The Scent of a Smell: An Extensive Comparison between Textual and Structural Smells

Fabio Palomba[1], Annibale Panichella[2], Andy Zaidman[1], Rocco Oliveto[3], Andrea De Lucia[4]
[1]TU Delft, The Netherlands, [2]SnT Centre — University of Luxembourg, Luxembourg,
[3]University of Molise, Italy — [4]University of Salerno, Italy
f.palomba@tudelft.nl, annibale.panichella@uni.lu, a.e.zaidman@tudelft.nl
rocco.oliveto@unimol.it, adelucia@unisa.it

Abstract—Code smells are symptoms of poor design or implementation choices that have a negative effect on several aspects of software maintenance and evolution, such as program comprehension or change- and fault-proneness. This is why researchers have spent a lot of effort on devising methods that help developers to automatically detect them in source code. Almost all the techniques presented in literature are based on the analysis of structural properties extracted from source code, although alternative sources of information (*e.g.,* textual analysis) for code smell detection have also been recently investigated. Nevertheless, some studies have indicated that code smells detected by existing tools based on the analysis of structural properties are generally ignored (and thus not refactored) by the developers. In this paper, we aim at understanding whether code smells detected using textual analysis are perceived and refactored by developers in the same or different way than code smells detected through structural analysis. To this aim, we set up two different experiments. We have first carried out a software repository mining study to analyze how developers act on textually or structurally detected code smells. Subsequently, we have conducted a user study with industrial developers and quality experts in order to qualitatively analyze how they perceive code smells identified using the two different sources of information. Results indicate that textually detected code smells are easier to identify and for this reason they are considered easier to refactor with respect to code smells detected using structural properties. On the other hand, the latter are often perceived as more severe, but more difficult to exactly identify and remove.

Index Terms—Code Smells, Empirical Study, Mining Software Repositories

1 Introduction

Technical debt is a metaphor introduced by Cunningham in 1993 to indicate *"not quite right code which we postpone making it right"* [25]. The metaphor tries to explain the compromise of delivering the most appropriate but still immature product, in the shortest time possible [25], [18], [53], [60], [92]. *Code smells*, *i.e.,* symptoms of poor design and implementation choices applied by programmers during the development of a software project [35], represent an important factor contributing to technical debt [53]. The research community spent a lot of effort studying the extent to which code smells tend to remain in a software project for long periods of time [4], [22], [61], [84], as well as their negative impact on non-functional properties of source code, such as program comprehension [1], change- and fault-proneness [48], [49], testability [77], [68] and, more generally, maintainability [97], [111], [109]. As a consequence, several tools and techniques have been proposed to help developers in detecting code smells and to suggest refactoring opportunities [12], [8], [65], [67], [74], [75], [100].

So far, almost all detectors try to capture code smell instances using structural properties of source code as the main source of information. However, recent studies have indicated that code smells detected by existing tools are generally ignored (and thus not refactored) by the developers [4], [10], [94]. A possible reason is that developers do not perceive the code smells identified by the tool as actual design problems or, if they do, they are not able to practically work on such code smells. In other words, there is misalignment between what is considered smelly by the tool and what is actually refactorable by developers.

In a previous paper [75], we introduced a tool named TACO that uses textual analysis to detect code smells in source code. The performances of this novel tool have been compared with the ones of traditional structural-based tools. Besides showing that TACO has good performances, the results indicate that textual and structural techniques are complementary: while some code smell instances in a software system can be correctly identified by both TACO and the alternative structural approaches, other instances can be only detected by one of the two alternative approaches [75].

In this paper, we investigate whether code smells detected using textual information are as difficult to identify and refactor as structural smells or if they follow a different pattern during software evolution. To this aim, we conducted two different studies investigating how developers **act on** code smell instances of the same type but detected either by TACO or by the structural-based tools (but not both). Our conjecture is that code

smells detected using textual analysis are easier to identify and refactor by developers with respect to code smells detected by structural-based tools.

To verify our hypotheses, we firstly performed a software repository mining study considering 301 releases and 183,514 commits from 20 open source projects in order (i) to verify whether textually and structurally detected code smells are treated differently, and (ii) to analyze their likelihood of being resolved with regards to different types of code changes, *e.g.,* refactoring operations. Since our quantitative study cannot explain relation and causation between code smell types and maintenance activities, we perform a qualitative study with 19 industrial developers and 5 software quality experts in order to understand (i) how code smells identified using different sources of information are perceived, and (ii) whether textually or structurally detected code smells are easier to refactor. In both studies, we focused on five code smell types, *i.e., Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, and *Promiscuous Package*.

The results of our studies indicate that textually detected code smells are perceived as harmful as the structural ones, even though they do not exceed any typical software metrics' value (*e.g.,* lines of code in a method). Moreover, design problems in source code affected by textual-based code smells are easier to identify and refactor. As a consequence, developers' activities tend to decrease the intensity of textual code smells, positively impacting their likelihood of being resolved. Vice versa, structural code smells typically increase in intensity over time, indicating that maintenance operations are not aimed at removing or limiting them. Indeed, while developers perceive source code affected by structural-based code smells as harmful, they face more problems in correctly identifying the actual design problems affecting these code components and/or the right refactoring operation to apply to remove them.

Structure of the paper. Section 2 introduces the textual and structural code smell detection techniques exploited to identify the two categories of code smells object of our study. In Section 3 we report the design and the results of the software repository mining empirical study where we analyze how code smells detected using different sources of information are treated during their evolution. Section 4 reports the design and results of the study aimed at understanding the developers' perception of structurally and textually detected code smells. Section 5 discusses the related literature on code smells, while Section 6 concludes the paper.

2 Textual and Structural Code Smell Detection

Starting from the definition of design debt proposed in [19], [35], [85], [107], researchers have devised tools and techniques to detect code smells in software systems. Most of them are based on the analysis of the structural properties of source code (*e.g.,* method calls) and on the combination of structural metrics [51], [56], [65], [67], [69], [72], [88], [95], [99], [103], while in recent years the use of alternative sources of information (*i.e.,* historical and textual analysis) have been explored [74], [75], together with methodologies based on machine learning [3], [33] and search-based algorithms [17], [46], [47], [87].

Besides code smell detectors, refactoring techniques may be also adopted to identify code smells in source code [11], [12], [16], [30], [100], [101]. Rather than identifying code smell instances directly, such approaches recommend refactoring operations aimed at removing a code smell. Also in this case, the primary source of information exploited is the structural one [30], [100], [101], while few works have explored a combination of structural and textual analysis [11], [12], [16].

Table 1 briefly describes the code smells considered in this study. Basically, we focus on code smells characterizing *poorly cohesive* code elements, *i.e., Long Method* [35], *Blob* [35], and *Promiscuous Package* [36], and *misplaced* code elements, *i.e., Feature Envy* [35] and *Misplaced Class* [36]. All of them belong to the initial catalog defined by Fowler [35] or its newer version available on-line [36].

More specifically, the *Long Method* affects methods implementing a main functionality together with other auxiliary functions that should be placed in other methods [35]. The *Blob* is a class containing methods implementing two or more different responsibilities [35], while the *Promiscuous Package* is a package containing classes implementing different unrelated responsibilities [13], [36]. *Feature Envy* affects methods having more relationships with other classes with respect to the class they are actually in [35]. Finally, the *Misplaced Class* represents a class located in a package that contains other classes that are not related to its function [36], [82], [97]. The interest in these code smells is dictated by the fact that they have been recognized as important threats to the maintainability of a software system [1], [40], [48], [50], [97], [109], but also because they are considered harmful by developers [73].

In our study, we consider a smell *textual* when it is detected using a textual-based detection technique, *i.e.,* it is characterized by high textual scattering among the elements it contains (*e.g.,* textual content of methods or statements). On the other hand, a code smell is *structural* if it is detected by a detector purely based on the analysis of structural properties of source code (*e.g.,* number of attributes, size or number of dependencies with other classes). The following subsections describe the detection rules applied in the context of our empirical study.

2.1 Textual-based Code Smell Detection

Only one technique is able to identify all the code smells considered in our study by solely relying on textual analysis, namely **TACO** (**T**extual **A**nalysis for **C**ode smell de**T**ection) [75]. TACO follows a three-step process: (i) textual content extraction, (ii) application of Information

TABLE 1
The Code Smells considered in our Study

Name	Description
Blob	A large class implementing different responsibilities [35].
Feature Envy	A method is more interested in a class other than the one it actually is in [35].
Long Method	A method that implements more than one function [35].
Misplaced Class	A class that should be placed in another package [36], [82], [97].
Promiscuous Package	A large package composed of sets of classes implementing different functionalities [13], [36].

Retrieval (IR) normalization process, and (iii) application of specific heuristics in order to detect code smells related to promiscuous responsibilities (*e.g.,* Blob).

In the first step, the approach extracts all textual elements needed for the textual analysis process of a software system, *i.e.,* source code identifiers and comments. Then, the approach applies a standard IR normalization process [7] aimed at (i) separating composed identifiers, (ii) reducing to lower case letters the extracted words, (iii) removing special characters, programming keywords, and common English stop words, and (iv) stemming words to their original roots via Porter's stemmer [83]. The code smell detection process relies on Latent Semantic Indexing (LSI) [27], an extension of the Vector Space Model (VSM) [7], that models code components as vectors of terms occurring in a given software system. LSI uses Singular Value Decomposition (SVD) [24] to cluster code components according to the relationships among words and among code components (co-occurrences). The original vectors (code components) are then projected into a reduced *k* space of concepts to limit the effect of textual noise. To this aim, TACO uses the well-known heuristic proposed by Kuhn *et al.* [54], *i.e.,* $k = (m \times n)^{0.2}$ where m denotes the vocabulary size and n denotes the number of documents (code components). In the third step code smells are detected by measuring the lack of textual similarity among their constituent code components (*e.g.,* vectors) using the cosine distance.

Following such a process, a *Blob* is detected (i) by computing the average similarity among the methods of the class, which corresponds to the conceptual cohesion of a class defined by Marcus and Poshyvanyk [63]; and (ii) by applying the following formula measuring the probability P_B that a class is affected by the *Blob* code smell:

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

where $ClassCohesion(C)$ represents the textual cohesion of the class C [63]. Using the same steps, TACO is able to detect *Long Method* instances. Specifically, the code blocks composing a method are firstly extracted exploiting the approach by Wang *et al.* [105]. This approach is able to automatically segment a method into a set of "*consecutive statements that logically implement a high-level action*" [105]. Once TACO identifies the sets of statements (*i.e.,* segments) composing the method, it considers each of them as a single document. Then, the probability a method is smelly is measured by applying the following formula:

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

where $MethodCohesion(M)$ represents the textual cohesion of the method M and it is computed as the average similarity among the segments composing a method.

Instances of *Promiscuous Package* are instead detected by exploiting the lack of cohesion among the classes composing a package. In particular, TACO applies the following formula:

$$P_{PP}(P) = 1 - PackageCohesion(P) \quad (3)$$

where $PackageCohesion(P)$, *i.e.,* the textual cohesion of the package P, is computed as the average similarity among the classes of the considered package.

On the other hand, to detect the *Feature Envy* code smell, for each method M belonging to the class C_O, the approach firstly retrieves the more similar class ($C_{closest}$) by computing the textual similarity between M and the set of classes in the system sharing at least one term with M. Then, the probability that the method is smelly is given by the difference between the textual similarities of M and the two classes $C_{closest}$ and C_O:

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

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 $C_{closest}$ and C_O. Finally, TACO identifies *Misplaced Class* instances by retrieving the package $P_{closest}$ (*i.e.,* the more similar package) for a class C contained in the package P_O, and then computing the probability that this class is misplaced by measuring the difference between the textual similarities of C and the two packages $C_{closest}$ and C_O:

$$P_{MC}(C) = sim(C, P_{closest}) - sim(C, P_O) \quad (5)$$

Also in this case, 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.

2.2 Structural-based Code Smell Detection

Related literature proposes a large variety of techniques able to detect code smells from a structural point of view [28]. However, none of them can simultaneously detect all the code smells considered in our study. Therefore, we had to select more than one technique to carry out our investigation. Given our definition of structural code smells, we discarded all the approaches that use a combination of more sources of information (*e.g.,* the techniques by Bavota *et al.* [8], [11], [12]), as well as the approaches using other types of information (*e.g.,* the change history information [74]). Furthermore, we avoided the use of industrial tools such as inCode [43] and iPlasma [44], and code quality checkers (*e.g.,* PMD[1] or Checkstyle[2]) for two main reasons. In the first place, for most of them there is no available empirical evaluation about their detection accuracy; secondly, even though some tools (*e.g.,* inCode) are inspired by the detection strategies proposed by Marinescu [65], they have an accuracy comparable to tools exploited in this study [28], which are described in the following.

Given the size of our empirical studies, we selected the structural code smell detection tools that provided the best compromise between detection accuracy and computational performance. For this reason, where possible we selected code smell detection tools (*e.g.,* [67]) instead of refactoring recommenders. In particular, we did not use *Extract Method* [93], [101] and *Extract Class* [9], [12], [14], [30] refactoring tools, because they use more computationally expensive algorithms to recommend one or more possible splittings of a method or class (note that the refactoring recommendation would also indicate the possible detection of a code smell). As a result, we selected *DECOR* [67] for the detection of *Long Method* and *Blob* instances because (i) it has been employed in several previous investigations on code smells demonstrating good performance [32], [34], [41], [50], [72], [74]; and (ii) it is simple to re-implement as its detection rules are based on the analysis of code metrics extractable from source code. This approach uses a set of rules, called rule cards[3], describing the characteristics a code component should have in order to be classified as smelly. In practice, rules are sets of conditions based on code metrics (*e.g.,* line of codes) with respect to fixed thresholds. In the case of Blob, a smelly instance is detected when a class has an LCOM5 (Lack of Cohesion Of Methods) [89] higher than 20, a number of methods and attributes higher than 20, and it has a *one-to-many* association with data classes. Note that while the original rule card proposed by Moha *et al.* also incorporates a textual rule to select the classes having the role of controllers (*i.e.,* classes that manage the processing of other classes [67]), in the context of this paper we excluded that part in order to obtain a pure structural-based detector. As for *Long Method*, DECOR classifies a method as affected by the code smell if it has more than 100 lines of code.

