOR is able to achieve a precision of 62% and a recall of 57% (F-measure=60%). Specifically, on average, TACO is 15% more accurate in detecting this type of smell. The only exception is represented by the `jVLT` project. In this case, DECOR is able to identify one out of three Blob instances present in the system without any false positive (precision=100%, recall=34%), while TACO outputs two candidates, of which one is a false positive (precision=50%, recall=34%). In particular, TACO fails in the suggestion of the class `Utils` of the package `net.sourceforge.jvlt.utils`. Even if the methods of this class are not cohesive, since the class implements several utility methods used by other classes, it can not be considered a Blob since it does not centralize the behavior of a portion of the system. On the contrary, an example of *Blob* correctly detected by TACO can be found in the class `AudioFile` of the `aTunes` project. This class has the goal to map an entity, but actually it implements several methods for the management of such entities and also for the management of users' playlists. DECOR can not detect this smell since the class does not seem to be a *controller* class[6]. Looking at the complementarity in Table VII, we observed that the textual approach is able to detect a consistent number of correct instances missed by DECOR. Indeed, TACO is able to capture 41% of classes affected by *Blob* missed by DECOR, while DECOR detects 19% of instances missed by TACO. Noticeably, 40% of correct instances are identified by both the approaches. This result highlights how the use of textual analysis can be particularly suitable for detecting the *Blob* code smell.

D. Promiscuous Package Discussion

Over the set composed of 91 *Promiscuous Package* instances, TACO achieves 71% of precision and 76% of recall (F-measure=73%), outperforming on 9 systems out of 10 the alternative structural-based technique. This result clearly indicates how the use of textual information is beneficial in order to identify packages composed of classes implementing different responsibilities. The only exception regards the `aTunes` project, in which the two techniques obtain the same accuracy (F-Measure=40%). Specifically, in this system there are 3 promiscuous packages, and the two approaches are able to correctly detect only one instance each. TACO detects as promiscuous the `net.sourceforge.atunes.kernel.actions` package, that is characterized by 133 classes implementing actions related to the management of (i) the buttons present in the UI, (ii) the options for saving audio files in local, and (iii) the synchronization of the playlists. In this

[6] DECOR identify a controller class if its name contains a suffix in a set {*Process, Control, Command, Manage, Drive, System*}

TABLE VI
MISPLACED CLASS - TACO ACCURACY COMPARED TO THE APPROACH PROPOSED BY ATKINSON AND KING [34].

	#Actual	TACO			Approach in [34]		
Project	Smells	Precision	Recall	F-measure	Precision	Recall	F-measure
Apache Ant	4	75%	75%	75%	50%	50%	50%
aTunes	0	-	-	-	-	-	-
Eclipse Core	11	73%	73%	73%	78%	64%	70%
Apache Hive	2	50%	100%	67%	20%	50%	29%
Apache Ivy	10	50%	100%	67%	16%	50%	25%
Apache Lucene	27	82%	82%	82%	71%	46%	56%
JVLT	0	-	-	-	-	-	-
Apache Pig	7	90%	90%	90%	75%	14%	24%
Apache Qpid	2	67%	100%	80%	0%	0%	0%
Apache Xerces	16	75%	75%	75%	67%	50%	57%
Overall	**57**	**77%**	**84%**	**81%**	**53%**	**37%**	**43%**

TABLE VII
OVERLAP BETWEEN TACO AND THE STRUCTURAL TECHNIQUES (ST). FOR LONG METHOD AND BLOB THE STRUCTURAL TECHNIQUE IS DECOR, FOR FEATURE ENVY IT IS JDEODORANT, FOR PROMISCUOUS PACKAGE IT IS THE APPROACH PROPOSED IN [33], FOR MISPLACED CLASS IT IS THE APPROACH PROPOSED IN [34].

