Automatic Test Smell Detection using Information Retrieval Techniques

Fabio Palomba[1], Andy Zaidman[2], Andrea De Lucia[3]
[1]University of Zürich, Switzerland — [2]Delft University of Technology, The Netherlands — [3]University of Salerno, Italy

***Abstract*—Software testing is a key activity to control the reliability of production code. Unfortunately, the effectiveness of test cases can be threatened by the presence of faults. Recent work showed that static indicators can be exploited to identify test-related issues. In particular *test smells*, *i.e.*, sub-optimal design choices applied by developers when implementing test cases, have been shown to be related to test case effectiveness. While some approaches for the automatic detection of test smells have been proposed so far, they generally suffer of poor performance: as a consequence, current detectors cannot properly provide support to developers when diagnosing the quality of test cases. In this paper, we aim at making a step ahead toward the automated detection of test smells by devising a novel textual-based detector, coined TASTE (Textual AnalySis for Test smEll detection), with the aim of evaluating the usefulness of textual analysis for detecting three test smell types, *General Fixture*, *Eager Test*, and *Lack of Cohesion of Methods*. We evaluate TASTE in an empirical study that involves a manually-built dataset composed of 494 test smell instances belonging to 12 software projects, comparing the capabilities of our detector with those of two code metrics-based techniques proposed by Van Rompaey *et al.* and Greiler *et al.* Our results show that the structural-based detection applied by existing approaches cannot identify most of the test smells in our dataset, while TASTE is up to 44% more effective. Finally, we find that textual and structural approaches can identify different sets of test smells, thereby indicating complementarity.**

***Index Terms*—Test smells; Empirical Studies; Mining Software Repositories.**

I. Introduction

During the evolution of software systems, developers continuously apply new changes with the aim of improving existing features, developing new requirements, or fixing faults experienced by users [1], [2], [3]. Since such changes are often done while struggling with upcoming deadlines [4], [5] or without caring too much of the quality of the applied design choices [6], it becomes of paramount importance for developers to check whether their commits might possibly introduce new software faults [7], [8], [9], [10]. This check is the goal of regression testing [11], that consists of (possibly selectively) re-executing the test cases included in the test suite associated with the production code. Past and recent research showed that the results of regression testing are used by developers to decide on whether to merge a pull request [12] or to deploy the system [13], [14], [15].

Unfortunately, even test suites are sometimes affected by faults that might preclude the effective testing of software systems [16], [17]. For instance, a typical issue is flakiness [13], which appears when a test case exhibits both a passing and a failing result with the same code, being therefore not deterministic [13]. Recent papers [18], [19] showed how problems in test code might often be due to the presence of specific design problems introduced by developers: these are called *test smells* [20]. Similarly, Bell *et al.* [21] showed that static analysis can be effectively exploited to identify problems contained in test suites.

As a matter of fact, despite the findings achieved so far, software developers still have low support for diagnosing issues in test code [22], [23]. Indeed, on the one hand Automatic Static Analysis Tools (ASATs) [24] are often not effective in identifying issues in test cases [25]; on the other hand, test smell detectors have been (i) reported to suffer of poor performance [26] or (ii) tested in small-scale empirical studies that threaten their generalizability [27], [28]. Thus, there is still the need for automated approaches and/or empirical studies that help developers when diagnosing the quality of test cases.

In this paper we aim at making a step toward diagnosing the quality of test cases with automated tools. Specifically, following the several successful applications of textual analysis in the context of code smell detection [29], [30], refactoring [31], [32], [33], and improvement of the quality of automatically generated test cases [34], we conjecture that this source of information can also be successfully adopted for the detection of test smells, possibly overcoming the poor performance limitation of existing detectors [26], [27], [28].

To this aim, we devise a novel textual-based test smell detector called TASTE (**T**extual **A**naly**S**is for **T**est sm**E**ll detection), which uses Information Retrieval techniques [35] for detecting three test smell types, *i.e.*, *General Fixture*, *Eager Test*, and *Lack of Cohesion of Test Methods*.

We employ TASTE in an empirical study aimed at assessing its performance and complementarity with respect to two existing approaches based on structural analysis such as those proposed by Van Rompaey *et al.* [26] and Greiler *et al.* [27]. The study is conducted on a large-scale dataset involving 494 *manually validated* test smell instances belonging to 12 open-source software projects. The results of the study show that TASTE has good performance in detecting all the three test smells considered, reaching up to 85% in terms of F-Measure. Moreover, our approach is up to 44% more effective than the baselines, highlighting the high potential of textual analysis for the task of test smell detection and also confirming previous results on the low capabilities of code metrics-based detectors. Finally, we discover some complementaries between the two experimented approaches, *i.e.*, textual and structural information can capture different sets of correct test smell

instances, meaning that their combination can be exploited to improve the automatic identification of test smells.

Structure of the paper. Section II discusses the related literature. Section III presents the novel textual-based approach devised. Section IV reports the design of the empirical study, while Section V discusses the achieved results. In Section VI we discuss the threats that could affect the validity of our empirical studies. Finally, Section VII concludes the paper and highlights our future research agenda.

II. Related Work

Traditionally, the research community mainly focused on *code smells* [6], *i.e.*, design issues arising in production code. In this regard, several empirical investigations into the nature of code smells, their evolution, and their relevance for developers have been carried out [4], [30], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49]; At the same time, a number of approaches—relying on different types of sources of information and methodologies—have been proposed [50], [51], [52], [53], [54], [55], [56], [57], [58].

The investigations of test smells are more recent and started with Beck [15], who initially highlighted the importance of having well-designed test code. Following this seminal work, van Deursen *et al.* [59], [60] defined a catalog of 11 test smells together with refactoring operations aimed at removing them, which was later extended by Meszaros [61].

Based on these catalog, Van Rompaey *et al.* [26] introduced a code metric-based technique able to identify two test smells, namely *General Fixture* and *Eager Test*. However, the empirical evaluation of the approach—conducted on a single software project—revealed a low accuracy of this approach in the detection of both test smell types and showed the limitations of using structural analysis for detecting test smells.