In general, tools detecting the *Feature Envy* code smell [8], [88], [100] are also *Move Method* refactoring tools, although algorithms used to recommend refactoring solutions are more lightweight than the decomposition algorithms used by *Extract Method* and *Extract Class* refactoring tools. We selected *JDeodorant* [100] as there are no other structural-based tools able to detect this emphMove Method smell with a comparable accuracy [28]. Moreover, it is important to note that other refactoring tools having similar performance to *JDeodorant*, *e.g., MethodBook* [8] and *JMove* [88], (i) rely on a combination of conceptual and structural analysis or (ii) require some parameter tuning. Thus, they are not suitable for our purposes. Given a method M, *JDeodorant* forms a set T of candidate target classes where M might be moved. This set is obtained by examining the entities (*i.e.,* attributes and methods) that M accesses from the other classes. In particular, each class in the system containing at least one of the entities accessed by M is added to T. Then, the candidate target classes in T are sorted in descending order according to the number of entities that M accesses from each of them. In the subsequent steps, each target class T_i is analyzed to verify its suitability to be the recommended class. In particular, T_i must satisfy three conditions to be considered in the set of candidate suggestions: (i) T_i is not the class M currently belongs to, (ii) M modifies at least one data structure in T_i, and (iii) moving M in T_i satisfies a set of behavior preserving preconditions (*e.g.,* the target class does not contain a method with the same signature as M) [100]. The set of classes in T satisfying all the conditions above are put in the suggested set. If the set of suggestions is non-empty, the approach suggests to move M to the first candidate target class following the order of the sorted set T. On the other hand, if the set of suggestions is empty, the classes in the sorted set T are analyzed again by applying milder constraints than before. In particular, if a class T_i is the class owning M, then no refactoring suggestion is performed and the algorithm stops. Otherwise, the approach checks if moving the method M into T_i satisfies the behavior preserving preconditions. If so, the approach suggests to move M into T_i. Thus, an instance of the *Feature Envy* code smell is identified.

Finally, as for *Misplaced Class* and *Promiscuous Package*, we re-implemented the approaches proposed by Atkinson and King [6] and by Girvan *et al.* [38], respectively. This choice was driven by the fact that, to the best of our knowledge, these are the only structural tools able to identify these code smell types. Other approaches able to recommend *Move Class* and *Extract Package* refactoring operations (see *e.g.,* [11], [16]) combine both conceptual and structural information, thus being not suitable in our context.

The technique selected for the detection of *Misplaced Class* instances [6] traverses the abstract syntax tree of

1. https://pmd.github.io
2. http://checkstyle.sourceforge.net
3. http://www.ptidej.net/research/designsmells/

TABLE 2
Characteristics of the Software Projects in Our Dataset

System	#Releases	#Commits	Classes	Methods	KLOCs
ArgoUML	16	19,961	777-1,415	6,618-10,450	147-249
Apache Ant	22	13,054	83-813	769-8,540	20-204
aTunes	31	6,276	141-655	1,175-5,109	20-106
Apache Cassandra	13	20,026	305-586	1,857-5,730	70-111
Eclipse Core	29	21,874	744-1,181	9,006-18,234	167-441
FreeMind	16	722	25-509	341-4,499	4-103
HSQLDB	17	5,545	54-444	876-8,808	26-260
Apache Hive	8	8,106	407-1,115	3,725-9,572	64-204
Apache Ivy	11	601	278-349	2,816-3,775	43-58
Apache Log4j	30	2,644	309-349	188-3,775	58-59
Apache Lucene	6	24,387	1,762-2,246	13,487-17,021	333-466
JEdit	29	24,340	228-520	1,073-5,411	39-166
JHotDraw	16	1,121	159-679	1,473-6,687	18-135
JVLT	15	623	164-221	1,358-1,714	18-29
Apache Karaf	5	5,384	247-470	1,371-2,678	30-56
Apache Nutch	7	2,126	183-259	1,131-1,937	33-51
Apache Pig	8	2,857	258-922	1,755-7,619	34-184
Apache Qpid	5	14,099	966-922	9,048-9,777	89-193
Apache Struts	7	4,297	619-1,002	4,059-7,506	69-152
Apache Xerces	16	5,471	162-736	1,790-7,342	62-201
Overall	**301**	**183,514**	**25-2,246**	**188-17,021**	**4-466**

a class C in order to determine, for each feature, the set T of classes referencing them. Then, the classes in T are sorted based on the package they belong to in order to extract the number of dependencies each package $P \in T$ has with the class C. If C has more dependencies with a different package with respect to the one it is actually in, an instance of *Misplaced Class* is detected. Our re-implementation relies on the publicly available Java Development Tools APIs[4].

The approach selected for the detection of *Promiscuous Package* instances [38] is based on a clustering algorithm that groups together classes of a package based on the dependencies among them. In the re-implementation, we exploited the X-Means algorithm [80], an extension of the traditional K-Means [59] where the parameter X (*i.e.,* the number of clusters the algorithm must form) is automatically configured using a heuristic based on the Bayesian Information Criterion [80]. If the algorithm finds more than one cluster, it means that the classes contained in the package under analysis contain unrelated responsibilities and, therefore, an instance of the *Promiscuous Package* code smell is detected.

3 Study I: The Evolution of Textual and Structural Code Smells

In this study, we mined several software repositories to empirically investigate how developers deal with textually and structurally detected code smells.

3.1 Empirical Study Definition and Design

The *goal* of the empirical study is to evaluate the impact of different sources of information on developers' notion of code smells. Our *conjecture* is that code smells characterized by an inconsistent vocabulary are easier to identify and/or easier to remove for developers when compared to code smells characterized by structural problems, such as a high number of dependencies or large size, since conceptual aspects of source code can provide direct insight that a developer can use to understand and work on code components affected by code smells. The *context* of the study consists of the five code smells presented in Section 2. We conducted our analyses on twenty open source software projects. Table 2 reports the characteristics of the analyzed systems[5], namely the number of public releases, and their size in terms of number of commits, classes, methods, and KLOC. Among the analyzed projects, twelve projects belong to the Apache ecosystem[6] hosted on GitHub, and eight projects belong to the Sourceforge repository[7]. Given the list of projects available in the two repositories, we randomly selected twenty systems among the most popular ones having at least 500 commits. These filters allowed to (i) identify popular systems in the two repositories, and (ii) discard systems having a short development history. As a result, we analyzed projects belonging to different ecosystems, having different size and scope.

Our investigation aims at answering the following research questions:

- **RQ$_1$:** *Are textually or structurally detected code smells more likely to be resolved?*
- **RQ$_2$:** *Do structural or textual code smells evolve differently with respect to different types of changes (Bug fixing, Enhancement, New feature, Refactoring)?*

To answer **RQ$_1$**, we first manually detected the releases (both major and minor ones) of the software projects in our dataset, for a total of 301 releases. Then, our `ChangeHistoryMiner`[8] tool analyzed each release R of a software project p_i to detect code components (*i.e.,* methods or classes) affected by one of the considered code smells.

To monitor the evolution of code smells, a simple truth value representing the presence or absence of a code smell instance is not enough because we might not evaluate how the severity of structurally and textually detected code smells varies (decreases/increases) over the releases of the projects in our dataset. Hence, once a code smell was detected we monitored its evolution in terms of *intensity*, *i.e.,* in terms of variation of the degree of severity of a code smell.

Computing the *intensity* is easy for TACO, since it outputs a value $\in [0; 1]$ indicating the probability that a code component is affected by a code smell. In the other cases, we followed the guidelines by Marinescu [64], who suggested to compute the severity index by considering how much the value of a chosen metric exceeds a given threshold. In particular, DECOR classifies a code component as smelly if and only if a set of conditions (rules) are satisfied, where each condition has the form $\texttt{if metric}_i \geq \texttt{threshold}_i$. Therefore, the higher the

4. http://www.eclipse.org/jdt/

5. The list of repositories is available in our on-line appendix [76]
6. http://www.apache.org/ verified April 2017
7. https://sourceforge.net
8. The tool is available in our online appendix [76].

TABLE 3
Overlap between TACO and the structural techniques (ST) employed in the study.

Code Smell	TACO∩ST		TACO\ST		ST\TACO	
	#	%	#	%	#	%
Long Method	364	60%	188	31%	53	9%
Feature Envy	101	46%	58	26%	62	28%
Blob	138	42%	138	42%	49	16%
Promiscuous Package	43	28%	78	51%	33	21%
Misplaced Class	12	21%	39	67%	8	12%
Overall	**658**	**48%**	**501**	**37%**	**205**	**15%**

distance between the actual code metric ($\texttt{metric}_i$) and the fixed threshold value ($\texttt{threshold}_i$), the higher the *intensity* of the flaw. Thus, we measured the intensity of classes detected as Blob by DECOR as follows: (i) we computed the differences between the actual values of software metrics (*e.g.*, LCOM5, number of methods, etc.) with respect to the corresponding thresholds reported in the rule card [67]; (ii) we normalized the obtained scores in $[0; 1]$, and (iii) we measured the final intensity as the mean of those normalized scores.

As *Long Method*s are detected by only looking at the LOC (lines of code), the intensity is measured as the normalized difference between the LOC in a method and its threshold in the rule card, which is 100.

JDeodorant marks a method m as *Feature Envy* if and only if it has more structural dependencies with another class C^* with respect to the number of dependencies m has with the original class C (and if all preconditions are preserved). Therefore, the *intensity* is given by the normalized difference of the number of dependencies with C^* (new class) and the number of dependencies with C (original class).

The same strategy can be applied to measure the *intensity* of *Misplaced Class* instances. Indeed, as the technique by Atkinson and King [6] identifies this code smell by looking at the difference between the dependencies a class C has toward a package P^* and the dependencies C has with the original package P, the *intensity* is given by the normalized difference between them.

Finally, we measured the *intensity* of *Promiscuous Package* code smell by applying a *min-max* normalization on the number of clusters of classes found by the approach for a package P. In this way, the higher the number of clusters detected the higher the proneness of the package to be promiscuous.

It is worth noting that since our goal is to investigate to what extent textual and structural code smells evolve differently, in this study we do not consider the code smell instances identified by both types of detector, *i.e.,* textual-based and structural-based detectors. Table 3 shows the data, aggregated by code smell type, about (i) the number of instances detected by both the detectors (column "TACO∩ST"), (ii) the number of instances detected by TACO and missed by the structural detectors (column "TACO\ST"), and (iii) the number of instances detected by the structural detectors and missed by TACO (column "ST\TACO"). Therefore, we excluded 658 (48%) code smell instances of the 1,364 detected. Thus, our analysis is carried out on 52% of the code smell instances detected by the tools.

TABLE 4
Tags assigned to commits involving code smells.

Tag	Description
Bug fixing	The commit aimed at fixing a bug [102].
Enhancement	The commit aimed at implementing an enhancement in the system [102].
New feature	The commit aimed at implementing a new feature in the system [102].
Refactoring	The commit aimed at performing refactoring operations [102].

From the analysis of the structural and textual smelliness, we obtained two distributions for each type of code smells: one related to the variation of intensity for textual code smells (Δ_{text}) over the different releases, and the other one regarding the variation of intensity for structural code smells (Δ_{struct}) over the same releases. Negative values for Δ_{text} (or Δ_{struct}) indicate that the intensity of textual (or structural) code smells decreased over time, while positive values indicate increase of the intensity. To verify whether the differences (if any) between Δ_{text} and Δ_{struct} are statistically significant, we used the non-parametric Wilcoxon Rank Sum test [23] with ρ-value = 0.05. We also estimated the magnitude of the observed differences using Cliff's Delta (or d), a non-parametric effect size measure [39] for ordinal data. We followed the guidelines in [39] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

As for $\mathbf{RQ}_2$, we are interested in understanding whether particular types of changes made by developers have a higher impact on the increase/decrease of the intensity of code smells. To this aim, we conducted a *fine-grained* analysis, investigating all the commits available in the repositories of the involved projects (overall, we mined 183,514 commits) in order to understand *what type of action the developer was doing when modifying smelly classes*. Given a repository r_i, `ChangeHistoryMiner` mines the entire change history of r_i, and for each commit involving a code smell runs the mixed technique proposed by Tufano *et al.* [102] in order to detect the types of changes shown in Table 4, *i.e., Bug Fixing*, *Enhancement*, *New Feature*, and *Refactoring*. To this aim, we downloaded the issues for all 20 software projects from their BUGZILLA or JIRA issue trackers. Then, we checked if a commit involving a textual or structural code smell was actually related to any collected issues. To link issues to commits, the approach by Tufano *et al.* complements two distinct approaches: the first one is based on regular expressions [29], which match the issue ID in the commit note, while the second one is *ReLink*, the approach proposed by Wu *et al.* [108], which considers several constraints, *i.e.,* (i) a match exists between the committer and the contributor who created the issue in the issue tracking system, (ii) the time interval between the commit and the last comment posted by the same

contributor in the issue tracker is less than seven days, and (iii) the cosine similarity between the commit note and the last comment referred above, computed using the Vector Space Model (VSM) [7], is greater than 0.7. When it was possible to find a link between a commit and an issue, and the issue had a type included in the catalogue of tags shown in Table 4, then the commit was automatically classified. In the other cases, we assigned the tags using a semi-automatic process. Specifically, we used a keyword-based approach to detect a commit's goal similar to the one presented by Fischer *et al.* [29], and then we manually validated the tags assigned by analyzing (i) the commit message and (ii) the unix diff between the commit under analysis and its predecessor. Overall, we tagged 27,769 commits modifying instances of textually and structurally detected code smells. For 18,276 of them, we found the tag automatically, while the remaining 9,493 were manually assigned.

Out of the total 9,493 commits, we needed to fix the initial classification of 3,512 commits (i.e., 37% of commits were misclassified) made by the approach by Fischer *et al* [29]. Of these, 1,545 were related to enhancement operations wrongly classified as new feature implementations (44% of the misclassified instances), 1,229 to non-documented refactoring operations (i.e., 35% of the misclassified instances), 562 to bug fixes wrongly classified as enhancements (i.e., 16% of the misclassified instances), and finally 176 to the implementation of new features wrongly classified as enhancements (5% of the misclassified instances). We adopted a formal procedure to assign a tag to these commits. The first two authors (the inspectors) independently analyzed each of the 9,493 commits with the aim of identifying its goal. Once this first stage was completed, the inspectors compared their classifications: the commits classified in the same manner by both the inspectors were not discussed, while in the other cases the inspectors opened a discussion to resolve the disagreement and reach consensus. The overall agreement[9] between the two inspectors was 84% (*i.e.,* 7,982 commit goals classified in the same manner over the total 9,483). This process took approximately 320 man/hour distributed in one month of work.

Once we obtained the tagged commits, we investigated how the different types of code changes (independent variables) impacted the variation of intensity of textual and structural code smell (dependent variable). In particular, for each object project and for each kind of code smell we applied logistic regression models [42] using the following equation:

$$\pi(\text{BF}, \text{E}, \text{NF}, \text{R}) = \frac{e^{C_0 + C_1 \cdot \text{BF} + C_2 \cdot \text{E} + C_3 \cdot \text{NF} + C_4 \cdot \text{R}}}{1 + e^{C_0 + C_1 \cdot \text{BF} + C_2 \cdot \text{E} + C_3 \cdot \text{NF} + C_4 \cdot \text{R}}} \quad (6)$$

where the independent variables are the number of *Bug Fixing* (BF), *Enhancement* (E), *New Feature* (NF) and *Refactoring* (R) operations applied by developers during the time period between two subsequent releases; the (dichotomous) dependent variable is whether the intensity increases/decreases between two subsequent versions; and C_i are the coefficients of the logistic regression model. Then, for each model we analyzed (i) whether each independent variable was significantly correlated with the dependent variable as estimated by the Spearman rank correlation coefficient (we considered a significance level of $\alpha = 5\%$), and (ii) we quantified such a correlation using the Odds Ratio (OR) [91] which, for a logistic regression model, is given by e^{C_i}. Odd ratios indicate the increase in likelihood of a code smell intensity increase/decrease as a consequence of a one-unit increase of the independent variable, e.g., number of bug fixing operations (BF). For example, if we found that *Refactoring* has an OR of 1.10 with textual Blobs, this means that each one-unit increase of the *Refactoring* made on a textual Blob mirrors a 10% higher chance for the Blob of being involved in a decrease of its intensity.

