Noname manuscript No.
(will be inserted by the editor)

Investigating the Relation between Lexical Smells and Change- and Fault- Proneness: An Empirical Study

Latifa Guerrouj · Zeinab Kermansaravi · Venera Arnaoudouva · Benjamin C. M. Fung · Foutse Khomh · Giuliano Antoniol · Yann-Gaël Guéhéneuc

Received: date / Accepted: date

Abstract Past and recent studies have shown that design smells which are poor solutions to recurrent design problems make object-oriented systems difficult to maintain, and that they negatively impact the class change- and fault-proneness. More recently, lexical smells have been introduced to capture recurring poor practices in the naming, documentation, and choice of identifiers during the implementation of an entity. Although recent studies show that developers perceive lexical smells as impairing program understanding, no study has actually evaluated the relationship between lexical smells and software quality as well as their interaction with design smells.

Latifa Guerrouj
École de Technologie Supérieure, Montréal, Canada
E-mail: Latifa.Guerrouj@etsmtl.ca

Zeinab Kermansaravi
École Polytechnique de Montréal, Canada
E-mail: Zeinab.Kermansaravi@polymtl.ca

Venera Arnaoudouva
Washington State University, USA
E-mail: Venera.Arnaoudova@wsu.edu

Benjamin C. M. Fung
McGill University, Canada
E-mail: Ben.Fung@mcgill.ca

Foutse Khomh
École Polytechnique de Montréal, Canada
E-mail: Foutse.Khomh@polymtl.ca

Giuliano Antoniol
École Polytechnique de Montréal, Canada
E-mail: Giuliano.Antoniol@polymtl.ca

Yann-Gaël Guéhéneuc
École Polytechnique de Montréal, Canada
E-mail: Yann-gael.Gueheneuc@polymtl.ca

In this paper, we detect 29 smells consisting of 13 design smells and 16 lexical smells in 30 releases of three projects: ANT, ArgoUML and Hibernate. We analyze to what extent classes containing lexical smells have higher (or lower) odds to change or to be subject to fault-fixing than other classes containing design smells.

Our results show and bring empirical evidence on the fact that lexical smells can make, in some cases, classes with design smells more fault-prone. In addition, we empirically demonstrate that classes containing design smells only are more change and fault-prone than classes with lexical smells only.

1 Introduction

Design smells are bad practices in software development; they represent "poor" design or implementation solutions to recurring design problems [1,2]. Most often developers introduce design smells when they are not knowledgable enough about a system, they do not have the needed expertise to solve the problem at hand, or they do not understand the logic behind how it works. Design smells do not usually prevent a program from functioning normally. However, their presence reveals the existence of flaws in the system's design or implementation. Previous studies indicate that design smells may affect software comprehensibility [3] and possibly increase change and fault-proneness [4,5]. Ehsan *et al.* [6] have found that design smells can be used to predict faults as files that have design smells tend to have a higher density of faults than other files. A recent investigation by Yamashita and Moonen has shown that the majority of developers are concerned about design smells [7]. Recently, researchers have introduced another family of smells called, lexical smells [8]. Lexical smells are defined as recurring poor practices in the naming, documentation, and choice of source code identifiers in the implementation of an entity. Their introduction was motivated by the role played by source code lexicon in capturing and encoding developers' intent and knowledge [9,10] as well as in source code understandability [11,12]. Lexical smells have been shown to enhance fault prediction when used along with traditional structural metrics [13]. In addition, they have been proven to negatively impact concept location [14]. Recently, researchers have studied how developers perceive them [15].

Although some previous works have investigated the relation between the occurrence of design smells and a class change- and fault-proneness, to the best of our knowledge we are the first to investigate the additional impact lexical smells can have on class change- and fault-proneness when occurring with design smells. Specifically, we compare, in terms of change- and fault-proneness between 1) classes with both design and lexical smells and classes with design smells only, 2) classes containing both design and lexical smells and classes with lexical smells only, as well as 3) classes with design smells only and those with lexical smells only. As baseline, we use design smells since they have been already proven to correlate with changes and faults [5].