Later on, Greiler *et al.* [27], [28] reported the high diffuseness of test smells affecting test fixtures. To deal with such diffuseness, they also devised an automated tool named TestHound, able to identify instances of six test smell types, *i.e.*, *General Fixture*, *Lack of Cohesion of Test Methods*, *Test Maverick*, *Vague Header Setup*, *Dead Fields*, and *Obscure In-Line Setup* [27]. They evaluated the approach through semi-structured interviews that revealed that TestHound might support developers when diagnosing the quality of test code, especially when it is used to identify instances of *General Fixture* and *Lack of Cohesion of Test Methods*.

As detailed later in the paper, we perform a complementary evaluation of the approaches defined by Van Rompaey *et al.* [26] and Greiler *et al.* [27]: while the aforementioned approaches were evaluated on few systems or used semi-structured interviews, our investigation involves a large dataset with the aim of more extensively evaluating the performances of their approaches. We also compare the performance of the textual-based approach we propose in this paper with the the one achieved by the approaches proposed by Van Rompaey *et al.* [26] and Greiler *et al.* [27].

Other previous work is mainly related to the empirical analysis of the diffusion and impact of test smells. For instance, Tufano *et al.* [5] conducted an empirical study aimed at measuring the perceived importance of test smells and their lifespan during the software life cycle. They found that test smells are usually introduced during the first commit involving the affected test classes, but more importantly in 80% of the cases they are never removed. Bavota *et al.* [19] found that test smells are highly diffused and that their presence has a strong negative impact on the maintainability of the affected classes. These findings were later confirmed in the context of automatically generated test code [62]. It is worth noting that, in the context of their empirical investigation, Bavota *et al.* [19] also defined a test smell detector for automatically detecting instances of six test smell types. However, this detector is only based on simple heuristics that have the goal of overestimating the presence of test smells, in order to ease the subsequent phase of manual validation aimed at removing false positives. Given its characteristics, this tool cannot be properly considered a real test smell detector and, for this reason, we do not consider it in our study.

Spadini *et al.* [63] have investigated the relation of test smells and the source code quality of production code, showing that there is also a clear relationship between the two.

Finally, Palomba and Zaidman [18] discovered that the presence of test smells can lead the test code to be non-deterministic, and that refactoring represents a key flakiness-fixing strategy. All these empirical studies motivate our work: indeed, test smells can be considered a threat to the reliability of software systems, and the definition of automated detection approaches can support developers in monitoring the health status of test cases.

III. Taste: A Textual-based Test Smell Detector

Textual analysis has been repeatedly reported as a useful source of information for code smell detection [29], [30], refactoring [31], [32], [33], and other software engineering tasks such as the improvement of the quality of automatically generated test cases [34]. For this reason, we believe that this might represent a good source of information also for the identification of test smells. We defined Taste (**T**extual **A**naly**S**is for **T**est sm**E**ll detection), a textual-based approach that applies some heuristics to identify three test smell types coming from the catalogs by van Deursen *et al.* [59] and Greiler *et al.* [27], in particular:

1) *General Fixture [59].* This smell arises when the `setUp` fixture is too general and different tests only access part of it. Van Deursen *et al.* [59] reported that this smell makes the comprehension of the test class harder.
2) *Eager Test [59].* This smell represents a test method that exercises more methods of the object under test, thus being not cohesive and harder to read and understand. Furthermore, it makes tests more dependent on each other and harder to maintain.
3) *Lack of Cohesion of Test Methods [27].* This smell occurs if test methods (excluding the `setUp` fixture) are grouped together in one test class but they are not cohesive, causing comprehensibility and maintainability issues.

We focused on these particular three test smells because of their diffuseness on both open and closed source systems and their harmfulness for maintainability [28], [19]. Furthermore, the choice was driven by the presence of automated test smell detectors that can identify instances of these smells, *i.e.*, the approaches defined by Van Rompaey *et al.* [26] and Greiler *et al.* [27], whose performance can be compared with the one achieved by the devised detector. Finally, they have characteristics that make the use of Information Retrieval techniques possible, allowing us to properly evaluate the usefulness of textual analysis for test smell detection. While other test smells can be detected using the technique by Greiler *et al.* [27], *i.e.*, *Test Maverick*, *Vague Header Setup*, *Dead Fields*, and *Obscure In-Line Setup*, we excluded them from our analysis since they are only partially relevant for developers [5], [27].

In the remaining of this section, we detail the general detection approach as well as the rules applied for the identification of each target test smell.

A. General Detection Process

Essentially, TASTE adopts three main steps to identify candidate test smell instances, *i.e.*, (i) extraction of textual components from test cases, (ii) text normalization, and (iii) detection rule application.

Starting from the set of JUnit test classes composing the software project under analysis, in the first step the approach extracts the textual content characterizing each test class. More specifically, source code identifiers and comments are taken as basis for the identification of test smells. These identifiers and comments are then normalized through the application of a standard Information Retrieval (IR) process. In particular, four steps are executed [64]: (i) composed identifiers are separated using a camel case splitting, which splits words based on underscores, capital letters and numerical digits; (ii) letters of extracted words are reduced to lower case to reduce noise; (iii) special characters, programming keywords and common English stop words are removed; and (iv) words are stemmed to their original roots by using the well-known Porter's stemmer [65]. Finally, the normalized words are weighted using the *term frequency - inverse document frequency* (*tf-idf*) schema [64], which reduces the relevance of too generic words that are contained in most source components.

The normalized textual content of each JUnit test class is then individually analyzed and different heuristics are applied to identify the three test smells considered. The actual detector relies on an implementation of the Latent Semantic Indexing (LSI) [66], that is an extension of the Vector Space Model (VSM) [64] and models test classes as vectors of terms occurring in a given software system. LSI relies on Singular Value Decomposition (SVD) [67] to cluster code components according to the co-occurrences among words and tests. Then, the original vectors (test classes) are projected into a reduced k space of concepts to limit the effect of textual noise. To set the size of the reduced space (k) we employed the heuristic proposed by Kuhn *et al.* [68], which provided good results in many software engineering applications: In particular, $k = (m \times n)^{0.2}$ where m denotes the vocabulary size and n denotes the number of documents (test classes 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. The following sections report the exact heuristics applied to detect the target test smells.