Overall, the data extraction to answer $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ took five weeks on 4 Linux machines having dual-core 3.4 GHz CPU (2 cores) and 4 Gb of RAM.

3.2 Analysis of the Results

Table 5 reports the mean and the standard deviation scores of the variation of intensity for textual (Δ_{text}) and structural (Δ_{struct}) code smells, collected for *Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, and *Promiscuous Package* instances. The results clearly indicate that textual smells are treated differently than structural ones: in most cases the intensity of textual code smells tends to decrease over time, *i.e.,* the Δ_{text} values are negative; vice versa, the intensity of structural code smells tends to increase over time, as indicated by the positive Δ_{struct} scores. For example, blobs in `JVLT` detected by structural tools have an average Δ_{struct}=0.86, *i.e.,* their structural metrics (*e.g.,* LCOM5) increase (worsen) by 86% on average at each new release. Instead, for the same project, the intensity of textual Blobs decreases (improves) 21% on average. An interesting example can be found in `Apache Ant`, when analyzing the evolution of the class `Property` of the `org.apache.tools.ant.taskdefs` package. The class is responsible for managing the Ant build properties. In the first versions of the project—from version 1.2 to version 1.5.4—the class was affected by a *Blob* code smell (it had a level of textual intensity equal to 0.83) since it implemented seven different ways to set such properties. During its evolution, the intensity has been reduced by developers through the application of different types of operations, such as code overriding (version 1.6) and refactorings (version 1.6.1), leading to a decrease of the complexity of the class, and consequently to the removal of the *Blob* code smell. Currently, the class is responsible to set the execution environment of the build process by getting the desired properties using a string. A similar discussion can be made for the other studied code smells. Code

9. Measured using the Jaccard similarity coefficient [45], *i.e.,* the number of commit tags classified in the same way by the inspectors over the number of all the commits.

TABLE 5
Mean and Standard Deviations of Δ_{text} and Δ_{struct} of our dataset. Decreasing variations are reported in **bold** face. TS = Textual Smells; SS = Structural Smells.

Project	Blob				Feature Envy				Long Method				Misplaced Class				Promiscuous Package			
	TS		SS		TS		SS		TS		SS		TS		SS		TS		SS	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
ArgoUML	**-0.08**	±0.23	0.13	±0.33	**-0.03**	±0.20	0.10	±0.26	**-0.03**	±0.17	0.11	±0.22	-	-	-	-	**-0.12**	±0.25	0.38	±0.24
Apache Ant	**-0.11**	±0.27	0.18	±0.47	**-0.05**	±0.22	0.11	±0.43	**-0.04**	±0.20	0.09	±0.38	**-0.05**	±0.12	0.04	±0.39	**-0.07**	±0.13	0.37	±0.24
aTunes	**-0.08**	±0.24	0.11	±0.25	**-0.01**	±0.27	0.10	±0.41	**-0.06**	±0.28	0.08	±0.42	-	-	-	-	**-0.01**	±0.11	0.49	±0.33
Apache Cassandra	**-0.08**	±0.25	0.18	±0.35	**-0.06**	±0.26	0.21	±0.19	**-0.06**	±0.26	0.23	±0.24	-	-	-	-	**-0.15**	±0.12	0.33	±0.25
Eclipse Core	**-0.08**	±0.23	0.12	±0.31	**-0.03**	±0.21	0.18	±0.37	**-0.03**	±0.23	0.15	±0.35	**-0.04**	±0.14	0.05	±0.14	**-0.08**	±0.16	0.44	±0.31
FreeMind	**-0.07**	±0.21	0.23	±0.41	0.01	±0.01	0.15	±0.36	0.01	±0.06	0.16	±0.32	-	-	-	-	**-0.05**	±0.07	0.14	±0.09
HSQLDB	**-0.07**	±0.21	0.15	±0.27	**-0.03**	±0.13	0.08	±0.14	**-0.04**	±0.12	0.14	±0.11	**-0.03**	±0.30	0.11	±0.15	**-0.09**	±0.22	0.38	±0.14
Apache Hive	**-0.04**	±0.16	0.14	±0.45	0.04	±0.14	0.12	±0.38	**-0.02**	±0.17	0.13	±0.43	**-0.01**	±0.10	0.03	±0.11	**-0.03**	±0.27	0.24	±0.36
Apache Ivy	**-0.10**	±0.23	0.19	±0.32	**-0.03**	±0.14	0.10	±0.37	**-0.01**	±0.18	0.10	±0.36	**-0.04**	±0.25	0.11	±0.09	**-0.02**	±0.14	0.27	±0.42
Apache Log4j	**-0.05**	±0.16	0.18	±0.13	0.01	±0.08	0.25	±0.12	**-0.01**	±0.13	0.21	±0.11	-	-	-	-	0.02	±0.11	0.32	±0.22
Apache Lucene	**-0.11**	±0.24	0.11	±0.32	**-0.02**	±0.07	0.12	±0.25	**-0.01**	±0.07	0.11	±0.23	0.01	±0.05	0.14	±0.06	**-0.03**	±0.07	0.38	±0.14
JEdit	**-0.11**	±0.25	0.12	±0.15	**-0.04**	±0.23	0.23	±0.35	**-0.02**	±0.25	0.23	±0.35	-	-	-	-	**-0.16**	±0.05	0.26	±0.11
JHotDraw	**-0.09**	±0.23	0.10	±0.27	**-0.04**	±0.23	0.13	±0.05	**-0.04**	±0.22	0.12	±0.19	**-0.07**	±0.10	0.01	±0.04	**-0.08**	±0.12	0.11	±0.24
JVLT	**-0.21**	±0.27	0.86	±0.03	**-0.10**	±0.41	0.68	±0.07	**-0.10**	±0.41	0.76	±0.20	-	-	-	-	**-0.04**	±0.17	0.08	±0.04
Apache Karaf	**-0.06**	±0.16	0.29	±0.12	**-0.02**	±0.15	0.16	±0.35	**-0.02**	±0.16	0.16	±0.38	**-0.07**	±0.34	0.05	±0.12	**-0.15**	±0.19	0.21	±0.34
Apache Nutch	**-0.09**	±0.24	0.12	±0.05	**-0.01**	±0.02	0.05	±0.32	**-0.02**	±0.08	0.05	±0.31	-	-	-	-	0.01	±0.27	0.23	±0.09
Apache Pig	**-0.02**	±0.30	0.09	±0.36	**-0.06**	±0.17	0.08	±0.31	**-0.01**	±0.12	0.11	±0.24	**-0.17**	±0.07	0.02	±0.13	**-0.24**	±0.25	0.12	±0.04
Apache Qpid	**-0.09**	±0.23	0.06	±0.41	**-0.08**	±0.29	0.11	±0.26	**-0.01**	±0.15	0.10	±0.22	**-0.23**	±0.17	0.25	±0.11	0.02	±0.13	0.04	±0.11
Apache Struts	**-0.04**	±0.14	0.10	±0.18	**-0.01**	±0.03	0.11	±0.27	**-0.02**	±0.11	0.10	±0.25	-	-	-	-	**-0.18**	±0.33	0.25	±0.12
Apache Xerces	**-0.06**	±0.19	0.16	±0.31	**-0.02**	±0.12	0.09	±0.31	**-0.03**	±0.12	0.10	±0.28	**-0.08**	±0.16	0.28	±0.29	**-0.03**	±0.20	0.29	±0.21
Overall	**-0.09**	**±0.24**	**0.14**	**±0.35**	**-0.03**	**±0.20**	**0.15**	**±0.35**	**-0.04**	**±0.19**	**0.14**	**±0.34**	**-0.06**	**±0.18**	**0.17**	**±0.22**	**-0.15**	**±0.23**	**0.37**	**±0.26**

TABLE 6
Comparison between Δ_{text} and Δ_{struct} for Blob, Feature Envy and Long Method. We use S, M, and L to indicate small, medium and large Cliff's d effect sizes respectively. Significant p-values are reported in bold face

	Textual vs. Structural														
	Blob			**Feature Envy**			**Long Method**			**Misplaced Class**			**Promiscuous Package**		
Project	**p-value**	d	**M**	**p-value**	d	**M**	**p-value**	d	**M**	**p-value**	d	**M**	**p-value**	d	**M**
ArgoUML	<**0.01**	-0.76	L	<**0.01**	-0.85	L	<**0.01**	-0.88	L	-	-	-	<**0.01**	-0.51	L
Apache Ant	<**0.01**	-0.66	L	<**0.01**	-0.67	L	<**0.01**	-0.71	L	<**0.01**	-0.79	L	<**0.01**	-0.62	L
aTunes	<**0.01**	-0.84	L	<**0.01**	-0.52	L	<**0.01**	-0.59	L	-	-	-	<**0.01**	-0.57	L
Apache Cassandra	<**0.01**	-0.76	L	<**0.01**	-0.89	L	<**0.01**	-0.91	L	-	-	-	<**0.01**	-0.60	L
Eclipse Core	<**0.01**	-0.83	L	<**0.01**	-0.77	L	<**0.01**	-0.74	L	<**0.01**	-0.76	L	<**0.01**	-0.78	L
FreeMind	<**0.01**	-0.78	L	<**0.01**	-0.83	L	<**0.01**	-0.79	L	-	-	-	<**0.01**	-0.72	L
HSQLDB	<**0.01**	-0.83	L	<**0.01**	-0.72	L	<**0.01**	-0.84	L	<**0.01**	-0.92	L	<**0.01**	-0.84	L
Apache Hive	<**0.01**	-0.65	L	<**0.01**	-0.70	L	<**0.01**	-0.68	L	<**0.01**	-0.89	L	<**0.01**	-0.98	L
Apache Ivy	<**0.01**	-0.89	L	<**0.01**	-0.78	L	<**0.01**	-0.63	L	<**0.01**	-0.73	L	<**0.01**	-0.66	L
Apache Log4j	<**0.01**	-0.76	L	<**0.01**	-0.82	L	<**0.01**	-0.78	L	-	-	-	<**0.01**	-0.79	L
Apache Lucene	<**0.01**	-0.83	L	<**0.01**	-0.89	L	<**0.01**	-0.91	L	<**0.01**	-0.68	L	<**0.01**	-0.71	L
JEdit	<**0.01**	-0.95	L	<**0.01**	-0.79	L	<**0.01**	-0.76	L	-	-	-	<**0.01**	-0.63	L
JHotDraw	<**0.01**	-0.80	L	<**0.01**	-0.92	L	<**0.01**	-0.89	L	<**0.01**	-0.72	L	<**0.01**	-0.61	L
JVLT	<**0.01**	-1.00	L	<**0.01**	-1.00	L	<**0.01**	-0.98	L	-	-	-	<**0.01**	-0.70	L
Apache Karaf	<**0.01**	-1.00	L	<**0.01**	-0.87	L	<**0.01**	-0.79	L	<**0.01**	-0.72	L	<**0.01**	-0.68	L
Apache Nutch	<**0.01**	-1.00	L	<**0.01**	-0.81	L	<**0.01**	-0.81	L	-	-	-	<**0.01**	-0.97	L
Apache Pig	<**0.01**	-0.78	L	<**0.01**	-0.90	L	<**0.01**	-0.82	L	<**0.01**	-0.64	L	<**0.01**	-0.71	L
Apache Qpid	<**0.01**	-0.70	L	<**0.01**	-0.84	L	<**0.01**	-0.85	L	<**0.01**	-0.59	L	<**0.01**	-0.65	L
Apache Struts	<**0.01**	-0.94	L	<**0.01**	-0.88	L	<**0.01**	-0.90	L	-	-	-	<**0.01**	-0.79	L
Apache Xerces	<**0.01**	-0.85	L	<**0.01**	-0.81	L	<**0.01**	-0.86	L	<**0.01**	-0.95	L	<**0.01**	-0.82	L
Overall	**<0.01**	**-0.78**	**L**	**<0.01**	**-0.78**	**L**	**<0.01**	**-0.77**	**L**	**<0.01**	**-0.74**	**L**	**<0.01**	**-0.69**	**L**

elements affected by textual code smells are seemingly more carefully managed by developers. On the other hand, code smells detected by DECOR tend to have a different evolution. For instance, the evolution of the method `org.hsqldb.JDBCBench.createDatabase` of the `HSQLDB` project is quite representative. This method should manage the functionality for creating a new database, but during evolution its size strongly increased as more sub-functionalities have been added, resulting in a *Long Method*. Interesting is the comment left by a developer in the source code of the method at version 1.7.3 of the project: *"Totally incomprehensible! One day or another, we should fix this method... I don't know how!"*. This comment gives strength to our initial conjecture, namely that textual code smells are easier to identify and refactor with respect to structural code smells.

Our preliminary findings seem to confirm the observations made by Vidal *et al.* [104] on the limited support provided by structural-based code smell detectors due to the fact that they tend to highlight a large amount of design problems that developers are not able to deal with. Also, the statistical tests confirmed our results (see Table 6). Specifically, for all the studied code smells the difference between the two distributions Δ_{text} and Δ_{struct} is always statistically significant (ρ-values<0.01), *i.e.*, the variations of intensity for structural and textual code smells are statistically different. It is worth noting that the magnitude of Cliff's d measure is always *large*.

Having observed that textual and structural code

TABLE 7
Percentage of Different Types of Changes applied over Textual and Structural Code Smells. NF = New Feature; BF = Bug Fixing; R = Refactoring; E = Enhancement

Code Smell	Textual Smells				Structural Smells			
	NF	BF	R	E	NF	BF	R	E
Blob	10	32	14	44	10	32	10	48
Feature Envy	12	28	14	46	10	34	8	48
Long Method	8	34	13	45	8	38	6	48
Misplaced Class	15	21	14	50	17	25	5	53
Promiscuous Package	11	33	17	39	9	37	7	47

smells are treated differently, we turn our attention to investigating which types of operations are performed by developers on the two sets of code smells and to what extent such operations have an effect on the increase/decrease of their intensity. As for operations having the effect of increasing the intensity of textually and structurally detected code smells, we did not find a clear relationship between specific changes and the increase of intensity. When considering textually detected code smells, we found that for 35% of changes implementing new features the intensity tends to increase; 57% of times an increase is due to enhancement or bug fixing activities. Also for structurally detected code smells, we observed that most of the times (91%) changes aimed at implementing new features, enhancing or fixing bugs in the project tend to increase the code smell intensity. Moreover, for both textual and structural code smells, we found that in a small percentage (8% for textual code smells, 9% for structural code smells) refactoring operations increase the level of intensity. Even though this result can appear unexpected, it confirms previous findings that sometimes refactorings can also be the cause of introducing code smells [102]. A complete report of this analysis is available in our online appendix [76].