This work is also the first to detect such a variety and large number of smells. We explore 29 smells consisting of 13 design smells and 16 lexical ones that we identified using widely-adopted techniques from the literature [8,16]. Our investigation focus on 13 design smells from Brown *et al.* [1] and Fowler [2]. We chose these design smells because they are representative of design and implementation problems related to object-oriented systems. In addition, they have been thoroughly described and received significant attention from researchers (*e.g.*, [1,5,6]). As for lexical smells, we selected the family described in [15] since its represents the most recent catalog of lexical smells, we detected them using the most recent approach for identifying lexical smells [8].

We empirically show through the analysis of 30 releases from three different projects that, in many cases, the occurrence of lexical smells can make classes with design smells more fault-prone. In addition, classes with both lexical and design smells are more change- and fault- prone than classes containing lexical smells only. Furthermore, classes with design smells only are more change and fault-prone than classes containing lexical smells only. We believe that such findings bring more awareness to developers about the additional role that lexical smells can have on fault-proneness when they occur with design smells. A software manager could use our design and lexical smells detection approaches applied in this work to assess the volume of classes with such families of smells to possibly better estimate the effort needed for refactoring.

Paper organization. The rest of the paper is organized as follows. Section 2 describes the methodology followed while Section 3 reports our empirical study. In Section 4, we show the findings of our study. Section 5 discusses the threats to validity. Section 6 presents related work. Finally, Section 7 concludes and outlines directions for future work.

2 Methodology

This section describes the methodology followed and summarized in Fig. 1. It consists of (i) mining data repositories, (ii) detecting design and (iii) lexical smells across different releases of the studied systems, as well as (iv) identifying changes and post-release defects.

2.1 Step 1: Data Collection and Processing

The first phase of our methodology consists of mining data repositories. We analyze a total of 30 releases from different open-source systems (*i.e.*, ArgoUML, ANT, and Hibernate). We selected these projects since they are made publicly available to the research community and practitioners, they have a considerable number of releases, committers, as well as development history. Our study includes 12 releases of ArgoUML, 11 releases of Hibernate, and 7 releases of

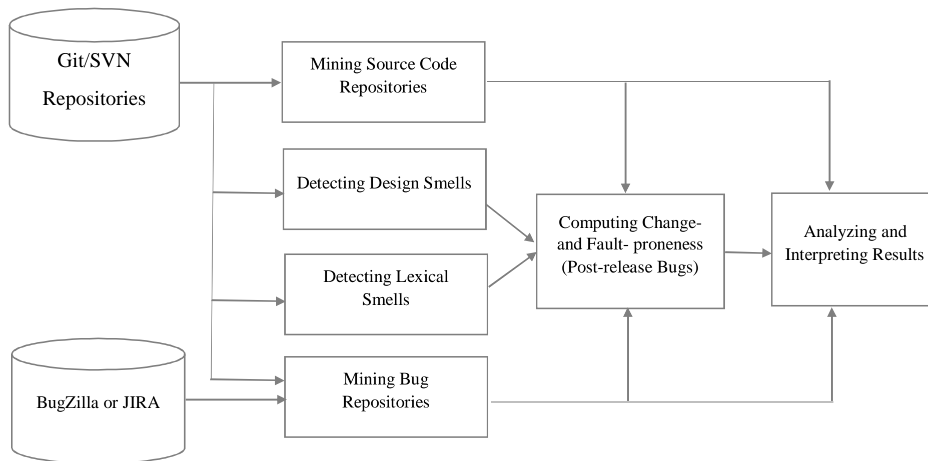

Fig. 1 Main steps of the followed methodology.

ANT. ArgoUML[1] is an open-source UML modeling tool. Hibernate[2] (ORM) is an open-source Java persistence framework project while ANT[3] is a system related to software build processes. We chose these systems because they belong to different domains and have different sizes.

The first step of our data collection process consists of downloading the source code of the considered releases for all systems, which we used as an input for the design and lexical smells detection approaches. We then mined the source code change history repositories from the version control systems of the systems, *i.e.*, Git[4] for ANT and Hibernate and SVN for ArgoUML, to identify changes and fault-fixes. The Git/SVN repository of each system was downloaded using appropriate perl scripts and the data was then stored in a PostgreSQL database. We used SQL queries to obtain the source code change history of each system release as well as information including the number of changes, classes that underwent changes, summary of the changes, change logs, etc.