Code Smell	TACO∩ST		TACO\ST		ST\TACO	
	#	%	#	%	#	%
Long Method	227	58%	116	30%	48	12%
Feature Envy	53	48%	30	28%	26	24%
Blob	79	40%	81	41%	37	19%
Promiscuous Package	26	29%	43	49%	19	22%
Misplaced Class	13	23%	35	63%	8	14%

case, the structural technique can not detect the instance because of the dependencies among the classes, due to the fact that all the classes inherit the `AbstractAction` class and belong to the menu visible by the end user. On the other hand, TACO fails in the detection of the `net.sourceforge.atunes.gui.views.controls` package, that is composed of 26 classes related to different aspects of the management of the dialogs of the application. The structural technique correctly detects the smell since it is able to cluster the classes into sub-packages, while TACO can not detect it because of the consistent vocabulary used in the classes. Looking at Table VII, we can see that the textual technique captures the most part of the instances missed by the alternative approach (i.e., 49%), while the structural technique detect 29% of instances missed by TACO. Finally, it is interesting to note how only the 23% of instances are detected by both the techniques.

E. Misplaced Class Discussion

Regarding the *Misplaced Class*, the textual technique reaches 77% of precision and 84% of recall (F-measure=81%), while the alternative approach has a precision of 53%, with a recall of 37% (F-measure=43%). This result clearly shows the usefulness of textual analysis for detecting classes not properly located. An example smell detected by TACO can be found in the `Apache Lucene` project, where the class `InstantiatedIndex` of the package `org.apache.lucene.store` has different dependencies with the current package, but has topics more related to the package `org.apache.lucene.index`. In contrast, the approach proposed in [34] is the only one able to detect the `mapReduceLayer.PhyPlanSetter` class as misplaced in the `Apache Pig` project. Here, the class has a vocabulary more similar to the package where it is actually in, but it should clearly be placed in the `physicalLayer` package. When considering the overlap metrics (Table VII), we confirm the actual superiority of TACO with respect the structural technique. Indeed, we found that 63% of correctly detected instances are only found by TACO, 23% of instances are detected by both approaches, while a smaller percentage (i.e., 14%) of smells are only identified by the alternative structural approach.

Summary for RQ$_1$: The proposed textual-based approach provides good performance in detecting code smells. Among the 10 projects, its precision ranges between 67% and 77%, while its recall between 72% and 84%.

Summary for RQ$_2$: 100% of the times TACO outperforms DECOR during the detection of *Long Method* instances, 90% of the times when considering the detection of classes affected by *Blob*. As for *Feature Envy*, TACO achieves better performances than JDeodorant in 70% of the cases. When applied to *Promiscuous Package* detection, 90% of the times our approach performs better with respect to the technique proposed in [33], while in all the cases TACO outperforms the technique proposed in [34] in the detection of *Misplaced Class* instances. Thus, on average, in the 90% of the cases TACO outperforms alternative structural-based approaches in detecting smells.

Summary for RQ$_3$: Our findings showed some complementarities between textual and structural information, suggesting that our novel approach and the structural code analysis techniques could nicely complement each other to obtain better performance in detecting smells. However, such combination is not trivial. As an example, considering the *Feature Envy* smell: We observed that a simple combination obtained by using AND/OR operators is not enough. Indeed, in the AND case the precision strongly increases to 84% (+17% than TACO, +27% than JDeodorant) but the recall decreases to 38% (-34% than TACO, -31% than JDeodorant); in the OR case the recall increases to 91% (+24% than TACO, +34% than JDeodorant), while the precision dramatically decreases to 33% (-39% than TACO, -36% than JDeodorant). The construction of a hybrid technique is part of our future agenda.

VI. THREATS TO VALIDITY

This section describes the threats that can affect the validity of our empirical study.

Construct Validity. Threats to construct validity are mainly due to the definition of the *oracle* for the studied software projects. In the context of our paper, we rely on the oracles publicly available in [61]. However, we cannot exclude that the oracle we used misses some smells, or else identified some false positives. Another threat is the use of comments during the detection process, since not all the systems have them. To deal with this, we re-run the case study by just considering

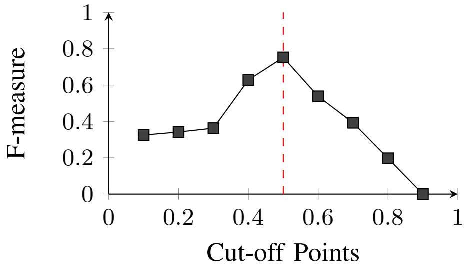

Fig. 2. F-measure achieved with different cut-off points. Dashed red line corresponds to our cut-off point.