B. General Fixture Detection

This test smell is characterized by the presence of a too general `setUp` method [59]. We conjecture that *JUnit test classes affected by General Fixture present pairs of test methods having zero textual similarity even if they use objects instantiated in the setUp method.* Let $T = \{t_{setUp}, t_{tearDown}, t_1, \ldots, t_n\}$ be the set of methods of the JUnit test class under analysis, we say that a pair of test methods (t_i, t_j) is disjoint if the cosine similarity between t_i and t_j is zero and they use portions of the *setUp* method t_{setUp}. Therefore, we define the set *Disjoint_Pairs* as reported below:

$$\begin{aligned} \textit{Disjoint\_Pairs} \quad = \quad & \{(t_i, t_j) : sim(t_i, t_j) = 0 \\ & \wedge\ sim(t_i, t_{setUp}) \neq 0 \\ & \wedge\ sim(t_j, t_{setUp}) \neq 0 \\ & \wedge\ \ t_i, t_j \in T\} \end{aligned}$$

Starting from the definition above, we compute the probability of the JUnit test class T to be affected by a *General Fixture* according to the following equation:

$$P_{GF}(T) = \frac{\mid \textit{Disjoint\_Pairs} \mid}{\mid \{(t_i, t_j) : t_i, t_j \in T\} \mid} \tag{1}$$

where P_{GF} measures the percentage of pairs (t_i, t_j) that are disjoint.

C. Eager Test Detection

Since test methods affected by this smell check several methods of the object to be tested, our conjecture is that *they are methods in which there is a low textual similarity among the tested methods*. For this reason, we first replace the test method calls with the actual source code called by the test method. This is done since a test method has generally few lines of code and, thus, there are too few terms for setting up a similarity technique. More formally, let $t = \{m_1, \ldots m_n\}$ be the set of method calls performed by the JUnit test method under analysis, where m_i is the *i-th* method call in t. We first modify the test method by replacing all its method calls with the corresponding body of the called production methods, *i.e.*, the modified test method is $t' = \{m'_1, \ldots m'_n\}$ where the method call m_i is replaced by the corresponding production code body m'_i. Starting from t' we compute its textual cohesion [69] as the average similarity between its constituent methods m'_i as follows:

$$TestMethodCohesion(t) = \underset{i \neq j}{\text{mean}}\ sim(m'_i, m'_j) \quad (2)$$

where n is the number of method calls in t, and $sim(m'_i, m'_j)$ denotes the cosine similarity between two expanded method calls m'_i and m'_j in t'. Starting from our definition of textual cohesion of t, we compute the probability that t' is affected by *Eager Test* using the following formula:

$$P_{ET}(t) = 1 - TestMethodCohesion(t) \quad (3)$$

D. Lack of Cohesion of Test Methods Detection

This smell basically characterizes a JUnit class that is poorly cohesive. For this reason, we conjecture that instances of this smell can be textually identified by looking at *classes having a low textual similarity between the methods they contain*. Thus, let $T = \{t_{setUp}, t_{tearDown}, t_1, \ldots, t_n\}$ be the methods of the JUnit test class under analysis, we first compute the class cohesion using the following formula:

$$TestClassCohesion(T) = \underset{i \neq j}{\text{mean}}\ sim(m_i, m_j) \quad (4)$$

where $sim(m_i, m_j)$ denotes the cosine similarity between two test methods m_i and m_j in T. It is important to note that we exclude the `setUp` method during the evaluation of the test class cohesion since the definition of the smell does not refer to fixtures. Starting from the definition of textual cohesion of T, we then compute the probability that T is affected by a *Lack of Cohesion of Test Methods* using the following formula:

$$P_{LCTM}(T) = 1 - TestClassCohesion(T) \quad (5)$$

E. Turning the Probability into a Boolean Representation

All the detection rules applied by TASTE produce a probability that indicates the likelihood that a test class/method is affected by a certain test smell. In the context of the test smell detection problem, we needed to convert such probabilities in a boolean value {`true, false`} so that the performance of the devised approach can be compared to the one achieved by the code metrics-based approaches. To this aim, we empirically calibrated the optimal probability threshold to use by testing how the performance of TASTE (in terms of F-Measure) varies while variating the threshold. More specifically, we tried all settings from 0.1 to 0.9 in steps of 0.1. As a result, 0.6 was found as the optimal threshold for all the considered systems. Thus, we used it in our empirical study.

IV. EMPIRICAL STUDY DEFINITION AND DESIGN

This section defines the goal of our empirical study in terms of research questions and the subject systems we considered, and discusses the methodology followed to address the goals of our research.

TABLE I
CHARACTERISTICS OF THE SOFTWARE SYSTEMS IN OUR DATASET

System	Classes	Methods	JUnit Test Classes	KLOCs
Apache Ant 1.8.0	813	8,540	99	204
Apache Cassandra 1.1	586	5,730	130	111
HSQLDB 2.2.8	444	8,808	11	260
Apache Hive 0.9	1,115	9,572	13	204
Apache Ivy 2.1.0	349	3,775	80	58
Apache Log4j 1.1	349	3,775	54	58
Apache Lucene 3.6	2,246	17,021	297	466
Apache Karaf 2.3	470	2,678	70	56
Apache Nutch 1.4	259	1,937	52	51
Apache Pig 0.8	922	7,619	361	184
Apache Qpid 0.18	922	9,777	130	193
Apache Struts 3.0	1,002	7,506	85	152

A. Research Questions

The first *goal* of the study was to evaluate TASTE, with the purpose of understanding whether and to what extent the use of textual information can be useful to detect test smells in software projects. Hence, we define the first research question (RQ):

RQ$_1$. *To what extent can* TASTE *detect test smells in software systems*?

Once assessed the performance of TASTE, we then compare it with the alternative code metric-based approaches proposed by Van Rompaey *et al.* [26] and Greiler *et al.* [27]. In this case, the goal is to understand how our approach works when compared with the existing ones. This led to our second research question:

RQ$_2$. *How does* TASTE *work when compared with existing code-metric based approaches?*

Finally, we evaluated the extent to which the two sources of information experimented, *i.e.*, textual and structural, are complementary. Thus, we defined our last research question:

RQ$_3$. *To what extent do approaches based on textual and structural analysis complement each other*?