In the last step, we mined bug repositories corresponding to each system with the purpose of identifying changes that were fixing faults. For ArgoUML, issues dealing with fixing faults are marked as "DEFECT" in the issue tracking system[5]. For ANT, we mined BugZilla[6] while JIRA[7] was mined to determine fault-fixing issues for Hibernate. Section 2.4 describes the steps of this phase in details.

[1] http://argouml.tigris.org/
[2] *http://hibernate.org/*
[3] *http://ant.apache.org/*
[4] *http://git-scm.com/*
[5] *http://argouml.tigris.org/issues*
[6] *https://www.bugzilla.org/*
[7] *https://www.atlassian.com/software/jira*

Finally, we use statistical tests to analyze the collected data and address our research questions.

2.2 Step 2: Identifying Design Smells

Code smells/antipatterns are "poor" implementation and–or design choices, thought to make object-oriented systems hard to maintain. In practice, code smells may concern the design of a class and hence concretely manifest themselves in the source code as classes with specific implementation. We call such smells *Design Smells*.

To identify design smells in each release of the studied projects. We use the DECOR (Defect dEtection for CORrection) approach [16]. DECOR is based on a thorough domain analysis of code and design smells from the literature, from which is built a domain-specific language. This language uses rules to describes design smells, with different types of properties: lexical (*e.g.*, class names), structural (*e.g.*, classes declaring public static variables), internal (*e.g.*, number of methods), and the relation among properties (*e.g.*, association, aggregation, and composition relations among classes). Using this domain-specific language, DECOR proposes the descriptions of several design smells. It also provides algorithms and a framework, DeTeX, to convert design smell descriptions automatically into detection algorithms. DeTeX allows detecting occurrences of design smells in systems written in various object-oriented programming languages, such as Java or C++. We used DECOR because it has been widely-acknowledged and used in past and recent research [5,4,3]; it achieves 100 percent of recall and a precision greater than 31% in the worst case, with an average greater than 60%. More precisely, DECOR yields 100% of recall and have precisions between 41.1 and 87% for three types: Blob, SpaghettiCode, and SwissArmyKnife. The detection algorithms for these three types have an average accuracy of 99% for the Blob, of 89% for the SpaghettiCode, and of 95% for the SwissArmyKnife; and a total average of 94% [16].

In this study, we focus on 13 design smells from Brown *et al.* [1] and Fowler *et al.* [2]. The motivation behind our choice is that these design smells have been thoroughly described and that they have received significant attention from researchers [1,5,6]. We could detect several occurrences of these design smells across the studied releases, and they are representative of design and implementation problems related to object-oriented systems.

- **AntiSingleton**: A class that provides mutable class variables, which consequently could be used as global variables.
- **Blob**: A class that is too large and not cohesive enough, that monopolises most of the processing, takes most of the decisions, and is associated to data classes.
- **ClassDataShouldBePrivate**: A class that exposes its fields, thus violating the principle of encapsulation.

– **ComplexClass**: A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs.
– **LargeClass**: A class that has (at least) one long method.
– **LazyClass**: A class that has few fields and methods (with little complexity).
– **LongMethod**: A class that has a method that is overly long, in term of LOCs.
– **LongParameterList**: A class that has (at least) one method with a too long list of parameters with respect to the average number of parameters per methods in the system.
– **MessageChain**: A class that uses a long chain of method invocations to realise (at least) one of its functionality.
– **RefusedParentBequest**: A class that redefines inherited methods using empty bodies, thus breaking polymorphism.
– **SpaghettiCode**: A class declaring long methods with no parameters and using global variables. These methods interact too much using complex decision algorithms. This class does not exploit and prevents the use of polymorphism and inheritance.
– **SpeculativeGenerality**: A class that is defined as abstract but that has very few children, which do not make use of its methods.
– **SwissArmyKnife**: A class whose methods can be divided in disjunct set of many methods, thus providing many different unrelated functionalities.