identifiers. Results are in line with the ones obtained in Section V. The interested reader can find detailed results in our online appendix [60]. Finally, another threat is related to our re-implementation of both *SEGMENT* [57] and *DECOR* [22], which was needed because of lack of tools. However, our re-implementations use the exact algorithms defined by Wang *et al.* [57] and by Moha *et al.* [22], and have already been used in earlier work [24], [49], [62].

Internal Validity. An important factor that might affect our results is represented by the cut-off point we used to detect code smells. To have higher reliability of our choice, we also investigated the effects of different cut-off points on the performance of TACO when detecting smells for all the systems considered in our study. For example, Figure 2 plots the F-measure scores achieved by TACO when using different cut-off points when detecting *Long Method* on the `aTunes` project. We can notice that the best F-measure is achieved when using as cut-off point the median of the probability distribution, i.e., the dashed red line in Figure 2. Similar results are also obtained for the other projects and smell types, as reported in our online appendix [60]. Another threat to internal validity is represented by the settings used for the IR process. During the pre-processing, we filtered the textual corpus by using well known standard procedures: stop word list, stemmer and the *tf-idf* weighting schema, and identifiers splitting [50]. For LSI, we choose the number of concepts (k) using the heuristics proposed by Kuhn *et al.* [54].

External Validity. We demonstrated the feasibility of our approach focusing our attention on smells of different nature and of different levels of granularity. However, there might be other smells that can be potentially detected using TACO and not considered in this paper [2], [35]. Such an investigation is part of our future agenda. Another threat can be related to the number of subject systems used in our empirical evaluation. To show the generalizability of our results, we conducted an empirical study involving 10 Java open source systems having different size and different domains. It could be worthwhile to replicate the evaluation on other projects written in different programming languages.

VII. Conclusion and Future Directions

In this paper we presented TACO (**T**extual **A**nalysis for **C**ode smell detecti**O**n), an approach able to detect code smells. It does so by analyzing the properly decomposed textual blocks composing a code component, in order to apply IR methods and measuring the probability that a component is affected by a given smell. We demonstrated the usefulness of textual analysis for smell detection instantiating TACO for five code smells, i.e. *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package*, and *Misplaced Class*. The performance of the proposed textual detector has been assessed on 10 open source systems and compared with the ones provided by existing approaches solely relying on structural analysis. The results of the study demonstrated that TACO exhibits a precision ranging between 67% and 77%, and a recall that ranges between 72% and 84%, often outperforming alternative structural-based detectors. Moreover, we found some complementarity between textual and structural information, suggesting that their combination could be beneficial to obtain better detection accuracy. For the aforementioned reason, our future research agenda includes the definition of a combined technique, as well as the evaluation of the usefulness of the proposed textual approach in the detection of other code smell types. Also, we will further investigate the characteristics of textual smells in order to compare their impact on change- and fault-proneness with the results achieved in previous studies considering smells detected using structural techniques.

References

[1] D. L. Parnas, "Software aging," in *Proc. Int'l Conf on Software Engineering (ICSE)*. ACM/IEEE, 1994, pp. 279–287.

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

[6] 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.

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

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

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

[10] 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 *European Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 181–190.

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

[12] 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.

[13] 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.

[14] 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.

[15] A. Yamashita and L. Moonen, “Exploring the impact of inter-smell relations on software maintainability: An empirical study,” in *Proc. Int’l Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 682–691.

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

[17] 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.

[18] M. J. Munro, “Product metrics for automatic identification of “bad smell” design problems in java source-code,” in *Proc. Int’l Software Metrics Symposium (METRICS)*. IEEE, 2005, p. 15.

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

[20] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, “A bayesian approach for the detection of code and design smells,” in *Proc. Int’l Conf. on Quality Software (QSIC)*. IEEE, 2009, pp. 305–314.

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

[22] 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 Trans. on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

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

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

[25] 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.