In the following we report the methodological steps conducted to answer our research questions.

B. Context Selection

The *context* of the study consisted of (i) test smells, (ii) software systems where to perform the experiments, and (iii) baseline code metric-based test smell detectors.

As for the former, we considered the three test smell types described in Section III, *i.e.*, *General Fixture*, *Eager Test*, and *Lack of Cohesion of Methods*.

As for the subject systems, Table I reports the characteristics of the 12 open-source systems analyzed[1], *i.e.*, their size in

[1]The list of repositories is available in our online appendix [70]

terms of number of classes, number of methods, number of JUnit test classes, and KLOC. The choice of the subject systems was driven by two main constraints: (i) the availability of the source code and (ii) the availability of test classes on which to run the test smell detectors. Hence, starting from the list of open-source projects available on GITHUB[2] we randomly selected 12 systems having at least 10 JUnit test classes and 50 KLOC. It is worth noting that we limited the analysis to systems developed in JAVA because the considered baseline detectors (as well as our textual-based smell detection approach) only work on this type of systems.

Finally, with respect to code metric-based approaches to use as baselines ($\mathbf{RQ}_2$), we selected the approaches devised by Van Rompaey *et al.* [26] and Greiler *et al.* [27]. The former was used to detect instances of *General Fixture* and *Eager Test*, while the latter was used to identify *Lack of Cohesion of Test Methods* instances. While also the approach by Greiler *et al.* [27] is able to identify *General Fixture* instances, we employed the one by Van Rompaey *et al.* [26] because of experimental tests (reported in our online appendix [70]) which showed its superiority in terms of accuracy of the detection: thus, in the scope of the paper we only report the results obtained by the best technique for the *General Fixture* detection.

More in detail, the test smell detector proposed by Van Rompaey *et al.* [26] computes four normalized size metrics, *i.e.*, Number of OBject Used in setup (NOBU), Number of Production-Type Uses (NPTU), Number of Fixture Objects (NFOB), and Average Fixture Usage of test methods (AFIU), to characterize the likelihood that a test fixture is too general: if the average value of these metrics exceeds 0.5, then it detects a smell instance. As for *Eager Test*, the technique computes the number of production calls made by a certain test method: if this sum is higher than 3, a test smell is detected.

The approach proposed by Greiler *et al.* [27] (TESTHOUND) identifies *General Fixture* instances by using the ratio between the number of setup fields a test method has and the total number of setup fields existing in the class: if it is higher than 0.7 the fixture is marked as smelly. To detect the *Lack of Cohesion of Test Methods* smell, the approach exploits an adjusted version of the Henderson-Seller Lack of Cohesion of Method metric [71]: specifically, it computes the cohesion between the test methods in a JUnit class excluding helper or setup methods (thus focusing only on the methods that actually exercise production code) and if this value is higher than 0.4 a smell instance is detected.

To the best of our knowledge, these two approaches are the only ones available for the detection of the considered set of smells. For this reason, their selection was driven by the choice of test smells on which we focused. It is important to note that we relied on the original implementation of the tools to avoid threats due to their re-implementation.

C. Oracle Construction

To answer our research questions, we needed an oracle reporting the actual test smell instances present in the considered systems. To the best of our knowledge, there is no annotated set of test smells available in literature. Thus, we had to manually build our own oracle. Starting from the definition of the test smells reported in literature [27], [59], we asked two external developers having more than 10 years of experience in testing industrial software systems[3] to independently inspect the projects under analysis with the aim of identifying test classes/methods affected by the three test smells considered. Specifically, they were provided with the source code of the considered projects and three spreadsheets, one for each test smell to analyze: (i) the first two spreadsheets contained the list of all the test classes belonging to the systems under analysis, and that were used to classify instances of the class-level smells, namely *General Fixture* and *Lack of Cohesion of Test Methods*; (ii) the third one contained all the test methods of the software projects analyzed, which were used to classify the *Eager Test* smell. The task was to assign a truth value in the set {`true, false`} to each code component present in the spreadsheets: the inspector assigned the value `true` when a code component was affected by a test smell, `false` otherwise. Once the inspectors had completed this task, the produced oracles were compared, and the inspectors discussed the differences, *i.e.*, smell instances present in the oracle produced by one inspector, but not in the oracle produced by the other. All the code components positively classified by both the inspectors were considered as actual smells. Regarding the other instances, the inspectors opened a discussion in order to resolve the disagreement and jointly took a decision. At the end of this process, the oracle comprised of 21 actual instances for *General Fixture*, 268 for *Eager Test*, and 205 for *Lack of Cohesion of Test Methods* (494 instances in total). To measure the level of agreement between the two inspectors, we computed the Jaccard similarity coefficient [72], *i.e.*, number of smell instances identified by both the inspectors over the union of all the instances identified by them. The overall agreement between the two inspectors before the discussion was 76%. Of the remaining 24% of the cases, they reached an agreement during the discussion. A spreadsheet reporting the data about the agreement computation is available in our replication package [70].

D. Methodology

Once the oracle was built, to answer $\mathbf{RQ}_1$ we ran TASTE on the set of test cases contained in the considered systems. It is worth noting that the detector applies its detection rules on the JUnit classes contained under the `test` folder of a software project, with the aim of excluding production classes. This step output a list of candidates for each considered test smell that we could compare against the oracle previously built.

To evaluate the performance of TASTE, we employed precision and recall [64], which are defined as follow:

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

[2]https://github.com

[3]Information omitted because of double-blind reviewing.

where TP and FP represent the set of true and false positive test smells detected by the detector, respectively, while FN (false negatives) represents the set of missed test smell instances contained in the oracle. 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 address our $\mathbf{RQ}_2$ and compare TASTE with the code-metric based approaches defined by Van Rompaey *et al.* [26] and Greiler *et al.* [27], we first needed to run such alternative detectors on the same set of test cases considered in $\mathbf{RQ}_1$. The list of candidate smells identified by the techniques as output of this step was then compared to the oracle, so that we could compute precision, recall, and F-Measure of the alternative approaches and understand whether and to what extent these indicators differ from those computed for TASTE.