2.3 Step 3: Identifying Lexical Smells

The third phase of our methodology consists of identifying lexical smells in each release of the studied projects. We detect lexical smells at the level of each system's release using the LAPD (Lexical Anti-Patterns Detection) approach presented and described in [8] for Java source code; it relies on the Stanford natural language parser [17] to identify the Part-of-Speech of the terms constituting the identifiers and comments and to establish relations between those terms. We used LAPD because, to the best of our knowledge, it is the most recent novel approach that deals with large number of lexical smells; it has a catalog of 16 lexical smells. The rationale and specifications of these lexical smells are detailed in [8]. In the following, we list the lexical smells detected by LAPD and used in this work.

– **"Get" - more than an accessor**: A getter that performs actions other than returning the corresponding attribute without documenting it.
– **"Is" returns more than a Boolean**: The name of a method is a predicate suggesting a true/false value in return. However, the return type is not Boolean but rather a more complex type allowing, thus, a wider range of values without documenting them.
– **"Set" method returns**: A set method having a return type different than void and not documenting the return type/values with an appropriate comment.

- **Expecting but not getting a single instance**: The name of a method indicates that a single object is returned but the return type is a collection.
- **Validation method does not confirm**: A validation method (*e.g.*, name starting with "validate", "check", "ensure") does not confirm the validation, *i.e.*, the method neither provides a return value informing whether the validation was successful, nor documents how to proceed to understand.
- **"Get" method does not return**: The name suggests that the method returns something (*e.g.*, name starts with "get" or "return") but the return type is void. The documentation should explain where the resulting data is stored and how to obtain it.
- **Not answered question**: The name of a method is in the form of predicate whereas the return type is not Boolean.
- **Transform method does not return**: The name of a method suggests the transformation of an object but there is no return value and it is not clear from the documentation where the result is stored.
- **Expecting but not getting a collection**: The name of a method suggests that a collection should be returned but a single object or nothing is returned.
- **Method name and return type are opposite**: The intent of the method suggested by its name is in contradiction with what it returns.
- **Method signature and comment are opposite**: The documentation of a method is in contradiction with its declaration.
- **Says one but contains many**: The name of an attribute suggests a single instance, while its type suggests that the attribute stores a collection of objects.
- **Name suggests Boolean but type does not**: The name of an attribute suggests that its value is true or false, but its declaring type is not Boolean.
- **Says many but contains one**: The name of an attribute suggests multiple instances, but its type suggests a single one. Documenting such inconsistencies avoids additional comprehension effort to understand the purpose of the attribute.
- **Attribute name and type are opposite**: The name of an attribute is in contradiction with its type as they contain antonyms. The use of antonyms can induce wrong assumptions.
- **Attribute signature and comment are opposite**: The declaration of an attribute is in contradiction with its documentation. Whether the pattern is included or excluded is, thus, unclear.

2.4 Step 4: Identifying Post-Release Defects

The fourth phase of our methodology consists of identifying post-release defects. To determine whether a change fixes a fault, we search, using regular expressions, in change logs from the system versioning Git/SVN for co-occurrences of fault identifiers with keywords like "fixed issue #ID", "bug ID", "fix", "defect", or "patch". A similar approach was applied to identify

fault-fixing and fault-inducing changes in prior works [18,19]. Following current practices on the identification of post-release defects [18,20,21], we define post-release faults as those with fixes recorded in the six-month period after the release date. Once this step is performed, we identify, for each bug ID, the corresponding bug report from the corresponding issue tracking system, *i.e.*, Bugzilla[8] or Jira[9] and extract relevant information from each report including:

- Issue ID.
- Issue type, *i.e.*, fault, enhancement, feature, patch, feature request, etc.
- Issue status, *i.e.*, new, closed, reponed, resolved, fixed, verified, or not.
- Issue resolution, *e.g.*, fixed, invalid, duplicate, etc.

We extracted further information about bugs such as the priority of the bug, its opening and closing dates, as well as the bug summary. We did not leverage them in this investigation but kept them in our database for possible further investigations.

We first make sure that the issues correspond to the system (*i.e.*, product) under analysis since some communities (*e.g.*, Apache) use the same issue tracking system for multiple products. Second, we verify whether the issue IDs identified at the level of commits from the Git/SVN versioning system are true positives. Then, we differentiate fault fixes from other types of issues involving enhancements, feature requests, etc. based on the issue type, status, and resolution. As in prior works [18,19,22], we search faults characterized by "CLOSED" status and "FIXED" resolution. In such a way, fault fixes are used as measure of fault-proneness and invalid or duplicate issues are excluded. Our pipeline for the extraction of bug data mirrors the methodology followed by recent studies on smells [4] as well as studies conducted in other contexts (*e.g.*, code review, refactoring, quality assurance, etc.) [18,19,22,20,21].