Regarding the operations reducing the level of intensity, Table 7 reports the percentage of the different types of changes, *i.e., New Feature* (NF), *Bug Fixing* (BF), *Refactoring* (R), and *Enhancement* (E), applied to the set of textual and structural code smells in our dataset. Considering the results achieved in previous work [10], [22], [52], [81], [94], the most unexpected result is the one related to the percentage of refactoring operations. In fact, even if the number of refactoring operations performed on code smells remains quite low—confirming that code smells are poorly refactored—we observed that textual code smells are generally more prone to be subject to these operations (*Blob*=+4%, *Feature Envy*=+6%, *Long Method*=+7%, *Misplaced Class*=+9%, *Promiscuous Package*=+10%). In the following, we provide detailed results on the types of changes positively influencing the intensity for each code smell in our study.

Blob. Table 8 shows the Odds Ratios of the different types of changes applied to textual and structural code smells, obtained when building a logistic regression model for data concerning the decrease of code smell intensity. In the following, we will mainly focus our discussion on statistically significant values. First of all, we can notice that changes tagged as *Refactoring* often have higher chance to decrease the intensity of textual *Blob*s (the ORs are higher than 1 in 85% of significant ORs). Thus, refactoring operations applied to *Blob* instances characterized by textual problems have a higher chance of being effective in the reduction of the code smell intensity. Another unexpected result regards what we found for the category *Enhancement*: indeed, also in this case such changes have more chance to be effective in the reduction of the complexity of a textual *Blob*. A possible reason behind this finding concerns the better ability of developers to enhance code components affected by textual code smells, as they have less difficulties understanding the problems affecting the source code. This result somehow confirms previous findings on the usefulness of textual information for program comprehension [57]. As for structural *Blob* instances, the results show that *Bug Fixing* operations have a higher chance to reduce the code smell intensity. This means that code components having low quality as measured by software metrics are mainly touched by developers only when a bug fix is required. Looking at these findings, *we can conclude that textual Blob instances are on the one hand more prone to be refactored and on the other hand more likely to be resolved by such operations, while the complexity of structural Blob instances is mainly reduced through bug fixing operations*. This claim is also supported by the analysis of the number of textual and structural *Blob* instances actually removed in our dataset (see Table 10). Indeed, we observed that 27% of textual code smells in our dataset have been removed over time, and in 12% of the cases they have been removed using refactoring operations. At the same time, we can see that the percentage of structural *Blob* instances removed over time is much lower (16%) and the percentage of refactorings is 7% lower with respect to textual blobs.

Feature Envy. The ORs achieved when applying the logistic regression model relating the types of changes to the decrease of code smell intensity for *Feature Envy* are reported in Table 8. In most cases changes classified as *New Feature*, *Bug Fixing*, and *Refactoring* do not reduce the intensity of either textual and structural *Feature Envy* instances. Instead, the enhancement operations made on textually detected *Feature Envy* code smells have, overall, 14% more chance of reducing the code smell intensity. Looking at the results of structurally detected *Feature Envy* code smells, none of the analyzed changes seem to lead to an intensity reduction. Moreover, it seems that textually detected *Feature Envy* instances differ from structurally detected ones, since other than being removed more frequently (see Table 10), the refactoring operations are slightly more effective (+2%) in the removal of the code smell. Since this code smell arises when a method has more in common with another class with respect to the one it is actually in, such a difference can

TABLE 8
OR of different types of changes applied to *Blob*, *Feature Envy*, and *Long Method* instances when building logistic model. Statistically significant ORs are reported in **bold** face.

Project	Blob								Feature Envy								Long Method							
	Textual Smell				Structural Smell				Textual Smell				Structural Smell				Textual Smell				Structural Smell			
	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E
ArgoUML	0.78	0.85	1.11	**1.50**	0.81	**1.00**	0.87	0.89	0.81	0.97	0.89	**1.10**	0.80	**0.83**	**0.93**	0.95	0.87	**1.05**	0.99	1.01	0.73	0.89	**0.89**	0.74
Apache Ant	0.99	**1.01**	1.00	1.01	0.97	1.01	**0.99**	**1.00**	0.88	**0.93**	0.91	**1.02**	0.87	**0.82**	0.81	**0.92**	0.86	1.01	**1.11**	**1.02**	0.69	**0.98**	0.84	0.89
aTunes	1.01	0.98	**1.01**	**1.14**	0.98	**1.01**	0.95	0.87	**0.98**	0.89	**0.92**	1.01	0.87	**0.77**	0.91	0.96	0.89	**1.02**	**1.10**	1.00	0.99	0.97	**1.01**	0.88
Apache Cassandra	0.99	1.00	1.00	1.00	0.97	**1.01**	0.99	1.00	0.83	**0.87**	0.92	**1.03**	0.87	**0.78**	**0.76**	0.93	0.93	**0.98**	**1.01**	1.01	0.89	**0.92**	**0.95**	0.91
Eclipse Core	**0.88**	1.01	**1.10**	**1.34**	0.99	**0.97**	1.00	**1.02**	0.81	0.84	**0.97**	**1.04**	0.83	0.92	0.83	**0.97**	0.98	**1.10**	**1.15**	0.97	0.68	**1.02**	0.98	**1.01**
FreeMind	0.91	**1.02**	**0.89**	**1.22**	0.92	**0.98**	0.78	0.88	0.72	0.81	0.94	**1.02**	0.71	**0.82**	**0.91**	0.99	**0.86**	**1.02**	0.99	0.87	0.83	**0.97**	0.94	0.91
HSQLDB	1.01	0.97	**1.06**	**1.18**	1.01	0.93	**1.00**	0.99	0.75	**0.81**	0.98	1.10	**0.72**	**0.89**	0.93	0.84	0.92	0.97	**1.02**	**0.88**	0.91	0.97	**0.99**	**0.78**
Apache Hive	0.99	1.00	**1.01**	1.00	0.97	**1.01**	0.99	1.00	0.87	**0.93**	**0.84**	1.11	0.73	**0.74**	0.81	0.96	**0.89**	0.97	**1.00**	0.87	0.79	**0.81**	0.74	**0.89**
Apache Ivy	0.99	1.00	**1.02**	1.00	0.96	**0.99**	**0.99**	1.00	0.74	0.73	0.92	**1.08**	0.86	0.74	**0.72**	0.83	0.92	0.98	**1.01**	0.98	0.91	0.86	0.81	**0.92**
Apache Log4j	0.86	**1.02**	**1.06**	**1.19**	0.94	**0.98**	**1.01**	**1.01**	**0.83**	**0.92**	0.97	1.05	0.78	0.70	**0.91**	0.96	0.91	**1.02**	**1.10**	0.82	**0.72**	0.76	**0.93**	0.83
Apache Lucene	0.99	1.00	**1.20**	1.00	0.97	**1.01**	0.99	1.00	0.81	0.84	0.92	**1.04**	0.79	**0.84**	0.81	0.97	0.97	**1.14**	**1.36**	**1.02**	0.89	0.93	**1.05**	**0.82**
JEdit	0.88	1.04	1.18	**1.24**	0.0	0.65	0.18	0.66	**0.83**	0.98	0.82	1.03	0.85	**0.78**	0.97	0.98	0.85	**1.01**	0.99	0.84	0.92	**0.78**	0.92	0.89
JHotDraw	0.67	0.88	**1.02**	**1.09**	0.54	0.41	**0.88**	0.64	0.84	**0.87**	0.93	**0.99**	0.84	0.78	0.84	**0.91**	0.92	**1.21**	0.97	0.88	**0.93**	**0.99**	0.96	0.88
JVLT	0.51	0.75	**1.01**	0.97	0.87	**0.99**	**1.01**	0.77	0.95	0.83	0.99	**1.06**	0.91	0.84	**0.93**	0.97	0.88	1.02	**1.02**	0.87	**1.00**	0.99	0.99	0.97
Apache Karaf	0.99	1.00	1.00	**1.01**	0.96	1.00	**0.99**	1.00	0.86	**0.88**	0.98	**1.11**	0.87	**0.94**	0.82	0.98	0.91	1.01	0.97	**0.89**	0.96	**1.01**	0.89	**0.79**
Apache Nutch	0.99	1.00	**1.02**	1.00	**0.95**	0.99	0.93	1.00	0.81	0.78	**0.98**	1.08	0.89	**0.79**	**0.83**	0.93	0.86	0.97	1.03	**1.12**	**0.78**	**0.85**	0.96	0.98
Apache Pig	0.99	1.01	**1.12**	**0.97**	0.96	**1.01**	0.99	1.00	0.88	0.87	0.92	**1.07**	0.71	**0.76**	**0.77**	0.86	0.93	0.99	**1.02**	**1.12**	**0.87**	0.79	**0.96**	1.02
Apache Qpid	0.99	**1.10**	**1.04**	1.00	0.97	0.99	**1.00**	**1.01**	0.87	0.82	0.88	**1.12**	0.72	**0.86**	0.84	0.91	0.82	1.00	1.01	**1.22**	0.87	**1.02**	0.98	0.92
Apache Struts	0.87	**1.05**	**1.15**	**1.27**	0.93	0.97	0.91	**0.89**	0.87	0.97	**1.02**	**1.17**	0.73	0.83	**0.98**	**0.98**	0.76	**1.01**	1.06	1.11	0.87	0.99	**0.98**	**1.02**
Apache Xerces	0.92	1.01	**1.52**	**1.23**	0.95	**0.97**	0.91	**0.99**	0.78	0.92	0.97	**1.21**	0.86	0.83	0.92	**1.02**	1.01	**1.09**	1.05	**1.12**	**0.98**	**1.01**	0.99	1.03
Overall	**0.99**	**1.02**	**1.23**	**1.20**	**0.97**	**1.01**	**0.98**	**0.99**	**0.81**	**0.85**	**0.91**	**1.14**	**0.79**	**0.86**	**0.91**	**0.93**	**0.96**	**1.05**	**1.18**	**1.02**	**0.88**	**0.95**	**0.97**	**0.97**

TABLE 9
OR of different types of changes applied to *Misplaced Class* and *Promiscuous Package* instances when building logistic model. Statistically significant ORs are reported in **bold** face.

Project	Misplaced Class								Promiscuous Package							
	Textual Smell				Structural Smell				Textual Smell				Structural Smell			
	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E
ArgoUML	-	-	-	-	-	-	-	-	1.02	0.93	**1.04**	0.99	0.97	0.98	0.88	1.01
Apache Ant	0.81	**1.04**	**1.08**	0.91	0.82	1.01	0.79	**1.01**	0.96	0.80	**1.02**	0.71	**1.02**	0.82	0.94	**1.01**
aTunes	-	-	-	-	-	-	-	-	0.82	**1.01**	0.77	0.92	**1.01**	0.67	0.92	0.81
Apache Cassandra	-	-	-	-	-	-	-	-	0.72	1.04	**1.11**	0.77	0.80	**1.01**	0.99	0.60
Eclipse Core	0.97	1.01	**1.18**	0.83	0.98	0.88	0.91	1.01	1.01	0.92	**1.12**	0.84	0.95	**1.08**	0.81	0.72
FreeMind	-	-	-	-	-	-	-	-	0.84	1.02	0.97	0.99	0.98	**1.01**	0.85	0.68
HSQLDB	**0.86**	0.79	**1.17**	0.80	0.71	0.66	0.91	**1.02**	0.51	**1.01**	**1.02**	0.81	0.98	**1.05**	0.95	0.83
Apache Hive	0.91	0.87	**1.18**	0.98	0.79	0.71	0.92	**1.02**	0.89	0.99	0.80	0.77	0.92	0.82	0.84	**1.02**
Apache Ivy	0.86	0.78	0.99	0.92	0.78	0.92	0.87	**1.06**	0.88	0.61	**1.11**	0.87	0.91	0.82	0.65	**1.03**
Apache Log4j	-	-	-	-	-	-	-	-	0.72	0.69	**1.20**	0.92	0.55	**1.06**	0.91	0.83
Apache Lucene	0.81	1.04	**1.21**	0.72	0.52	0.64	0.89	**1.01**	**1.03**	0.98	**1.02**	0.78	0.56	0.61	0.54	**0.89**
JEdit	-	-	-	-	-	-	-	-	0.88	**0.99**	0.92	0.88	0.79	0.76	0.66	**0.93**
JHotDraw	0.82	1.01	**1.22**	0.91	0.88	0.87	0.72	**0.84**	0.72	0.77	1.05	1.08	**0.79**	0.82	0.99	0.60
JVLT	0.51	-	-	-	-	-	-	-	0.95	0.73	**1.09**	0.74	0.69	0.59	0.68	0.99
Apache Karaf	0.81	**1.01**	0.90	0.82	0.81	0.57	0.82	**0.95**	0.69	0.72	**1.01**	0.92	0.74	0.92	0.81	**1.04**
Apache Nutch	-	-	-	-	-	-	-	-	0.77	**1.02**	**1.08**	0.98	0.96	0.88	**0.92**	0.81
Apache Pig	1.01	0.81	**1.27**	0.89	0.52	0.76	0.99	**0.92**	0.78	0.91	**1.21**	0.72	0.74	0.72	0.89	0.71
Apache Qpid	0.83	0.88	**1.11**	0.98	1.01	**1.05**	0.66	0.82	0.60	0.72	**1.20**	0.82	0.72	0.61	0.86	0.87
Apache Struts	-	-	-	-	-	-	-	-	0.81	**1.01**	**1.06**	0.62	0.81	0.86	**0.99**	**1.02**
Apache Xerces	0.92	0.99	**1.09**	**1.02**	0.82	**1.01**	0.89	**1.05**	0.99	**1.05**	**1.14**	0.74	0.88	**1.04**	0.89	0.71
Overall	**0.89**	**0.99**	**1.15**	**0.92**	**0.76**	**0.91**	**0.94**	**1.02**	**0.86**	**1.01**	**1.10**	**0.84**	**0.93**	**0.98**	**0.85**	**0.98**

be explained considering the way developers perceive different types of coupling. Indeed, as shown by Bavota *et al.* [15], semantic coupling better reflects the mental model of developers. So, it seems that developers are able to better perceive the symptoms of a textual *Feature Envy*, by providing solutions to limit it, or in some cases provide accurate solutions to remove it.