Finally, to answer $\mathbf{RQ}_3$, we compared the sets of smell instances correctly detected by TASTE and by the baselines 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 test smells detected by the approach m_i, $correct_{m_i \cap m_j}$ measures the overlap between the set of true test smells detected by both approaches m_i and m_j, and $correct_{m_i \setminus m_j}$ appraises the true test smells detected by m_i only and missed by m_j. The latter metric provides an indication of how a test smell detection technique contributes to enriching the set of correct code smells identified by another approach, eventually revealing the complementarity between structural and textual information and paving the way for combination opportunities.

V. ANALYSIS OF THE RESULTS

In this section, we report the results that answer to our research questions. To avoid redundancies, we report the results for all the three research questions together, discussing each smell separately. We summarize the answers for each **RQ** at the end of the section.

Tables II, III, and IV show the results achieved by TASTE on the twelve subject systems for *General Fixture*, *Eager Test*, and *Lack of Cohesion of Test Methods*, respectively. At the same time, the tables report precision, recall, and F-Measure achieved by the alternative code metrics-based approaches. Besides the indications about the performance obtained on the single systems, we also report the overall performance of the detectors (row "Overall"), *i.e.*, the evaluation metrics achieved considering all the systems as a single one. When no instances of a certain smell were present in the oracle, it was not possible to compute the recall (division by zero), while the precision would be zero if at least one false positive is detected (independently from the number of false positives). In these cases a "-" is indicated in the project row.

In our online appendix [70] we provide a technical report in which we also included the number of true and false positive instances found by both TASTE and the alternative techniques. In addition, Table V reports values concerning overlap and differences between TASTE and the structural techniques: column "TASTE ∩ ST" reports the percentage of smell instances detected by both TASTE and the alternative structural approach; column "TASTE \ ST" reports the percentage of correct code smells correctly identified by TASTE but missed by the structural technique; finally, column "ST \ TASTE" reports the percentage of correctly code smells identified by the structural technique but not by TASTE. In the following, we discuss the results for each smell type.

A. General Fixture Discussion

Over the entire dataset, the inspectors that were in charge of discovering actual test smells only found 25 *General Fixture* instances. The achieved results in terms of test smells correctly identified, clearly indicate that TASTE overcomes the corresponding structural technique, *i.e.*, the one by Van Rompaey *et al.* [26]. In particular, TASTE is able to correctly identify 20 *General Fixture* instances out of the total 25, reaching a recall of 80%, while the structural technique has 24% of recall, since it only detects 6 instances correctly. Thus, our results (i) confirm previous observations made by Van Rompaey *et al.* [26] on the limited accuracy of their technique and (ii) show that textual analysis can be a useful source of information for the identification of *General Fixture* instances.

It is worth noting that the precision of TASTE is 57%, *i.e.*, 36% higher than the precision achieved by baseline. Even if the performance of our approach outperforms the considered baseline, one could observe that this level of precision results in a developer spending considerable amount of time to inspect candidate instances and discard the ones that are not affected by any smell. However, it is worth noting that, on average, the number of false positive candidate smells a developer should analyze is around 1, thus limiting the amount of extra effort spent by a developer. Interesting is the case of `Apache Pig`, where although our approach presents 5 false positive smells, it is able to reach the high value of 80% for the F-measure metric (recall of 80% and precision of 71%). It is worth discussing an example of *General Fixture* we found in the `Apache Struts` project and represented by the test class named `ActionContextCleanUpTest`. Here the textual content of the fixture is represented by two disjoint sets of concepts, *i.e.*, the first having the responsibility to set up an instance of the `InnerDispatcher` class, the second creating a `MockFilterChain` object. The two test methods belonging to the class access only one *or* the other concept when implementing their responsibilities. The structural approach cannot detect the smell since the metrics used for the detection, used to measure the normalized size of the fixture, are not enough for correctly classifying the instance

TABLE II
GENERAL FIXTURE - PERFORMANCE OF TASTE COMPARED TO THE ONE OF THE CODE METRICS-BASED TECHNIQUE PROPOSED BY VAN ROMPAEY *et al.* [26].

	TASTE			Code Metrics-based Technique		
	Precision	Recall	F-measure	Precision	Recall	F-measure
Apache Ant	50%	100%	67%	0%	0%	0%
Apache Cassandra	-	-	-	-	-	-
HSQLDB	34%	100%	50%	0%	0%	0%
Apache Hive	34%	100%	50%	0%	0%	0%
Apache Ivy	50%	50%	50%	0%	0%	0%
Apache Log4j	-	-	-	-	-	-
Apache Lucene	-	-	-	-	-	-
Apache Karaf	34%	100%	50%	0%	0%	0%
Apache Nutch	-	-	-	-	-	-
Apache Pig	71%	80%	75%	36%	34%	35%
Apache Qpid	50%	100%	67%	-	-	-
Apache Struts	67%	67%	67%	25%	34%	29%
Overall	**57%**	**80%**	**67%**	**21%**	**24%**	**23%**

as smelly. Indeed, the fixture has just two instantiated objects and it poorly uses non-test objects.

Some other interesting findings can be observed when considering the overlap between TASTE and the code metrics-based approach (see Table V). In this case, the technique by Van Rompaey *et al.* [26] is able to detect only 1 correct instance missed by TASTE, while on the other hand 71% of *General Fixture* smells are correctly detected by our approach only. Finally, a quite low percentage of smells, *i.e.*, 24%, are detected by both techniques. From a practical perspective, these results indicate that TASTE *"dominates"* the alternative technique, since it can correctly identify the vast majority of test smells detectable using code metrics. This observation is supported by an additional analysis that we performed: in particular, we tried to combine the results of the two techniques by exploiting AND/OR operators. In the AND case, we considered as smelly all the *General Fixture* candidates detected as such by both the approaches; in the OR case, we considered as smelly all candidates detected by one or the other approach. While the first combination obtained performance much lower than the single approaches (F-Measure≈23%), the OR combination has performance similar to TASTE (+1% of F-Measure): this confirms that the code metrics-based approach cannot contribute to the identification of test smells not identified by our textual-based technique.