2.5 Step 5: Identifying Defect-Inducing Changes

To make sure a fault was in the specific release, we have applied the widely-applied SZZ [23] algorithm. This algorithm links each defect fix to the source code change that introduced the original defect relying on information from Version control systems (*i.e.*, Git or SVN) and Issue Tracking Systems (*e.g.*, BugZilla or JIRA). The SZZ algorithm consists of three main steps. The first stage consists of identifying defect-fixes changes. SZZ searches, in change comments, for keywords such as "fixed issue #ID", "bug ID", "fix", "defect", "patch", "crash", "freeze", "breaks", "wrong", "glitch", "proper". The second step verifies if that change is really a defect fixing change using information from Issue tracking systems. For such a purpose, we search for the defect identification numbers mentioned in the change logs in the BugZilla or JIRA Issue Tracking Systems. The third step determines when the defect is introduced.

[8] https://www.bugzilla.org/
[9] https://www.atlassian.com/software/jira

We first use the *diff* command to locate the lines that were changed by the defect fix. Then, we use the *annotate* and *blame* commands to trace back to the last revision where the changes of lines have been made. If no defect report is specified in the fixing change, then similar to prior work [18], we assume that the last change before the fixing change was the change that introduced the defect [23, 18].

3 Study Description

In this section, we present the empirical study that we have performed to validate our research questions.

Table 1 Characteristics of the Analyzed Projects.

Projects	#Rel.	#Dev.	#Size (LOCs)	#All Classes	#Changes	#Classes Changed	#Faulty Changes
ANT	7	51	1,660,256	14,067	15,353	64,167	587
ArgoUML	13	25	644,829	27,822	5,300	23,153	201
Hibernate	10	89	7,239,075	21,876	9,075	89,658	179

The **goal** of this study is to investigate the relationship between design and lexical smells occurring on classes in object-oriented systems and software quality by analyzing the relation between the presence of smells from the two families and the change- and fault-proneness of classes.

The **purpose** is to show to what extent classes with lexical smells have higher odds to change or to be subject to fault-fixing changes than classes containing design smells or classes with no smell.

The **quality focus** is the change- and fault-proneness of classes in object oriented systems.

The **perspective** is that of researchers and practitioners interested in understanding the relation between the occurrence of lexical and/or design smells and a class change- and fault-proneness, which can be beneficial for quality assurance teams when prioritizing for example change- and fault-prone classes for testing.

The **context** consists of three open-source projects: ArgoUML, ANT, Hibernate. We analyze a total number of 30 releases: 12 releases for ArgoUML, 11 releases for Hibernate, and 7 releases for ANT. Table 1 summarizes the main characteristics of the analyzed systems including the number of releases, size, total number of classes for each system, number of developers, total number of changes, total number of changed classes, number of fault-fixing changes.

In terms of smells detected, Table 2 indicates for each project release, the number of design smells as well as lexical ones. Additionally, it shows the percentage (in parentheses) of classes with such families of smells with respect to the total number of classes. For example, the cell at the intersection of the ANT 151 release row and design smells column reports that the total

number of design smells detected in the release ANT 151 is 545 and that the percentage of classes containing these smells is 21.25 (*i.e.*, 384 out of a total of 1807 classes). We also report the percentage of classes with both design and lexical smells.

3.1 Research Questions

The study reported in this section aims at addressing the following research questions:

- **RQ1: Are classes with a particular family of smells (design, lexical, or both design and lexical) more change-prone than others?** Specifically, we test the following null hypothesis:
  H_{0_1}: The proportion of classes undergoing at least one change between two releases is not different between classes containing different families of smells.
- **RQ2: Are classes with a particular family of smells (design, lexical, or both design and lexical) more fault-prone than others?** Specifically, we test the following null hypothesis:
  H_{0_2}: The proportion of classes undergoing at least one fault-fixing change between two releases does not differ between classes with different families of smells.