Long Method. Table 8 reports the results achieved when comparing textually and structurally detected *Long Method* instances. Regarding long methods textually detected, there are several causes which relate with the decrease of their intensity. Overall, we can see that refactoring operations have 18% more chance of reducing the complexity of the code smell, while *Bug Fixing* and *Enhancement* operations have 5% and 2% more chance, respectively. These findings highlight how the decrease of the intensity of long methods textually detected depends on the activity of the developers, confirming previous findings by Silva *et al.* [94]. When analyzing the results for structurally detected long methods, we observed that there are no specific types of changes that strongly influence the decrease of code smell intensity. Thus, also in this case we can claim that textual long methods are easier to be detected and removed from the source code. Table 10 highlights these differences, since

textually detected *Long Method* instances are removed in 35% of cases over time (18% of removals are due to refactoring), while structurally detected long methods are removed in 20% of the cases (8% of cases due to refactoring activities).

Misplaced Class. When evaluating the differences between textually and structurally detected instances of this code smell (Table 9), we can outline a clear trend in the results. Indeed, textual *Misplaced Class* instances undergo a considerable reduction of their intensity when refactoring is applied (changes of this type have 15% more chance of reducing the intensity of the code smell). The claim is also supported by the percentage of code smell instances removed by developers shown in Table 10, where we can observe that the code smell is removed in 18% of the cases over time and, more importantly, 11% of removals are due to refactoring. As previously explained for the *Feature Envy* code smell, the reasons behind this strong result can be found in the developers' perception of software coupling [15]. In fact, the code smell arises when a class has responsibilities closer to the ones of another package with respect to the one it is actually in. Therefore, if developers better comprehend the semantic relationships between classes it is reasonable to assume that they are more inclined to move classes to better locations. Concerning instances of the code smell found by the structural approach, the situation is different because of the limited values of the ORs achieved by the considered types of changes. The higher value is the one found by considering changes of the category *Enhancement* (+2% more chance of reducing the intensity). This means that developers actually reduce the intensity of structural misplaced classes only in cases where an enhancement is needed, rather than limiting the intensity through appropriate refactoring operations. It is worth noting that the percentage of instances of this code smell removed over time is lower than the one of textual code smells (-11% of removals) and that refactoring is the cause of removal only in 3% of the cases.

Promiscuous Package. The discussion of the results for this code smell type is quite similar to previous ones. Indeed, from Table 9 we can observe that in most cases refactorings are limiting the intensity of textually detected instances (+10% chance to reduce the severity), while for structural code smells there are no specific types of changes that influence the decrease of code smell intensity. This result seems to highlight the higher ability of developers to deal with promiscuous packages characterized by scattered concepts rather than by structural symptoms. Therefore, also in this case we can claim that developers are more able to identify textual code smells than structural ones and provide adequate solutions to reduce the intensity of the code smell. Notably, such instances are removed in 26% of the cases over the release history of the projects analyzed (+13% with respect to structural instances), and 11% of the times the reason of the removal is refactoring (+7% with respect to structural promiscuous packages).

TABLE 10
Percentage of Removed Textual (TS) and Structural Code (SS) Smell instances

Code Smell	% TS Removed (% due to refact.)	% SS Removed (% due to refactor.)	Residual
Blob	27 (12)	16 (5)	**+11 (+7)**
Feature Envy	16 (4)	11 (2)	**+5 (+2)**
Long Method	35 (18)	20 (8)	**+15 (+10)**
Misplaced Class	18 (11)	7 (3)	**+11 (+8)**
Promiscuous Package	26 (11)	13 (4)	**+13 (+7)**

Summary for RQ$_1$. We found statistically significant differences in the way textually and structurally detected code smells are treated by developers. Specifically, the intensity of code smell instances characterized by textual problems tends to decrease over time, while the intensity of structural code smells always increases over the release history of the projects considered. Notably, we also found that textual code smells are more prone to be refactored (+7% on average).

Summary for RQ$_2$. Refactoring is the activity that influences the decrease of intensity of textual *Blob*, *Long Method*, *Misplaced Class*, and *Promiscuous Package* instances the most. The decrease of the intensity of *Feature Envy* instances is instead influenced by enhancement activities. For structurally detected code smells, we generally did not find specific types of changes explaining their reduction in terms of intensity. Moreover, textual code smells are removed, on average, 11% more than structural code smells and 7% more as a consequence of refactoring operations.

3.3 Threats to Validity

Threats to *construct* validity concern the relationship between theory and observation, and are mainly related to the measurements we performed in our study. Specifically, we monitored the evolution of the level of smelliness of textual and structural code smells by relying on five tools, *i.e.*, TACO, DECOR [67], *JDeodorant* [100], and the approaches by Girvan *et al.* [38] and Atkinson and King [6]. All these tools have been empirically validated, providing good performances. Nevertheless, we are aware that our results can be affected by the presence of false positives and false negatives. At the same time, our results might be affected by the presence of the so-called "conceptual" false positives [31], *i.e.*, code smell instances identified by detection tools as true positives but not perceived as such by developers. However, we limited this threat by investigating code smell types that previous research found to be critical for developers [40], [73], thus focusing on code smell instances likely to be perceived by developers as actual design problems.

We re-implemented all the structural-based techniques except for *JDeodorant* because these tools are not publicly

available. To make sure that our implementation was not biased, we replicated the studies presented in [6], [38], [67], obtaining similar results. When re-implementing DECOR we only considered the structural-based rules of the approach, while we avoided the use of the textual-based one exploited by the approach to identify the controller classes in a system. While this choice might have implied a higher recall and a lower precision of the approach, we assessed the accuracy of the tool replicating the study conducted by Moha *et al.* [67], achieving comparable results. We are aware of other techniques proposed in literature to detect code smells [28], as well as of the availability of industrial tools such as inCode [43] or iPlasma [44]. However, we selected DECOR because of its proven performances [32], [34], [41], [50], [72], [74], and the techniques by Atkinson and King [6] and Girvan and Newman [38] because of the lack of other techniques able to detect the *Promiscuous Package* and *Misplaced Class* code smells. Moreover, it is worth noting that for most of the industry-level detectors an empirical evaluation of the performance is not available.

For the *commit goal* tag assignment to commits involving code smells, we used methodologies successfully used in previous work [102]. Moreover, we manually validated all the tags assigned to such commits.

Since our goal was to investigate whether and to what extent textual and structural code smells evolve differently, our study did not focus on the *overlapping* instances, *i.e.,* code smells identified by both types of detector. However, for sake of completeness we re-ran our evolutionary study on this set of code smells. Being these instances detected by both textual and structural techniques exploited, we needed to evaluate both their textual and structural intensity evolution. As a result, we found that the increasing of both the intensity scores was limited during the evolution: in other words, we observed that the smelliness of overlapping instances increased over time but in a limited manner. From a practical perspective, while the intensity of code smells detected by only textual analysis decreases over time and the one of structural code smells increase over time, the smelliness of the overlapping instances remains almost stable during the evolution. A likely cause behind this result is that textual issues somehow counterbalance the structural problems of classes, making developers able to keep the quality of such classes under control. The complete results of this analysis are available in our online appendix [76].

Threats to *internal* validity concern factors that could have influenced our results. The fact that code smells are removed may or may not be related to the types of changes we considered in the study, *i.e.,* other changes could have produced such effects. Since the findings reported so far allow us to claim correlation and not causation, in Section 4 we corroborate our quantitative results with a user study where we involve industrial developers and software quality experts, with the aim of finding a practical explanation of our quantitative results.

Threats to *conclusion* validity concern the relationship between the treatment and the outcome. In our case, a threat can be related to the fact that we tailored the experiments presented in this paper to the benefit of the tool because of the knowledge of the inner-workings of the textual-based tool we previously built, generating the so-called *observer-expectancy* effect [86]. However, it is important to note that in this study we have not evaluated the benefits provided by the textual information on single snapshots of a software system, but we conducted an evolutionary study aimed at understanding whether the textual smelliness evolves differently from the structural one: thus, the knowledge of the inner working of the tool as well as the knowledge of the detection performance of TACO did not influence our observations. Moreover, as a further mitigation all the authors of this paper looked critically at the experimental data to ensure the correctness of the results.

Threats to *external validity* concern the generalization of the results. A specific threat to external validity is 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 20 Java open source systems having different size and different domains. However, it could be worthwhile to replicate the empirical study on other projects written in different programming languages.

4 Study II: The Perception of Textual and Structural Code Smells

In the previous section, we found evident differences in the way developers treat textual and structural code smells. However, we cannot speculate on the reasons behind such differences by solely looking at historical data. On the one hand, it is possible that developers perceive textual code smells more as problems than structural code smells, while on the other hand it is also possible that this difference relies on the fact that textual code smells are easier to maintain. To better understand the reasons behind the results achieved in the previous analysis, we conducted a qualitative study investigating how developers perceive instances of structural and textual code smells.

4.1 Empirical Study Definition and Design

This study reports on a qualitative investigation conducted with (i) professional developers through a questionnaire, and (ii) software quality experts through semi-structured interviews, with the *goal* of investigating their perception of code smells with respect to different sources of information, *i.e.,* textual and structural. Hence, we designed this study to answer the following research questions (RQs):

> **RQ$_3$:** *Do developers perceive design problems behind textual code smells more than design problems behind structural code smells?*

RQ$_4$: *Do developers find textual code smells easier to refactor than structural code smells?*

The *context* of the study consists of three software projects, *i.e.*, Eclipse `3.6.1`, Lucene `3.6`, and ArgoUML `0.34` that have already been used in Section 3. For each of them, we selected four instances of *Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, and *Promiscuous Package* adhering to the following process:

1) we computed the level of structural and textual intensity for all packages, classes, and methods in Eclipse, Lucene, and ArgoUML;
2) we selected two instances of each code smell type correctly detected by structural tools but not by TACO;
3) we also selected two other instances of each type of code smell that are correctly classified by TACO and not by structural tools.

Note that we selected instances classified by the detectors as the ones having highest intensity (*i.e.*, the selected instances have similar intensity values). Moreover, to avoid possible biases caused by the interaction of more code smells [1], [109], we selected instances affected by only one single type of code smell. Finally, we also randomly selected two code elements of different granularity (*i.e.*, a package, a class, or a method) not affected by any of the code smells considered in our study. This was done to limit the bias in the study, *i.e.*, avoiding that participants always indicated that the code contained a problem and the problem was a serious one. Table 11 details the characteristics of code elements used in the experiment.

To answer our research questions, we invited industrial developers from different application domains having a programming experience ranging between 2 and 9 years and that generally work on Java development. The invitations were sent via e-mail. We have contacted 77 developers in total, receiving 19 responses ($\approx$ 25% of response rate). Each participant performed the tasks related to one single software system: eight participants analyzed code smell instances for Eclipse, eight participants for Lucene, while three participants for ArgoUML.

The participants received the experimental material via the `eSurveyPro`[10] online platform and, hence, they used their own personal computer with their preferred IDE to answer the proposed questions. The survey provided (i) a pre-test questionnaire, (ii) detailed instructions to perform the experiment, and (iii) the source code of the three projects involved in the study, *i.e.*, Eclipse, Lucene, and ArgoUML.

During the initial step, participants were asked to sign a statement of consent and fill in a pre-test questionnaire in which we collected information about their programming experience and background on code smells. Afterwards, each participant was required to inspect a total of twelve code elements related to one of the three projects in our study, namely five pairs of instances affected by either structural or textual code smells (*i.e.*, *Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, and *Promiscuous Package*) plus a pair of code elements not affected by code smells.

The experiment was composed of six consecutive sessions with two tasks each. During each session, participants performed two tasks related to the same type of code smell but detected by different tools: one task involving a code smell detected by TACO and a second task related to the same type of code smell but detected by structural-based tools. In each task, participants were asked (i) to analyze the target code component (either a package, a class or a method), (ii) to fill in a post-task questionnaire indicating whether the code component

10. http://www.esurveyspro.com

TABLE 11
Java code elements used as objects in the study.

Project	Code Component	Type	Detector
Eclipse	Parser.java	Blob	DECOR
	CompletionEngine.java	Blob	DECOR
	JavaModelManager.java	Blob	TACO
	OperatorExpression.java	Blob	TACO
	SelectionEngine.findAllTypes	Feature Envy	JDeodorant
	DeltaProcessor.resourceChanged	Feature Envy	JDeodorant
	BasicSearchEngine.getMatchRuleString	Feature Envy	TACO
	ClassFile.traverse	Feature Envy	TACO
	ClassFile.addAttributes	Long Method	DECOR
	Expression.checkCastTypesCompatibility	Long Method	DECOR
	CompilerOptions.set	Long Method	TACO
	ASTParser.createASTs	Long Method	TACO
	SortElementsOperation.java	Misplaced Class	Technique in [6]
	SourceRefElement.java	Misplaced Class	Technique in [6]
	SourceMethod.java	Misplaced Class	TACO
	SourceMapper.java	Misplaced Class	TACO
	org.eclipse.jdt.internal.core.builder	Promiscuous Package	Technique in [38]
	org.eclipse.jdt.internal.core.dom.rewrite	Promiscuous Package	Technique in [38]
	org.eclipse.jdt.internal.formatter	Promiscuous Package	TACO
	org.eclipse.jdt.internal.compiler	Promiscuous Package	TACO
	CodeFormatterVisitor.visit	Non Smelly	-
	AbstractCommentParser.parseReference	Non Smelly	-
	ITypeParameter.java	Non Smelly	-
	org.eclipse.jdt.internal.compiler.codegen	Non Smelly	-
Lucene	BrazilianStemmer.java	Blob	DECOR
	HTMLParserTokenManager.java	Blob	DECOR
	DocumentsWriter.java	Blob	TACO
	IndexWriter.java	Blob	TACO
	ToStringUtil.getRomanization	Feature Envy	JDeodorant
	HTMLStripCharFilter.nextChar	Feature Envy	JDeodorant
	QueryParser.getFieldQuery	Feature Envy	TACO
	DisjunctionMaxQuery.explain	Feature Envy	TACO
	WikipediaTokenizerImpl.getNextToken	Long Method	DECOR
	WeightedSpanTermExtractor.extract	Long Method	DECOR
	CheckIndex.checkIndex	Long-method	TACO
	AttributeSource.java	Misplaced Class	Technique in [6]
	PrefixAwareTokenFilter.java	Misplaced Class	Technique in [6]
	InstantiatedIndex.java	Misplaced Class	TACO
	QueryParser.java	Misplaced Class	TACO
	org.apache.lucene.store	Promiscuous Package	Technique in [38]
	org.apache.lucene.util.packed	Promiscuous Package	Technique in [38]
	org.apache.lucene.analysis.de	Promiscuous Package	TACO
	org.apache.lucene.analysis.payloads	Promiscuous Package	TACO
	CodeStream. invokeStringConcatenationAppendForType	Non Smelly	-
	DefaultCodeFormatterOptions. setJavaConventionsSettings	Non Smelly	-
	InternalCompletionProposal.java	Non Smelly	-
	org.apache.lucene.util	Non Smelly	-
ArgoUML	JavaRecognizer.java	Blob	DECOR
	JavaLexer.java	Blob	DECOR
	GeneratorJava.java	Blob	TACO
	Modeller.java	Blob	TACO
	Init.init	Feature Envy	JDeodorant
	JavaRecognizer.field	Feature Envy	JDeodorant
	GeneratorJava.generateClassifier	Feature Envy	TACO
	UMLComboBoxModel.targetChanged	Feature Envy	TACO
	UseCaseDiagramRenderer.getFigEdgeFor	Long Method	DECOR
	ParserDisplay.parseOperation	Long Method	DECOR
	PropPanelAssociationEnd.makeFields	Long Method	TACO
	ProjectBrowser.initMenus	Long Method	TACO
	ResourceBundleHelper.java	Misplaced Class	Technique in [6]
	ModuleHelper.java	Misplaced Class	Technique in [6]
	CriticUtils.java	Misplaced Class	TACO
	NotationHelper.java	Misplaced Class	TACO
	org.argouml.language	Promiscuous Package	Technique in [38]
	org.argouml.model	Promiscuous Package	Technique in [38]
	org.argouml.cognitive.ui	Promiscuous Package	TACO
	org.argouml.xml	Promiscuous Package	TACO
	UMLComboBoxModel.init	Non Smelly	-
	ClassdiagramLayouter.layout	Non Smelly	-
	FigAssociation.java	Non Smelly	-
	org.argouml.kernel	Non Smelly	-

was affected by a code smell or not, and (iii) to suggest possible refactoring operations aimed at removing a detected code smell. Moreover, participants were also asked to evaluate the proneness of the analyzed code element to contain a code smell instance as well as the severity of different source code properties (*e.g.,* size, complexity, number of dependencies, etc.) using a Likert scale ranging from 1 to indicate a very low risk to 5 to denote very high risk. The questionnaire is available in our online appendix [76]. Participants were also allowed to browse other classes or methods to better understand the responsibilities of the code component under analysis and find possible dependencies. Clearly, we did not reveal the types of code smells, nor whether they were detected by structural or textual tools.