To broaden the scope of the discussion, we can observe how some test smell instances cannot be detected neither by our approach nor by baselines. This means that further studies aimed at investigating the characteristics of such instances might help in the definition of more accurate detection rules.

B. Eager Test Discussion

Among the 268 actual test methods affected by *Eager Test* identified by the involved inspectors, the textual approach reaches an overall recall of 77% and an overall precision of 75% (F-measure of 76%), and it clearly outperforms the metrics-based technique used as baseline, which is able to correctly detect 39% of the affected components, with a precision of 58%. Also in this case, the use of structural metrics does not provide good detection performance, confirming previous findings in the field [26] as well as the need of alternative approaches for detecting test smells. For instance, the test method `testGetSliceFromLarge` belonging to the `TableTest` class of the `Apache Cassandra` project is affected by *Eager Test* since it tests the slicing operation performed against several columns in a database table, but also it verifies the presence of multiple index entries. The structural approach is not able to detect the method as smelly because its metric profile does not suggest the presence of a design problem: indeed, the number of methods of the production class invoked is 2 (*i.e.*, it is lower than the threshold used by the approach for detecting the smell). At the same time, relying on textual information, TASTE was able to correctly identify the *Eager Test* instance. The superiority of our approach is also confirmed by the complementarity data shown in Table V. In particular, we observed that in the 56% of the cases, our approach detects smell instances missed by the alternative one, while 30% of correct instances are detected by the metrics-based approach too. Finally, in 14% of the cases, the structural approach performs better than TASTE. As an example, in the `Apache Ant` project, the method `testDriverCaching` of the `SQLExecTest` actually tests three different methods of the production class, checking the presence of a specific key in a database as well as both the insertion and update of a value into a column. Given these characteristics, the code metrics-based approach [26] can properly identify the smell. Conversely, the terms used in the test method are all related to the management of a few operations on a database that share almost the same vocabulary: this is the reason why TASTE is not able to flag the method as smelly.

TABLE III
EAGER TEST - PERFORMANCE OF TASTE COMPARED TO THE ONE OF THE CODE METRICS-BASED TECHNIQUE PROPOSED BY VAN ROMPAEY *et al.* [26].

	TASTE			Code Metrics-based Technique		
	Precision	Recall	F-measure	Precision	Recall	F-measure
Apache Ant	78%	85%	82%	65%	44%	53%
Apache Cassandra	81%	91%	85%	58%	44%	50%
HSQLDB	100%	50%	67%	0%	0%	0%
Apache Hive	67%	50%	57%	34%	25%	29%
Apache Ivy	67%	67%	67%	100%	34%	50%
Apache Log4j	-	-	-	-	-	-
Apache Lucene	75%	79%	77%	63%	37%	46%
Apache Karaf	75%	75%	75%	25%	25%	25%
Apache Nutch	75%	86%	80%	50%	43%	46%
Apache Pig	74%	72%	73%	58%	41%	48%
Apache Qpid	60%	50%	55%	50%	34%	40%
Apache Struts	75%	63%	69%	59%	68%	63%
Overall	**75%**	**77%**	**76%**	**58%**	**39%**	**47%**

As done for *General Fixture*, we also tried to combine the results of the two approaches using AND/OR operators. The most interesting finding occurred for the technique exploiting the OR operator: it is indeed able to improve the F-Measure by 9% and 38% with respect to TASTE and the alternative approach, respectively, reaching 85% as overall F-Measure. Thus, we confirm that textual and structural information can be complemented and that such a combination can lead to better detection performance.

C. Lack of Cohesion of Test Methods Discussion

The inspectors identified 205 instances of this test smell over the considered systems. Overall, our approach reaches an F-Measure of 62% (precision=63% and recall=60%) and, also in this case, it outperforms the alternative code metrics-based approach proposed by Greiler *et al.* [27]. While this result definitively confirms that the use of textual analysis can be beneficial for the task of test smell detection, it is also important to note that in this case the improvement achieved is lower than the previous test smells: indeed, TASTE has an F-Measure 9% higher than the baseline.

An interesting example of a test smell properly classified as such by TASTE is the `ActionMappingTest` class belonging to the `Apache Struts` project. This class can be considered poorly cohesive since it contains test cases aimed at exercising both the actual mapping of the information needed to invoke the framework and the setting of the external resources required for running it. In this case, the analysis of the textual cohesion of the class can reveal the presence of a smell because the methods contained in the class have a significantly scattered vocabulary: for instance, in the class we find methods like `testGetMapping` and `testGetUriParam`, which do not contribute to the implementation of a single responsibility. As for the structural analysis done by the baseline approach, it cannot identify this test smell instance because the methods share a number of field accesses that preclude the proper detection of the smelliness of the class.

Besides the performance analysis of TASTE with respect to the technique by Greiler *et al.* [27], further interesting results come from the analysis of the overlap between the two approaches. As shown in Table V, this is the only case in which we can observe a high complementarity: indeed, TASTE identifies 43% of *Lack of Cohesion of Test Methods* instances that the baseline approach cannot detect, while another 38% of them are missed by our approach and correctly identified only by the alternative one. Finally, only 19% of the instances are identified by both the techniques. This result indicates that textual and structural analysis are almost equally important and, more importantly, they seem to be used in a complementary way in order to improve the detection performance for this smell type.

To test this hypothesis, we combined the results of the two approaches using AND/OR operators as explained for the other smells. Interesting, we discover that by taking all the results given by the approaches (OR combination) it is possible to improve the performance by up to 25%, leading to an overall F-measure equal to 87%. Thus, we can conclude that the complementarity between TASTE and the technique by Greiler *et al.* [27] can be effectively exploited by means of a simple combination of the candidate smells output by both the approaches. Nevertheless, as part of our future agenda we plan to further investigate new ways to better combine complementary techniques.