H_{0_1} and H_{0_2} are two-tailed because we are interested in investigating whether a family of smells relate to an increase or decrease of change-proneness and fault-proneness.

3.2 Variables Selection

- **Independent variables**: number of classes containing the 29 smells where 13 of them are design smells and 16 are lexical ones. In our computations, we use variables $S_{i,j,k}$ which indicate the number of times that a class i has a design, lexical, or both design and lexical smells j in a release k. We aggregate these variables into a Boolean variable $S_{i,k}$ indicating if a class i has or not in any smells.
- **Dependent variables**: measure the phenomena related to classes with different families of smells:
  1. **Change-proneness**: refers to whether a class underwent at least a change between release k (in which it has some smells) and the subsequent release $k+1$. Changes are identified, for each class in a system, by looking at commits in their control-version systems (Git or SVN). For the sake of simplicity, we assumed to have one class per file, *i.e.*, as in prior works [5], we do not consider inner classes and non public top-level.

Table 2 Number of Design and Lexical smells in each release (in parentheses, the percentages of classes (Number of classes with smells/Total number of classes)).

ANT				ArgoUML				Hibernate			
Rel.	#Design	#Lexical	#Both	Rel.	#Design	#Lexical	#Both	Rel.	#Design	#Lexical	#Both
151	545 (384/1807)	95 (71/1807)	27 (27/1807)	0.14	1705 (831/1844)	109 (84/1844)	60 (24/1844)	3.6.1	1207 (737/1931)	893 (511/1931)	102 (100/1931)
152	551 (387/1824)	96 (70/1824)	29 (29/1824)	0.16	1650 (829/1829)	116 (89/1829)	70 (26/1829)	3.6.2	1182 (718/1932)	905 (516/1932)	94 (77/1932)
154	445 (300/1500)	72 (53/1500)	26 (26/1500)	0.18.1	2994 (1582/1882)	173 (134/1882)	85 (41/1882)	3.6.3	1200 (725/1933)	909 (518/1933)	0 (0)
170	685 (476/526)	263 (169/526)	90 (42/526)	0.20	1504 (772/1854)	190 (144/1854)	73 (43/1854)	3.6.4	1190 (723/1933)	912 (520/1933)	0 (0)
180	755 (539/2353)	268 (174/2353)	98 (93/2353)	0.22	1608 (818/1919)	189 (147/1919)	87 (41/1919)	3.6.6	1226 (769/1933)	919 (524/1933)	0 (0)
192	707 (489/2417)	280 (180/2417)	97 (83/2417)	0.24	1731 (933/1976)	192 (148/1976)	120 (45/1976)	3.6.7	3432 (1015/1933)	919 (524/1933)	0 (0)
15	551 (381/1839)	100 (76/1839)	27 (23/1839)	0.26	1691 (974/2216)	255 (193/2216)	122 (48/2216)	3.6.8	1661 (977/1935)	919 (528/1935)	0 (0)
				0.26.2	1634 (957/2216)	254 (192/2216)	108 (42/2216)	3.6.9	1850 (1107/1935)	919 (528/1935)	0 (0)
				0.28	1630 (931/2283)	364 (288/2283)	92 (50/2283)	4.2.5	2217 (1303/2654)	2217 (1303/2654)	0 (0)
				0.28.1	1677 (962/2283)	364 (288/2283)	32 (32/2283)	4.2.7	1109 (652/2665)	1109 (652/2665)	0 (0)
				0.30	1655 (954/2271)	343 (281/2271)	119 (38/2271)	4.3.0	1207 (719/3027)	1207 (719/3027)	0 (0)
				0.30.1	1692 (990/2271)	345 (282/2271)	126 (30/2271)				

2. **Fault-proneness**: refers to whether a class underwent at least a fault fixing change between releases k and $k+1$. We identified fault fixing changes following the methodology described in Section 2.4 based on the traceability of faults/issues to changes by matching their IDs in the commits [24] and the issue tracking systems.