Participants were instructed to spend no more than 30 minutes for completing each task and they were allowed to finish earlier if and only if they believed that all code smells were found (if any) *and* the corresponding refactoring operations were identified. Participants had up to four weeks to complete the survey.

To complement the analysis and receive opinions from people having a solid knowledge about source code quality [5] and code smells, in this stage we also recruited five software quality consultants: four consultants from the `Software Improvement Group` (SIG) in the Netherlands[11], and one consultant from `Continuous Quality in Software Engineering` (CQSE) in Germany[12]. Both companies carry out software quality assessments for their customers. The *mission* of both the companies is the definition of techniques and tools able to diagnose design problems in the clients' source code, with the purpose of providing consultancy on how to improve the productivity of their clients' developers. Our decision to involve these quality consultants was driven by the willingness to receive authoritative opinions from people that are used to finding design problems in source code. All participants have an average industrial experience of 4 years, an average programming experience of 9 years (one of them 20 years). This experiment was conducted in the headquarters of the SIG company when we interviewed the consultants from SIG, while the quality expert from CQSE was interviewed via Skype.

The consultants were asked to fill-in the same questionnaire provided to the industrial developers. However, due to time constraints, in this case the quality experts only answered the questions related to a subset of code smells, *i.e., Blob*, *Feature Envy*, and *Long Method*, belonging to two of the considered systems, *i.e.,* Eclipse and Lucene. We distributed the experimental material in order to have a balanced number of answers. For this reason, three experts answered questions related to Eclipse, the other two worked on code elements from Lucene.

11. https://www.sig.eu/en/
12. https://www.cqse.eu/en/

At the end of the experiment, the quality experts were also required to participate in an open discussion session of 30 minutes think about the tasks performed and to answer questions that we used to collect feedback for qualitative analysis. In particular, two of the authors first asked them to walk through the classes in order to explain the responsibilities they implemented; then, we asked them to explain if and how they identified a code smell in the source code (*e.g.,* which characteristic of the source code allowed them to recognize a code smell). The discussion session was conducted by the first two authors of this paper. The total duration of the experiment was 2 hours and 30 minutes, including the time needed to complete the experimental sessions, to fill in the questionnaires and participate in the discussion.

The data collected by the post-task questionnaires were used to answer $\mathbf{RQ}_3$ and $\mathbf{RQ}_4$. Specifically, we addressed $\mathbf{RQ}_3$ by analyzing the code smells identified by each participant for the code components; we compared their classification to the classification made by the textual and structural tools (*i.e., Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, and *Promiscuous Package*). Moreover, we also compared the distributions of the Likert values assigned by participants when providing the indication of the proneness of a code element to be involved in a code smell. In this way, we measured the perceived levels of risk for the two groups {*textual*, *structural*} of code smells to investigate how participants perceived the strength of code smells as identified by textual and structural based tools. To verify the statistical significance of the differences between the two distributions (*i.e.,* risk levels perceived for textual and structural code smells) we used the non-parametric Wilcoxon Rank Sum test with a significance threshold of $\rho - value = 0.05$.

For $\mathbf{RQ}_4$, we analyzed the refactoring operations suggested by participants for removing the identified code smells. Indeed, for each different code smell type there is a specific set of refactoring operations considered as suitable to address it. The refactoring associated with *Long Method* is *Extract Method*, for *Feature Envy* the corresponding refactoring is *Move Method*, for *Blob* an *Extract Class* refactoring is advised, for *Misplaced Class* a *Move Class* refactoring should be applied, and for *Promiscuous Package* the associated solution is represented by the *Extract Package* refactoring [36], [82], [98].

Finally, we provide an overview of the discussion, as well as hints provided by quality consultants during the open discussion session that followed the interview.

4.2 Analysis of the Results

Before answering the two research questions formulated in the previous section, we analyze to what extent professional developers perceived the presence of a code smell in code elements not containing any of the code smells considered in our study. As previously explained, this is a sanity check aimed at verifying whether participants were negatively biased. As a result, none of the involved

developers marked such code elements as affected by code smells. Hence, this result indicates the absence of a negative bias in the respondents.

Turning to the answers provided by participants when analyzing code elements affected by a code smell, Figure 1 depicts the bar plots reporting the percentage of code smells perceived (in grey) and identified (in black) by the professional developers who participated to the survey.

Specifically, with *perceived* we mean that the developer recognized the presence of a design problem in the code element inspected, but she was not able to define the code smell affecting it. Conversely, with *identified* we mean that the developer was not only able to understand that the inspected class was affected by a design problem, but she also provided a clear explanation of the problem affecting the class. Note that we consider a code smell as identified only if the design problem described by the participant can be directly traced to the definition of the code smell affecting the code element. For instance, when analyzing a Long Method instance, one of the involved developers explained that:

> *"This method is too long, hard to understand, test, and profile. It does multiple things – the comments already indicate that."*

Clearly, the definition given by the developer matches the definition of the code smell provided by Fowler [35]. Thus, we considered it as a code smell identified. Obviously, a code component that is correctly identified is also perceived (the opposite is not true). Besides the results on the perception of textual and structural code smell instances, Table 12 reports the percentage of refactoring operations correctly suggested by participants. Finally, Table 13 illustrates the code smells observed as well as the refactoring operations suggested by each quality expert involved in the study. The rows represent participants while the columns depict the type of code smell (*i.e.,* Blob, Feature Envy and Long Method) and the source of information used for the detection (*i.e.,* structural or textual).

Looking at Figure 1 we can immediately make two important general observations: first of all, in most cases textual code smells are considered actual problems by developers, even though they do not exceed any structural metrics' value. Secondly, not only does the analysis reveal important differences between textual and structural code smells in the way developers perceive them, but also that textually detected code smells are generally correctly identified. At the same time, professional developers are also generally able to provide good suggestions on how to refactor the identified code smells (see Table 12). This result confirms the observations made by Mäntylä and Lassenius [62], who showed how more experienced developers have higher abilities in the identification of code smells affecting the source code. This is particularly true for textual code smells, where the developers almost always indicate the right refactoring to apply (for all the code smells but *Feature Envy* more than 79% of participants described well the refactoring aimed at removing the code smell). This trend is confirmed when considering the answers provided by software quality consultants (see Table 13). In this case, we can observe that in all the cases the quality experts perceived the presence of a design problem in the proposed code components.

TABLE 12
Percentage of refactoring operations correctly suggested by professional developers.

Code Smell	Structural		Textual	
	#	%	#	%
Long Method	13	68%	15	79%
Feature Envy	0	0%	5	26%
Blob	9	47%	16	84%
Promiscuous Package	1	5%	17	89%
Misplaced Class	2	10%	18	95%

Blob. Blob instances characterized by textual problems are always perceived and correctly classified by the participants. On the other hand, only 53% of the structurally detected instances are identified by developers, even if the problem is always perceived. While this result is in line with previous findings demonstrating that complex or long code is more easily detectable by developers [73], the different nature of the code smells lead us to think that developers better understand design problems affecting textual properties of source code.

As for the quality consultants, structural blobs are correctly identified by participants in only three out of five cases. Note that in these cases participants also correctly indicate the refactoring operation (*i.e., Extract Class*) for removing the code smell. In the remaining two classes, they describe symptoms referable to other code smells, such as *Complex Class* [19] and *Duplicate Code* [35]. Moreover, participants were not able to identify an appropriate refactoring to remove these code smells. During the open discussion, the quality experts were invited to think aloud [55] on the design problems affecting the analyzed code components. In this session, participants re-elaborated their analysis, correctly detecting the previously missed *Blob* instances. They also explained that the main reason of the misclassification is *the extreme complexity of the* (structurally detected) *code components that does not allow the correct identification of the design problems affecting them*. For textual blobs, the discussion is different. Here we observed a complete agreement with the experts' perception: all classes detected as blobs by TACO are also identified as actual *Blob* instances by the participants. At the same time, it is worth observing that participants also identified the correct refactoring operation aimed at removing the code smell.

The open discussion session confirms our conjecture, *i.e.,* developers better understand code smells characterized by textual problems.

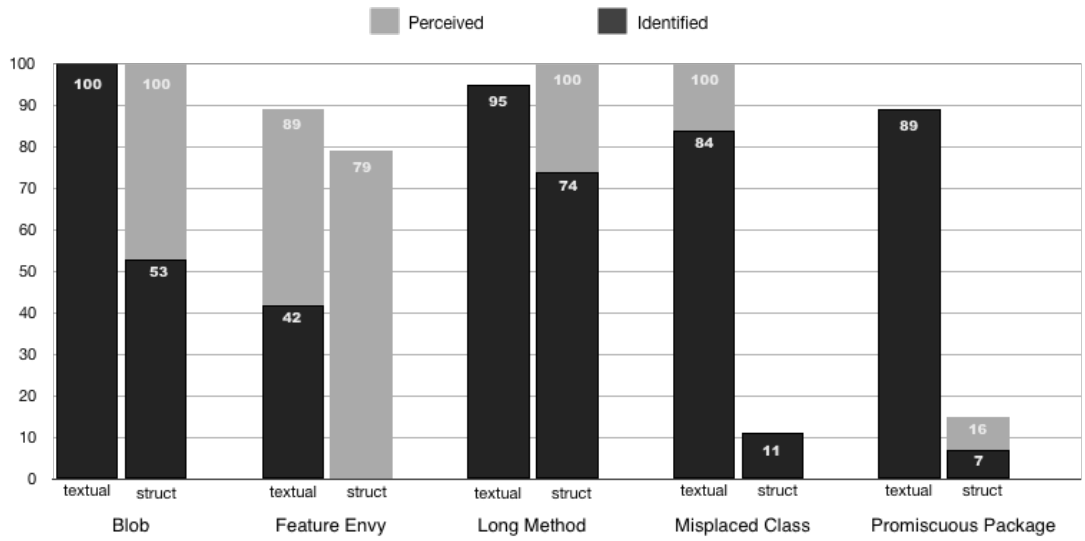

Fig. 1. Percentage of Textual and Structural Code Smells Perceived and Identified by Professional Developers.

TABLE 13
Results obtained from Study II. Labels marked with (*) indicate that the code smells identified by participants match the classification provided by textual or structural tools

Participant	Structural						Textual					
	Blob		Feature Envy		Long Method		Blob		Feature Envy		Long Method	
	Flaws	Refactoring	Flaws	Refactoring	Flaws	Refactoring	Flaws	Refactoring	Flaws	Refactoring	Flaws	Refactoring
Expert1	Blob* Long Param. List	Extr. Class*	High Complex.		Long Method*		Blob* Long Param. List High Complex.	Extr. Class*	Feature Envy* High Complex.	Move Method*	Long Method*	Extr. Method*
Expert2	Code Duplic.	Clone Reun.	High Complex.	Rename	Long Method*	Extr. Method*	Blob*	Extr. Class*	High Complex. Bad Identifiers	Rename	Long Method* High Complex	Extr. Method*
Expert3	Blob*	Extr. Class*	Long Method	Extr. Method	Long Method*	Extr. Method*	Blob* Bad Identifiers Redundant Code	Extr. Class*	Long Method	Extr. Method	Long Method* Feature Envy	Extr. Method* Move Method
Expert4	Complex Class		Long Method	Extr. Method Extr. Class	Long Method*	Introd. Polym.	Blob*	Extr. Class*	Feature Envy*	Move Method*	Long Method* Bad Identifiers	Extr. Method*
Expert5	Blob*	Extr. Class*	Long Method	Extr. Method	Long Method*	Extr. Method*	Blob*	Extr. Class*	Long Method Bad Identifiers	Extr. Method Rename	Long Method*	Extr. Method*

Long Method. When considering the *Long Method* code smell, the discussion is quite similar. Indeed, here we found that almost all the industrial developers perceived a problem. However, there are two details to further analyze. In the first place, unlike the structurally detected ones, 95% of the long methods detected by TACO and perceived by developers were also correctly identified. On the other hand, there was a specific case in which a textual *Long Method* instance was marked as not affected by any design problem. It is the `ASTParser.createASTs` method of the Eclipse project, which is responsible for the analysis of the source code classes of a Java project and the subsequent creation of the Abstract Syntax Tree (AST) for all of them. Even though the method does not have excessive length (*i.e.,* 68 lines of code), it should be considered as a design problem since it manages all the operations needed to build the AST of a class (*i.e.,* reading, parsing, and AST building). However, one of the professional developers involved in the study did not perceive this code smell.

On the other hand, the quality experts perceived and correctly identified all instances of long methods detected by either approach. However, Table 13 highlights evident differences between the refactoring operations suggested by participants to remove structural and textual code smells. Indeed, while for structurally detected code smells the correct refactoring (*i.e., Extract Method*) was identified in only three out of five cases, for textual code smell participants always indicated the *Extract Method* refactoring as ideal solution for removing the identified code smells. For instance, when indicating the possible refactoring for the long method `TieredMergePolicy.findMerges` of the `Apache Lucene` project, a quality consultant answered that "*parts of the method are separated by comments; at which points usually an extract method refactoring should be applied; then the names of the new methods can replace the comments*". This example is quite representative since the quality expert not only identified the correct refactoring to use, but also gave us hints on how the refactoring may be applied practically.