Summary for RQ$_1$. The devised textual-based approach can identify *General Fixture*, *Eager Test*, and *Lack of Cohesion of Test Methods* instances with an overall F-Measure of 67%, 76%, and 62%, respectively. We also observed that in some cases TASTE can achieve even higher

TABLE IV
LACK OF COHESION OF TEST METHODS - PERFORMANCE OF TASTE COMPARED TO THE ONE OF THE CODE METRICS-BASED TECHNIQUE PROPOSED BY GREILER *et al.* [27].

	TASTE			Code Metrics-based Technique		
	Precision	Recall	F-measure	Precision	Recall	F-measure
Apache Ant	64%	65%	65%	59%	61%	60%
Apache Cassandra	61%	58%	59%	55%	45%	50%
HSQLDB	67%	65%	64%	52%	44%	48%
Apache Hive	60%	60%	60%	55%	56%	56%
Apache Ivy	65%	55%	60%	45%	47%	46%
Apache Log4j	60%	58%	59%	58%	54%	56%
Apache Lucene	55%	50%	53%	51%	47%	49%
Apache Karaf	64%	60%	62%	55%	60%	48%
Apache Nutch	65%	60%	63%	63%	57%	60%
Apache Pig	68%	60%	64%	58%	58%	58%
Apache Qpid	63%	68%	65%	60%	55%	58%
Apache Struts	58%	65%	61%	44%	44%	44%
Overall	**63%**	**60%**	**62%**	**55%**	**52%**	**53%**

TABLE V
OVERLAP BETWEEN TASTE AND CODE METRICS-BASED TECHNIQUES (ST). FOR GENERAL FIXTURE AND EAGER TEST IT IS THE APPROACH PROPOSED IN [26], WHILE FOR LACK OF COHESION OF TEST METHODS (LCTM) IT IS THE TECHNIQUE BY GREILER *et al.* [27].

Test Smell	TASTE∩ST		TASTE\ST		ST\TASTE	
	#	%	#	%	#	%
General Fixture	5	24%	15	71%	1	5%
Eager Test	80	30%	151	56%	37	14%
LCTM	38	19%	89	43%	78	38%

values for all the considered evaluation metrics.

Summary for RQ$_2$. TASTE improves upon the code metrics-based techniques by up to 44%, 29%, 9%, respectively, for the three smell types. Our results show that textual information is more suitable than structural one for the identification of the three test smell types considered in the study.

Summary for RQ$_3$. Textual and structural information is sometimes complementary. In case of *General Fixture*, we observe that TASTE detects almost all the instances identified by the alternative approach, dominating it. As for *Eager Test* and *Lack of Cohesion of Test Methods*, instead, we find that the two approaches detect different sets of test smells: by exploiting such complementarity through an OR based combination, we can improve the detection performance of TASTE by up to 9% and 25%, respectively. Nevertheless, we observe that there exist other actual test smell instances that cannot be identified by any of the experimented approaches, calling for future investigations aimed at understanding the nature of such instances.

VI. THREATS TO VALIDITY

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

A. Construct Validity

Threats in this category are related to the relation between theory and observation. The main threat in this category is related to the way we defined the oracle of test smells in the studied software projects. With the aim of avoiding bias from our side, we asked two external developers—having more than ten years of testing experience—to build an oracle reporting the actual instances of *General Fixture*, *Eager Test*, and *Lack of Cohesion of Test Methods*. To ensure a fair process the two inspectors separately classified test fixtures and test classes/methods as smelly or not and solved instances with disagreement via an open discussion to achieve a shared decision. Moreover, the oracles were built before producing (and knowing) test smell instances detected by our tool as well as by the alternative structural approaches. However, we cannot exclude that test smell oracles can be subject to imprecision and incompleteness. As part of our future research agenda we plan to enlarge our study, possibly increasing the confidence on the oracle built so far by involving other external developers in the evaluation.

The code metrics-based approaches employed in the study, *i.e.*, the ones by Van Rompaey *et al.* [26] and Greiler *et al.* [27], represent the original implementations made available by the corresponding authors. In this way, we avoided possible threats due to re-implementation.

B. Internal Validity

An important factor that might affect our results is represented by the threshold we used to detect test smells.

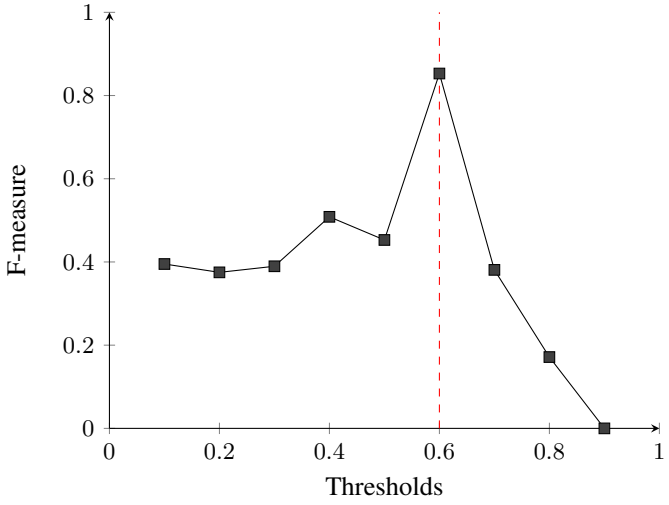

Fig. 1. F-measure scores achieved with different thresholds. Dashed red line corresponds to our threshold.

In the study, we adopted a threshold of 0.6 to distinguish smelly and non-smelly test cases. Such a threshold came from experimental results: specifically, as reported in Section III, we investigated the effects of different thresholds on the performance of TASTE when detecting test smells for all the systems considered in our study. For example, Figure 1 plots the F-measure scores achieved by TASTE when using different thresholds when detecting *Eager Test* on the `Apache Cassandra` project. The best F-measure is achieved when using 0.6 as threshold, *i.e.*, the dashed red line in Figure 1. Similar results are also obtained for the other projects and test smell types, as reported in our online appendix [70].

Another threat to internal validity is represented by the settings used for the IR process. During the preprocessing, we filtered the textual corpus by using standard procedures that have been widely-adopted in previous literature: stop word list, stemmer and the *tf-idf* weighting schema, and identifiers splitting [64]. For LSI, we choose the number of concepts (k) based on the heuristics proposed by Kuhn *et al.* [68].