[26] R. Buse and W. Weimer, “Learning a metric for code readability,” *IEEE Trans. on Softw. Engineering*, vol. 36, no. 4, pp. 546–558, July 2010.

[27] A. Marcus and J. I. Maletic, “Recovering documentation-to-source-code traceability links using latent semantic indexing,” in *Proc. Int’l Conf. on Software Engineering (ICSE)*. IEEE, 2003, pp. 125–135.

[28] A. Marcus, D. Poshyvanyk, and R. Ferenc, “Using the conceptual cohesion of classes for fault prediction in object-oriented systems,” *IEEE Trans. on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[29] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, “How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms,” in *Proc. Int’l Conf. on Software Engineering (ICSE)*. IEEE, 2013, pp. 522–531.

[30] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, “Automating extract class refactoring: An improved method and its evaluation,” *Empirical Softw. Engg.*, vol. 19, no. 6, pp. 1617–1664, Dec. 2014.

[31] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. de Lucia, “Improving software modularization via automated analysis of latent topics and dependencies,” *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 4:1–4:33, Feb. 2014.

[32] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, “Methodbook: Recommending move method refactorings via relational topic models,” *IEEE Trans. Softw. Engg.*, vol. 40, no. 7, pp. 671–694, July 2014.

[33] 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.

[34] 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.–.

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

[36] B. F. Webster, *Pitfalls of Object Oriented Development*, 1^{st} ed. M & T Books, February 1995.

[37] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[38] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, “Detecting defects in object-oriented designs: using reading techniques to increase software quality,” in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 1999, pp. 47–56.

[39] F. Simon, F. Steinbr, and C. Lewerentz, “Metrics based refactoring,” in *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2001, pp. 30–38.

[40] E. van Emden and L. Moonen, “Java quality assurance by detecting code smells,” in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2002.

[41] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, “Jdeodorant: Identification and removal of type-checking bad smells,” in *European Conf. on Softw. Maintenance and Reengineering (CSMR)*. IEEE, 2008, pp. 329–331.

[42] N. Tsantalis and A. Chatzigeorgiou, “Identification of extract method refactoring opportunities for the decomposition of methods,” *J. Syst. Softw.*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011.

[43] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, “Identification and application of extract class refactorings in object-oriented systems,” *J. Syst. Softw.*, vol. 85, no. 10, pp. 2241–2260, 2012.

[44] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, “Using structural and semantic measures to improve software modularization,” *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[45] 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.

[46] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, “A cooperative parallel search-based software engineering approach for code-smells detection,” *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, Sept 2014.

[47] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha, “Competitive coevolutionary code-smells detection,” in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8084, pp. 50–65.

[48] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, “Code-smell detection as a bilevel problem,” *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 6:1–6:44, Oct. 2014.

[49] F. Palomba, “Textual analysis for code smell detection,” in *Proc. Int’l Conf. on Software Engineering (ICSE)*. IEEE, 2015, pp. 769–771.

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

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

[52] 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*, no. 41, pp. 391–407, 1990.

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

[54] 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.

[55] T. DeMarco, *Structured Analysis and System Specification*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1979.

[56] M. Page-Jones, *The Practical Guide to Structured Systems Design: 2Nd Edition*. Upper Saddle River, NJ, USA: Yourdon Press, 1988.

[57] X. Wang, L. Pollock, and K. Vijay-Shanker, “Automatic segmentation of method code into meaningful blocks to improve readability,” in *Proc. Working Conf. Reverse Engineering (WCRE)*. IEEE, 2011, pp. 35–44.

[58] 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.

[59] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, “Using information retrieval based coupling measures for impact analysis,” *Empirical Softw. Engg.*, vol. 14, no. 1, pp. 5–32, 2009.

[60] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, “Online appendix: A textual-based technique for smell detection,” Tech. Rep., http://dx.doi.org/10.6084/m9.figshare.1590962.

[61] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia, “Landfill: An open dataset of code smells with public evaluation,” in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE, 2015, pp. 482–485.

[62] 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 *Int’l Conf. on Softw. Engineering (ICSE)*. IEEE, 2015, pp. 403–414.