Feature Envy. For the *Feature Envy*, we obtain different results when compared to the two types of code smells discussed above. While the problem is generally perceived by professional developers, none of them was able to characterize the symptoms behind the struc-

tural instances of this code smell. Conversely, textual instances are perceived more (89% vs 79%) and in some cases (42%) participants were able to describe the problem well. Concerning the refactoring suggestions, only five developers who identified the problem correctly suggested the application of a *Move Method* refactoring. For instance, let us consider the case of the `QueryParser.getFieldQuery` method of the Lucene system, affected by *Feature Envy* because it is more closely related to the `PhraseQuery` class. When analyzing it, developer #6 claimed that "*the method seems to be more related to the class PhraseQuery, even if there are not so many dependencies between the two classes*".

The observations that we collected from surveying the quality experts are in line with those reported above. Indeed, in the majority of the cases the consultants did not classify this code smell type correctly, even if they perceived that the subject code components actually had some design problems. However, it is worth noting that the method `getMatchRuleString` from the class `BasicSearchEngine` (Eclipse), and the method `explain` from class `DisjunctionMatchQuery` (Apache Lucene) were correctly classified as *Feature Envy* instances in agreement with the detection made by TACO (textual tool), and, at the same time, correct refactoring solutions were suggested (*Move Method* refactoring). On the one hand, the higher ability of developers to detect textual *Feature Envy* instances is a likely consequence of the fact that textual coupling is more easily perceived than structural coupling [15]. On the other hand, hints provided by participants in the open discussion can allow us to draw a conclusion on why developers often do not identify this code smell type. Indeed, they reported that "*dependency on other classes is the less important point*", indeed, "*as long as a method is short and well documented (e.g., with proper identifiers), there is no real need to move it to a different class*".

Misplaced Class and Promiscuous Package. The importance of textual aspects of source code for the practical usefulness of code smell detectors is even more evident when considering *Misplaced Class* and *Promiscuous Package* code smells. Indeed, in these cases the structurally detected instances are almost never perceived and identified, while the results achieved for textual instances are exactly the opposite (see Figure 1). At the same time, it is worth observing that the higher the granularity of the code smell, the lower the ability of the developers in perceiving structurally detected code smells. A possible reason behind this result is related to the need of developers of having higher level information to build a cognitive model to comprehend larger parts of source code [90], [66]. On the other hand, textual analysis seems to be easier to perform by developers to comprehend the problems affecting a higher-level code component such as a package.

General observations. In general, we observe two different trends for the code smell types (Table 13) detected by quality experts depending on whether code components are affected by structural and textual code smells. While for structurally detected code smells the experts generally identified only one design problem per code component (often missing the actual problem affecting the analyzed code), for textual code smells participants detected other problems related to textual aspects of source code, such as the presence of poor identifiers, in addition to the main design problem. For example, *Expert3* identified three different design problems for the class `OperatorExpression` extracted from the Eclipse project: it is a *Blob* (in agreement with TACO) affected by redundant code and with meaningless identifiers.

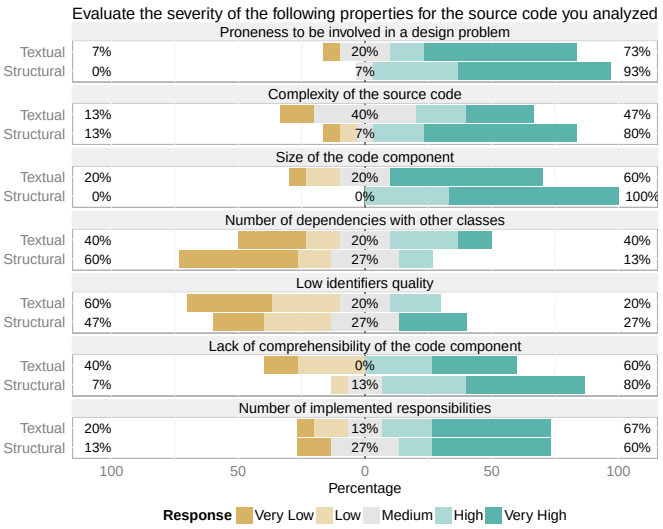

Fig. 2. Likert chart for post-task questionnaires, where the same questions are asked at the end of each task

At the end of each task, participants were asked to evaluate the severity of different properties (*e.g.*, size, complexity, number of dependencies, etc.) for the analyzed code using a Likert scale intensity from very-low to very-high. Figure 2 compares the scores assigned to the code smells according to whether they were detected by textual or structural tools.

Such an analysis allows us to gain more insight into the perception of code smells stemming from different sources of information.

In most cases, the structural instances were marked as more severe than the textual ones. While developers considered structural code smells as more severe, they were not able to correctly diagnose the problems behind this type of code smells because of their higher complexity. Therefore, we can claim that the analysis of textual properties is useful for code smell detection since they allow developers to understand design problems affecting code components. At the same time, considering the refactorings that participants suggested, there is the need for tools providing better support in the refactoring of structural code smells.

When considering properties as code size and num-

ber of responsibilities, we did not find any statistical difference between structural and textual code smells. However, in the open discussion session, the quality experts observed that "*size and number of responsibilities are quite related to each other*" when the goal is to understand whether a specific code unit is affected by a code smell or not. Indeed, they observed that if a class is too large this is a first symptom for the class to be a potential *Blob*. However, they also noted that "*raw number of lines is not the end of the story because developers have to read the code*", build their own mental model and then "*figure out whether the class or the method implements too many responsibilities, which is the main reason of other issues*". Moreover, the quality experts observed that "*the number of lines in a method and the presence of unrelated parameters are the two aspects to look at*" when identifying code smells at method level, where unrelated parameters denote "*parameters that are interpreted by developers as unrelated looking at the identifier names*". Finally, for code smells at the class level they mentioned that "*the number of concepts that can be derived by method names and attributes*" is the main source of information while the raw number of lines is not a good approximation of the phenomenon. Through this analysis, we learned that the size of a code unit does not always represent a good measure to identify code smells, since to better evaluate the presence of spurious responsibilities the analysis of textual components is needed.

Participants also evaluated the severity of two other *dimensions* for the code components analyzed, namely complexity and comprehensibility. From Figure 2 we can observe that participants perceived the structural code smells as more complex (80%) with respect to textual code smells (47%). The Wilcoxon test reveals that such difference is marginally significant (ρ=0.057) with a negative *medium* Cliff's d effect size (d=-0.36). Similarly, for comprehensibility we observe that 80% of participants labeled the structural code smells as difficult to comprehend (high or very high severity) with respect to 60% of participants for textual code smells. Such a difference is statistically significant with ρ=0.029 and has a negative *small* Cliff's d effect size (d=-0.275). This result highlights that structural code smells are perceived as more difficult to deal with.

To practically understand the effect of these differences, during the open discussion we asked the quality consultants to directly compare pairs of code components (*e.g.,* textual vs structural blobs) in order to illustrate possible steps for the application of a refactoring. All the experts stated that in each pair there was "*one instance much more complex to understand, which makes difficult the derivation of a precise refactoring operation that should be applied*". For example, *Expert1* reported that "*looking at the raw number of decision points*", the two methods `addAttributes` from the class `ClassFile` (structural long method) and `set` from the class `Options` (textual long method) "*seem to be equally complicated*", but by looking more carefully at the code, the method `set` "*is instead well structured because there is a pattern in the if conditions. For this method, it is easier to imagine that a potential way to fix the problem is to write different methods for different if conditions*". While the structural long method `addAttributes` from class `ClassFile` "*is very complicated and deciding the refactoring operations to apply is not so simple*". As another example, *Expert4* reported that the method `explain` from class `DisjunctionMatchQuery` "*is quite complicated despite its length: it contains too many concepts and some parameters are simply passed to other methods*". It is worth noting that this method contains only 17 lines of code and it is detected as *Feature Envy* by TACO (textual tool). The structural *Feature Envy*, *i.e.,* method `nextChar` from `HTMLStripCharFilter`, is perceived "*more complex to manage because it contains too many responsibilities spread across external classes; it implements a state machine with no meaningful names to help the comprehension*". Thus, we can claim that the higher comprehensibility of textual code smells seems to help developers in finding appropriate refactoring solutions. The result is in line with what we found studying the evolution of textual code smells, *i.e.,* they are generally more prone to be refactored, as well as more prone to be subject to activities aimed at reducing their intensity over time. More in general our results complement previous findings in the field, showing that even though developers are sometimes not properly able to perceive the presence of code smells in source code [4], [109], the usage of textual analysis provides useful additional information for the diagnosis of code smells. More importantly, these results highlight the need for novel detection approaches able to extract recommendations closer to the developers' perception of code smells, *e.g.,* by combining structural and textual information.

Summary for RQ$_3$. Textual code smells are generally perceived as actual design problems and correctly diagnosed by developers. Conversely, only in a few cases structurally detected code smells are correctly identified even though developers perceive them as more severe. As a consequence, structurally detected code smells tend to be not refactored over time, as pointed out in our mining study. We also noticed that while code smells affected by textual problems are easy to identify independently from the granularity of the code smell, structural code smells at a higher level of granularity are more difficult to identify.

Summary for RQ$_4$. Both industrial developers and quality consultants found code elements affected by structural code smells as more difficult to understand making the identification of the design problem and of the corresponding refactoring operations more difficult. This finding gives a practical explanation of the results of the mining study in Section 3: since textual code smells are perceived as easier to understand, developers more likely derive accurate refactoring operations to reduce their intensity. Therefore, we can conclude that textual

analysis is as useful as the more traditional structural analysis. Moreover, tools helping developers in refactoring structural code smells are particularly desirable, due to the higher complexity of dealing with these types of code smells.

4.3 Threats to Validity

The main threat related to the relationship between theory and observation (*construct* validity) is represented by the subjectivity of textual and structural code smell perception. To limit this bias, we carefully selected the participants by recruiting developers having industrial experience as well as a quite long career in software development. Moreover, we complemented our analysis by involving five quality consultants having a solid knowledge about software quality and code smells.

As for the factors potentially influencing our findings (threats to *internal* validity), we ensured that participants were not aware of the types of code smells affecting the provided instances nor the underlying techniques used for detection. Another threat is related to the fact that developers performed the study using their preferred IDE: this might have led them to browse the text and focus the attention on the identifiers, thus biasing the results in favor of the textual analysis. However, it is important to note that the experimental design reproduces a real-case scenario, where developers have to read code components and understand how to maintain (and refactor) them. Thus, our results on the one hand suggest that textual information is useful to understand code smells, while on the other hand highlight that visualization features might be required to abstract structural information, helping developers in diagnosing and refactoring structural code smells. Indeed, even if some IDEs might provide some visualization features we cannot say whether developers performing the task remotely used them. However, we observed that such features where not used by the interviewed quality experts.

Finally, as for the generalizability of our results (threats to *external* validity), possible threats can be related to the set of chosen objects and to the pool of the participants in the study. Concerning the chosen objects, we are aware that our study is based on code smell instances detected in two Java systems only, and that further studies are needed to confirm our results. In this study, we had to constrain our analysis to a limited set of code smell instances, because the task to be performed by each respondent had to be reasonably small (to ensure a decent response rate). As for the participants, we involved 19 industrial developers. While a replication of the study aimed at corroborating the results achieved could be worthwhile, the developers involved have a strong experience in the development. At the same time, we complement our analyses by involving 5 quality consultants having a solid knowledge about software quality and code smells.

5 Related Work

This section reports the literature related to (i) empirical studies investigating the evolution of code smells as well as their impact on non-functional attributes of the source code, (ii) empirical studies on the developers' perception of code smells, and (iii) studies investigating how developers perform refactoring operations in practice.

5.1 Code Smell Evolution

Concerning the evolution of code smells, three main studies have been proposed in literature. Tufano *et al.* [102] mined the evolution history of 200 software projects in order to investigate when and why code smells are introduced. Their findings show that code smells are generally introduced during the first commit of the artifact affected by the code smell, even though several instances are introduced after several maintenance operations performed on an artifact. Moreover, code smells are introduced because of operations aimed at implementing new features or enhancing existing ones, even if in some cases also refactoring can be the cause of code smell introduction.

Chatzigeorgiou and Manakos [22] found that the number of code smells generally increase over time, and that only few instances are removed from software projects. Interestingly, in most cases their removal is not due to specific refactoring operations, but is rather a side-effect of maintenance operations [22]. Peters and Zaidman [81] on the one hand confirmed these results, while on the other hand they found that even if developers are sometimes aware of the presence of code smells in their code, they often deliberately postpone refactoring activities. Our investigation confirms that code smells generally persist over time and that they are not often object of specific refactoring activities. However, we found that the textually detected code smells are more prone to be refactored.

Vidal *et al.* [104] observed that the number of code smells suggested by existing metric-based tools usually exceed the amount of design problems that developers can deal with. For this reason, they proposed a prioritization approach based on previous modifications of a class, important modifiability scenarios for the considered system, and the relevance of the code smell type. Our study on code smell perception confirms that not all code smells are equally important for developers, since structurally detected ones are more complex to diagnose and refactor than textually detected ones.

Recently, Oizumi *et al.* [70] investigated whether agglomerations of code anomalies can indicate evolutionary code smells. Key results from the study highlight that (i) 80% of semantic agglomerations are related to code smells, and (ii) change history information is not always able to locate code smells because they are often already introduced in the initial versions of systems. The results of our study are complementary to Oizumi *et al.*'s

[70], since we demonstrated how textually detected code smells are treated differently than structural ones.

5.2 Impact of Code Smells on Non-Functional Attributes of Source Code

In the first place, Abbes *et al.* [1] reported the results achieved from three controlled experiments aimed at investigating to what extent the *Blob* and *Spaghetti Code* code smells impact on program comprehensibility. In particular, they found that the presence of single instances of these code smells does not result in a reduction of program comprehensibility, while the co-occurrence of the two code smells significantly impact on developers' ability to understand the source code. These results have been confirmed by Yamashita and Moonen [109], who deeper analyzed the interaction of different code smells. They observed that maintenance issues are strictly related to the presence of more code smells in the same file. Later on, the same authors analyzed the impact of code smells on maintainability characteristics [111], identifying which maintainability factors are reflected by code smells and which ones are not basing their results on (i) expert-based maintainability assessments, and (ii) observations and interviews with professional developers. Yamashita *et al.* [110] also proposed a replicated study where they investigated the co-occurrence problem in both open and industrial systems, finding that the relation between code smells vary depending on the type of system taken into account. In order to ensure a fair comparison, the user study proposed in this paper selects code components (*i.e.*, packages, classes, or methods) affected by a single code smell instance.