3.3 Analysis Method

We study whether changes and faults in a class are related to the class containing a specific family of smells (*e.g.*, lexical or design smells) regardless of the kinds of smells from each family (*e.g.*, Blob or LazyClass design smells). More precisely, we test whether the proportions of classes exhibiting (or not) at least one change/fault significantly vary between classes with 1) design smells, 2) lexical smells, or 3) both design and lexical smells. Our analysis methods and statistical procedures applied in this study mirror the ones followed in previous studies about smells (*e.g.*, [4]).

To address **RQ1**, we compute the following:

1. **#Design**: number of classes of a project release for which there was at least one class change and at least one design smell among the 13 design smells detected.
2. **#Lexical**: number of classes of a project release for which there was at least one class change and at least one lexical smell among the 16 design smells detected.
3. **#Design-Lexical**: number of classes of a project release for which there was at least one class change and at least a design and a lexical smell (both) among the 29 design and lexical smells detected.
4. **#No-Design**: number of classes of a project release for which there was no design smell, while there was at least one class change.
5. **#No-Lexical**: number of classes of a project release for which there was no lexical smell, while there was at least one class change.
6. **#No-Design-Lexical**: number of classes of a project release for which there was no design and lexical smells at the same time, while there was at least one class change.

Then, we use the Fisher exact test [25] to assess whether the proportion between different families of smells significantly differs in terms of changes/faults. Specifically, we first test the statistical difference between the proportions of design and lexical smells (*i.e.*, (1,4) and (2,5)) in terms of changes/faults. Then, we test wether the difference between the proportions of design and lexical smells and design smells (*i.e.*, (3,6) and (1,4)) is statistically significant. Finally, we investigate the statistical difference, in terms of change- and fault-proneness, between the proportions of design and lexical smells and lexical smells (*i.e.*, (3,6) and (2,5)).

As for **RQ2**, we compute the same proportions above, for the different considered families of smells, but for faults (instead of changes) and then we

Table 3 Change-Proneness Results: Design and Lexical Smells vs. Design Smells (only).

Design and Lexical vs. Design Smells						
Release	#Design-Lexical	#Design	#No-Design-Lexical	#No-Design	Adj. p-value	OR
ANT 151	27	266	0	119	**<0.0001**	-
ANT 152	29	269	0	119	**<0.0001**	-
ANT 154	26	244	0	57	**0.012**	-
ANT 170	42	146	48	331	**0.0047**	1.98
ANT 180	93	357	5	183	**<0.0001**	9.51
ANT 192	83	292	14	198	**<0.0001**	4.01
ANT 15(MAIN)	23	162	4	220	**<0.0001**	7.77
Hibernate 3.6.1	100	736	2	22	1	1.49
Hibernate 3.6.2	77	589	17	149	0.68	1.14
Hibernate 3.6.3	0	538	0	181	1	0
Hibernate 3.6.4	0	452	0	274	1	0
Hibernate 3.6.7	0	304	0	420	1	0
Hibernate 3.6.8	0	315	0	455	1	0
Hibernate 4.2.5	0	512	0	504	1	0
Hibernate 4.2.7	0	492	0	486	1	0
Hibernate 4.3.0	0	469	0	639	1	0
ArgoUML 0.14	24	365	36	471	0.68	0.86
ArgoUML 0.16	26	397	44	437	0.10	0.65
ArgoUML 0.18	41	514	44	1077	**0.003**	1.95
ArgoUML 0.18.1	43	576	30	201	**0.0083**	0.50
ArgoUML 0.20	41	459	46	364	0.14	0.70
ArgoUML 0.22	45	653	75	285	**<0.0001**	0.26
ArgoUML 0.24	48	496	74	483	0.02	0.63
ArgoUML 0.26	42	435	66	525	0.22	0.76
ArgoUML 0.26.2	50	606	42	328	0.052	0.64
ArgoUML 0.28	96	374	64	591	0.232	0.79
ArgoUML 0.28.1	38	540	81	418	**<0.0001**	0.36
ArgoUML 0.30	241	370	965	595	**<0.0015**	0.50
ArgoUML 0.30.1	231	520	88	445	**<0.0001**	0.36

assess whether the differences between the computed proportions significantly differs in terms of faults.