Finally, the textual-based technique devised relies on both identifiers and comments when detects test smells. However, in a real-case scenario source code comments might be not available, possibly threatening the practicability of TASTE. To better understand the extent to which our technique can suffer of this issue, we completely re-run our empirical study just considering identifiers as source of information to be exploited for the textual-based identification of test smells. Results are in line with the ones discussed in Section V. A report of this analysis is available in our online appendix [70].

C. External Validity

In the context of this paper, we started analyzing the feasibility of the use of textual information for detecting three specific types of test smells, *General Fixture*, *Eager Test*, and *Lack of Cohesion of Test Methods*. They have different levels of granularity and have been reported in literature as pretty harmful for the maintainability of test code [28], [19]. We are aware that there might be other test smells that can be potentially detected using TASTE and not considered in this paper [59], [61]. Such an investigation is part of our future reseach agenda. Another threat can be related to the number of object systems used in our empirical evaluation. To show the generalizability of our results, we conducted a large empirical study involving twelve 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 as well as coming from an industrial context.

VII. CONCLUSION

In this paper we investigated whether and to what extent textual information can be effectively adopted for the task of test smell detection. To this aim, we defined TASTE (**T**extual **A**naly**S**is for **T**est sm**E**ll detection), an approach able to detect test smells by only relying on the textual component of source code. We instantiated our approach on three types of test smells that have been shown to be highly diffused and harmful for developers, *i.e.*, *General Fixture*, *Eager Test*, and *Lack of Cohesion of Test Methods*.

We demonstrated the validity of TASTE in an empirical study, where we (i) compared its performance with the one achieved by two baseline code metrics-based approaches such as those proposed by Van Rompaey *et al.* [26] and Greiler *et al.* [27] and (ii) evaluated the complementary between textual and structural information. The study was conducted on a set of twelve software projects, and in particular on 494 manually validated test smells.

Summing up, the contributions made by this paper are:

1) A novel approach able to exploit Information Retrieval methods for detecting instances of the three considered test smell types;
2) An large-scale empirical study on the performance of TASTE, which reveals that our approach can be more effective than the alternative code metrics-based approaches by up to 44%. As a second take, the study also showed that textual and structural analysis can be nicely complemented for two test smells, *i.e.*, *Eager Test* and *Lack of Cohesion of Test Methods*, reaching higher performance.
3) A replication package that contains the manually-validated set of test smells built in the context of this research and that can be used by other researchers for experimenting further improvements in the field of test smell detection. Moreover, it also includes the scripts used for our analysis [70].

Based on the findings achieved so far, our future research agenda includes the experimental analysis of more sophisticated and accurate combined techniques, as well as the evaluation of the usefulness of the proposed textual approach in the detection of other test smell types. Also, we plan to define a test smell prioritization approach that exploits the smelliness probability computed by TASTE.

REFERENCES

[1] M. M. Lehman, J. F. Ramil, P. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view," in *4th IEEE International Software Metrics Symposium (METRICS 1997), November 5-7, 1997, Albuquerque, NM, USA*, 1997, p. 20.

[2] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, "Enhancing change prediction models using developer-related factors," *Journal of Systems and Software*, vol. 143, pp. 14–28, 2018.

[3] G. Catolino and F. Ferrucci, "Ensemble techniques for software change prediction: A preliminary investigation," in *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), 2018 IEEE Workshop on*. IEEE, 2018, pp. 25–30.

[4] 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 (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[5] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 4–15.

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

[7] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 33.

[8] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *International Conference on Product Focused Software Process Improvement*. Springer, 2010, pp. 3–16.

[9] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[10] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production & test code," in *Proc. Int'l Conference on Software Testing, Verification, and Validation (ICST)*, 2008, pp. 220–229.

[11] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[12] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 358–368.

[13] J. Micco, "Flaky tests at Google and how we mitigate them," 2016, last visited, March 24th, 2017. [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[14] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*. ACM, 2017.

[15] Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[16] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 101–110.

[17] D. Hao, L. Zhang, H. Zhong, H. Mei, and J. Sun, "Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 683–686.

[18] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–12.

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

[20] A. Van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, 2001, pp. 92–95.

[21] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," 2018.

[22] R. V. Binder, "Testing object-oriented software: a survey," *Software Testing, Verification and Reliability*, vol. 6, no. 3-4, pp. 125–252, 1996.

[23] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Trans. Software Eng.*, vol. 40, no. 11, pp. 1100–1125, 2014.

[24] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Proc. Int'l Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 470–481.

[25] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, 2008.

[26] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[27] M. Greiler, A. van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 322–331.

[28] M. Greiler, A. Zaidman, A. van Deursen, and M.-A. Storey, "Strategies for avoiding text fixture smells during software evolution," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 387–396.

[29] F. Palomba, "Textual analysis for code smell detection," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 769–771.

[30] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering*, 2017.

[31] G. Bavota, A. 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.

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

[33] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. 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.

[34] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 130–141.

[35] G. G. Chowdhury, "Natural language processing," *Annual review of information science and technology*, vol. 37, no. 1, pp. 51–89, 2003.

[36] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 181–190.

[37] 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.

[38] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 612–621.

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

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

[41] 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.

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

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

[44] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.

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

[46] ——, "Do code smells reflect important maintainability aspects?" in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 306–315.

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

[48] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, pp. 1–34, 2017.

[49] ——, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, 2018.

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

[51] 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.

[52] 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.

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

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

[55] 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.

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

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

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

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

[60] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "On the interplay between software testing and evolution and its effect on program comprehension," in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 173–202.

[61] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, 2007.

[62] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*. ACM, 2016, pp. 5–14.

[63] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018.

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

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

[66] 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.

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

[68] 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.

[69] 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.

[70] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using natural language processing - online appendix," 2018. [Online]. Available: https://figshare.com/s/95951701d0f21177eac6

[71] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.

[72] P. Jaccard, "Etude comparative de la distribution florale dans une portion des alpes et des jura," *Bulletin de la Société Vaudoise des Sciences Naturelles*, no. 37, 1901.