Several previous studies tried to explain the relationship between code smells and change- and fault-proneness. Specifically, Khomh *et al.* [48] presented the results of a preliminary analysis where they found that smelly classes are significantly more change-prone than other classes not affected by code smells. In another study, the authors also discovered [49] that classes affected by code smells are more fault-prone than non-smelly classes. This is especially true for some specific code smell types, such as *Message Chains* [35]. The results achieved in the paper mentioned above have been confirmed by Gatrell and Counsell [37], who quantified the effect of refactoring on change- and fault-proneness of classes. The authors monitored a commercial C# system for one year, detecting the refactoring operations applied during the first four months, and examining such refactored classes for the second four months in order to determine whether and to what extent the refactoring actually produces classes having lower change- and fault-proneness. At the same time, the authors compared such classes with the classes of the system that were not refactored. Their findings revealed that refactored classes have a lower change- and fault-proneness, both considering the time period in which the same classes were not refactored and classes in which no refactoring operations were applied.

Li and Shatnawi [58] obtained similar results analyzing the post-release bugs of three software systems. The findings suggest that the refactoring activities improve the architectural quality, but also reduce the likelihood that a class will contain a bug in the future. Olbrich *et al.* [71] investigated the maintainability of two specific code smells, *i.e.*, *God Class* and *Brain Class*, finding that they were generally changed less after refactoring, but also that they contained a lower number of defects. D'Ambros *et al.* [26] studied the relationship between *Feature Envy*, *Shotgun Surgery* and the fault-proneness of code elements, however they have not found a clear correlation between the two phenomena. Finally, recent findings demonstrated the usefulness of smell-related information to improve the performances of bug prediction models [79].

5.3 Developers' Perception of Code Smells

Some previous studies analyzed the perception of code smells from the developers' perspective. Clearly, these studies are related to the one proposed in this paper. Arcoverde *et al.* [4] investigated how developers react to the presence of code smells in code. Their results indicate that code smells tend to remain in the source code for a long time. The main reason is that developers do not refactor the source code in order to avoid API modifications [4]. Yamashita and Moonen [112] reported the results of an empirical study aimed at evaluating the code smell severity perceived by industrial developers. They found that over 32% of the surveyed developers do not know the concept of a code smell and that in most cases their removal is not a priority since continuous deadlines and/or lack of adequate automatic tools represent effective deterrents. In addition, Sjøberg *et al.* [97] demonstrated that sometimes code smells do not represent an actual problem for the maintainers, and that class size impacts on maintainability performance more than the presence of code smells. The results of our study demonstrate how this observation is not entirely true when considering textual code smells. Indeed, developers tend to better diagnose design problems characterized by textually detected code smells with respect to structurally detected ones.

Mäntylä [62] reports the results of an empirical study conducted in the context of a Finnish software product company, where the author observed that (i) developers with different levels of experience rate the existence of a code smell affecting the source code differently, and (ii) code metrics do not fully correlate with the developers' perception of code smells. Our findings confirm both aspects studied by Mäntylä: on the one hand, we found that in general software quality consultants were able to better diagnose code smells than the other industrial developers; on the other hand, we observed that structurally detected code smells are felt harder to understand by developers.

More recently, Palomba *et al.* [73] surveyed developers

from three open source systems with the aim of understanding how they perceive 13 different code smell types. They observed that code smells related to complex/long source code are generally considered harmful for developers, while the perception of other code smell types strongly depend on the "intensity" of the problem. With respect to this paper, our empirical investigation (i) involves code smells detected using different sources of information, *i.e.,* textual and structural, and (ii) analyzes the way the developers treat code smells of different nature during evolution.

Finally, Aniche *et al.* investigated whether metric values are contextual to the architecture of a system [2]. Their study indicates that metric values differ significantly depending on the architectural context, with high metric values in certain architectural circumstances being found to be unproblematic. This leads to the conjecture that thresholds for metric values used in code smell detection should be adjusted depending on the architectural context.

5.4 Refactoring of Code Smells

In this category, there are three studies that are more closely related to the one presented here.

Firstly, Wang *et al.* [106] characterized the factors that motivate developers in performing refactoring operations, finding two main categories, *i.e., intrinsic motivators* and *external motivators*. The former are factors mainly related to the willingness of developers to maintain their code cleaned (*e.g.,* the *Responsibility with Code Authorship*). On the other hand, the latter category refers to motivations related to the willingness of developers to achieve external rewards, such as *Recognition from Others*, *i.e.,* high technical ability can help the software developers gain recognition.

Bavota *et al.* [10] investigated the relationship between code quality, intended as code metric values and code smells, and refactoring. Specifically, they analyzed the change history of 63 releases of three open source systems. The results indicate that in most cases refactoring is not performed on classes having a low metric profile, while almost 40 % of refactoring operations are performed on classes affected by a code smell. Unfortunately, they also found that only a small percentage (*i.e.,* 7%) of refactorings are effective and help in removing the code smells. In this paper, we demonstrate how refactoring operations performed on code elements characterized by textual problems are more efficient, other than more frequent, than refactoring operations applied on structurally detected code smells.

More recently, Silva *et al.* [94] surveyed `Github` contributors of 124 software projects in order to find the motivations behind the application of refactoring operations. The main finding is that refactoring is mainly driven by changes in the requirements rather than by the presence of code smells. This finding was also confirmed in a recent study on the relationship between changes and refactoring [78]. With the empirical study conducted in Section 3 we complement the findings reported in the paper by Silva *et al.* [94] by studying the way developers act on code smells of different nature, finding that refactoring code smells is more common when the code elements are affected by textual code smells.

6 Conclusion and Future Directions

In this paper, we conducted a systematic investigation to analyze how developers perceive and act on code smell instances depending on which source of information is used for the detection, *i.e.,* structural versus textual. To this aim, we mined historical data from 301 releases and 183,514 commits of the 20 open source projects in order to monitor developers' activities on textual and structural code smells over time. We also conducted a qualitative study with industrial developers and software quality consultants to analyze how they perceive and react to code smell instances of different nature. The results of the study highlight four key findings:

1) Textually detected code smells are generally perceived by industrial developers as actual design problems and as dangerous as structural code smells, even if they do not exceed any structural metrics' thresholds.
2) Textual and structural code smells are treated differently: the intensity of the former tends to decrease over time, while the intensity of the latter tends to increase.
3) Textually detected code smells are more prone to be resolved through refactoring operations or enhancement activities.
4) Textual code smells are perceived as easier to understand. As a consequence, accurate refactoring operations can be derived more quickly.

Our findings confirm that software metrics are not the unique source of information that developers use to evaluate the quality of source code [20], [96], [21]. The results achieved represent a call to arms for researchers and practitioners to investigate the human's perspective for building a new generation of code smell detectors and refactoring tools. Our future agenda includes, indeed, the definition of a new set of hybrid techniques that efficiently use both sources of information to detect code smells.

Moreover, our results shed light on an important motivation behind the lack of refactoring activities performed to remove code smells [4], [10], [94], *i.e.,* it seems developers are not able to practically work on structurally detected code smells. Therefore, our results clearly highlight the need of having techniques and tools able to help developers in refactoring code smells characterized by structural design problems, which are difficult to understand for developers. Finally, our findings also highlight the need of conducting controlled experiments with developers on the perception and identification of

structural code smells, to understand whether the use of IDEs including visualization facilities improves the way developers identify and remove structural code smells.

REFERENCES

[1] 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 Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 181–190.

[2] M. Aniche, C. Treude, A. Zaidman, A. van Deursen, and M. A. Gerosa, "SATT: Tailoring code metric thresholds for different software architectures," in *International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2016, pp. 41–50.

[3] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

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

[5] 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.

[6] D. Atkinson and T. King, "Lightweight detection of program refactorings," in *Asia-Pacific Software Engineering Conference*, Dec 2005, pp. 8–18.

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

[8] 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.

[9] G. Bavota, R. Oliveto, A. D. Lucia, G. Antoniol, and Y. G. Guhneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *International Conference on Software Maintenance*, Sept 2010, pp. 1–5.

[10] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, no. 9, pp. 1–14, Sep. 2015.

[11] 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.

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

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

[14] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in eclipse: The aries project," in *International Conference on Software Engineering*, 2012, pp. 1419–1422.

[15] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2013, pp. 692–701.

[16] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. de Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 1, pp. 4:1–4:33, Feb. 2014.

[17] 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.

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

[19] 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*. John Wiley and Sons, March 1998.

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

[21] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?" *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 3, pp. 24:1–24:28, Jun. 2016.

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

[23] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

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

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

[26] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *International Conference on Quality Software*, July 2010, pp. 23–31.

[27] 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.

[28] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *nternational Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '16. New York, NY, USA: ACM, 2016, pp. 18:1–18:12. [Online]. Available: http://doi.acm.org/10.1145/2915970.2915984

[29] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *International Conference on Software Maintenance*, 2003, pp. 23–.

[30] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012.

[31] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1, March 2016, pp. 609–613.

[32] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, "An experience report on using code smells detection tools," in *International Conference on Software Testing, Verification and Validation*, March 2011, pp. 450–457.

[33] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla, "Code smell detection: Towards a machine learning-based approach," in *International Conference on Software Maintenance*, Sept 2013, pp. 396–399.

[34] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment." *Journal of Object Technology*, vol. 11, no. 2, pp. 5: 1–38, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/jot/jot11.html#FontanaBZ12

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

[36] ——, "Refactoring catalog," 2017. [Online]. Available: https://refactoring.com

[37] M. Gatrell and S. Counsell, "The effect of refactoring on change and fault-proneness in commercial c# software," *Science of Computer Programming*, vol. 102, no. 0, pp. 44 – 56, 2015.

[38] 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.

[39] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[40] M. Hall, M. A. Khojaye, N. Walkinshaw, and P. McMinn, "Establishing the source code disruption caused by automated remodularisation tools," in *International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 466–470.

[41] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, pp. 33:1–33:39, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629648

[42] D. Hosmer and S. Lemeshow, *Applied Logistic Regression (2nd Edition)*. Wiley, 2000.

[43] inCode Team, "incode tool," 2017. [Online]. Available: https://marketplace.eclipse.org/content/incode-helium

[44] iPlasma Team, "iplasma tool," 2017. [Online]. Available: http://loose.upt.ro/reengineering/research/iplasma

[45] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bulletin del la Société Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, 1901.

[46] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *International Conference on Automated Software Engineering*. ACM, 2010, pp. 113–122.

[47] 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.

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

[49] 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.

[50] 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.

[51] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *International Conference on Quality Software*. IEEE, 2009, pp. 305–314.

[52] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring: Challenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, Jul. 2014.

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

[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] H. Kuusela and P. Paul, "A comparison of concurrent and retrospective verbal protocol analysis," *The American Journal of Psychology*, vol. 113, no. 3, pp. 387–404, 2000.

[56] 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.

[57] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and Software*, vol. 7, no. 4, pp. 325 – 339, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/016412128790032X

[58] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, pp. 1120–1128, 2007.

[59] A. Likas, N. Vlassis, and J. J. Verbeek, "The global k-means clustering algorithm," *Pattern Recognition*, vol. 36, no. 2, pp. 451 – 461, 2003, biometrics.

[60] E. Lim, N. Taksande, and C. B. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE Software*, vol. 29, no. 6, pp. 22–27, 2012.

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

[62] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, 2006. [Online]. Available: http://dx.doi.org/10.1007/s10664-006-9002-8

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

[64] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9:1–9:13, Sept 2012.

[65] ——, "Detection strategies: Metrics-based rules for detecting design flaws," in *International Conference on Software Maintenance*, 2004, pp. 350–359.

[66] A. V. Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, Aug 1995.

[67] 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.

[68] 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.

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

[70] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 440–451. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884868

[71] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 390–400.

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

[73] 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 *International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 101–110.

[74] 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.

[75] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *International Conference on Program Comprehension*, May 2016, pp. 1–10.

[76] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Online appendix: An extensive comparison between textual and structural smells," https://doi.org/10.6084/m9.figshare.3102244.

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

[78] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *International Conference on Program Comprehension*. IEEE, 2017, pp. 176–185.

[79] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *International Conference on Software Maintenance and Evolution*. IEEE, 2016, pp. 244–255.

[80] D. Pelleg and A. W. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters," in *International Conference on Machine Learning*. Morgan Kaufmann, 2000, pp. 727–734.

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

[82] B. Pietrzak and B. Walter, "Exploring bad code smells dependencies," in *International Conference on Software Engineering: Evolution and Emerging Technologies*. Amsterdam, The Netherlands, The

Netherlands: IOS Press, 2005, pp. 353–364. [Online]. Available: http://dl.acm.org/citation.cfm?id=1565142.1565180

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

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

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

[86] D. L. Sackett, "Dl: Bias in analytic research," *J Chron Dis*, pp. 32–51, 1979.

[87] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 1, pp. 6:1–6:44, Oct. 2014.

[88] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending move method refactorings using dependency sets," in *Working Conference on Reverse Engineering*, Oct 2013, pp. 232–241.

[89] B. H. Sellers, L. L. Constantine, and I. M. Graham, "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)." *Object Oriented Systems*, vol. 3, pp. 143–158, 1996.

[90] M. Shaw, "Larger scale systems require higher-level abstractions," *Software Engineering Notes*, vol. 14, no. 3, pp. 143–146, Apr. 1989.

[91] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

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

[93] D. Silva, R. Terra, and M. T. Valente, "Recommending automated extract method refactorings," in *International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 146–156. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597141

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

[95] F. Simon, F. Steinbr, and C. Lewerentz, "Metrics based refactoring," in *European Conference on Software Maintenance and Reengineering*. IEEE, 2001, pp. 30–38.

[96] C. Simons, J. Singer, and D. R. White, "Search-based refactoring: Metrics are not enough," in *International Symposium on Search-Based Software Engineering*, M. Barros and Y. Labiche, Eds. Springer, 2015, pp. 47–61.

[97] 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 Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.

[98] 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.

[99] 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*. ACM, 1999, pp. 47–56.

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

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

[102] 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 *International Conference on Software Engineering*. IEEE, 2015, pp. 403–414.

[103] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Working Conference on Reverse Engineering*. IEEE, 2002.

[104] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Journal of Automated Software Engineering*, vol. 23, no. 3, pp. 501–532, Sep. 2016. [Online]. Available: http://dx.doi.org/10.1007/s10515-014-0175-x

[105] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatic segmentation of method code into meaningful blocks to improve readability," in *Working Conference on Reverse Engineering*. IEEE, 2011, pp. 35–44.

[106] Y. Wang, "What motivate software engineers to refactor source code? evidences from professional developers," in *International Conference on Software Maintenance*, 2009, pp. 413 –416.

[107] B. F. Webster, *Pitfalls of Object Oriented Development*. M & T Books, February 1995.

[108] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: recovering links between bugs and changes," in *Symposium on the Foundations of Software Engineering*. ACM, 2011, pp. 15–25.

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

[110] 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 *International Conference on Software Maintenance and Evolution*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 121–130. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332458

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

[112] ——, "Do developers care about code smells? an exploratory survey," in *Working Conference on Reverse Engineering*. IEEE, 2013, pp. 242–251.