We also use the Odds Ratio (OR) [25] as an effect size measure. Odds ratio indicates the likelihood of an event (*i.e.*, change or fault) to occur. The OR is defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the set of classes with one family of smells or both, *i.e.*, lexical, design, or lexical and design smells (experimental group), to the odds q of it occurring in the other sample, *i.e.*, the set of classes containing another different family of smells from the three investigated families, *i.e.*, lexical, design, or lexical and design smells (control group): $OR = \frac{p/(1-p)}{q/(1-q)}$.

The interpretation of odds ratio is as follow. An odds ratio of 1 indicates that the event (*i.e.*, change or fault) is equally likely in both samples. OR $>$ 1 indicates that the event is more likely in the first sample (experimental group) while an OR $<$ 1 shows the opposite (control group).

Since we perform several tests on the same data, we adjust p-values using the Bonferroni correction procedure [25]. This procedure works as follow: it divides the critical p-value ($alpha$) by the number of comparisons, n, being made: $alpha/n$. In this study, we perform three pair of tests (*e.g.*, design vs. lexical smells) when analyzing change/fault proneness, the null hypothesis is, therefore, rejected only if the p-value is less than 0.016 (0.05/3). We use Bonferroni because it is a simple procedure [25].

Table 4 Change-Proneness Results: Design and Lexical Smells vs. Lexical Smells (only).

Design and Lexical vs. Lexical Smells						
Release	#Design-Lexical	#Lexical	#No-Design-Lexical	#No-Lexical	Adj. *p*-value	*OR*
ANT 151	27	58	0	13	**0.01**	-
ANT 152	29	57	0	13	**0.0093**	-
ANT 154	26	51	0	2	1	-
ANT 170	42	59	48	110	0.08	1.62
ANT 180	93	157	5	17	0.24	2.00
ANT 192	83	129	14	51	**0.011**	2.33
ANT 15(MAIN)	23	38	4	38	**0.0013**	5.66
Hibernate 3.6.1	100	157	2	354	**<0.0001**	112.42
Hibernate 3.6.2	77	131	17	385	**<0.0001**	13.24
Hibernate 3.6.3	0	209	0	309	1	0
Hibernate 3.6.4	0	208	0	312	1	0
Hibernate 3.6.7	0	63	0	461	1	0
Hibernate 3.6.8	0	60	0	468	1	0
Hibernate 4.2.5	0	29	0	1274	1	0
Hibernate 4.2.7	0	24	0	628	1	0
Hibernate 4.3.0	0	59	0	660	1	0
ArgoUML 0.14	24	26	36	58	0.28	1.48
ArgoUML 0.16	26	30	44	59	0.73	1.16
ArgoUML 0.18	41	50	44	84	0.12	1.56
ArgoUML 0.18.1	43	53	30	91	**0.0023**	2.45
ArgoUML 0.20	41	43	46	104	**0.0073**	2.14
ArgoUML 0.22	45	53	75	95	0.79	1.075
ArgoUML 0.24	48	62	74	131	0.22	1.36
ArgoUML 0.26	42	54	66	138	0.07	1.52
ArgoUML 0.26.2	50	69	42	219	**<0.0001**	3.76
ArgoUML 0.28	32	44	64	244	**0.00031**	2.76
ArgoUML 0.28.1	38	53	81	228	**0.0059**	2.01
ArgoUML 0.30	30	41	96	241	0.033	1.83
ArgoUML 0.30.1	38	51	88	231	**0.0091**	1.95

4 Results and Discussion

We now present and discuss the results of the empirical study that we conducted to answer the research questions formulated in Section 3.

4.1 RQ1. Are classes with a particular family of smells more change-prone than others?

1. Classes containing both design and lexical smells vs. classes with design smells

Table 3 summarizes the obtained odds ratios.

For ANT, in all analyzed releases, Fisher's exact test indicates a significant difference in the proportion of changed classes between the group of classes containing in both design and lexical smells and those having design smells only. Odds ratios vary across systems and, within each system, across releases. For ANT, we found an *OR* greater than 1 in all releases. The *OR* ranges from 1.98 (ANT 170) to 9.51 (ANT 180). This finding means, that for ANT, classes with both design and lexical smells are more change-prone than classes containing design smells only.

For ArgoUML, in 6 releases (out of a total of 13), Fisher's exact test indicates a significant difference in the proportion of changed classes between the group of classes with both design and lexical smells and those containing design smells only. Odds ratios vary across systems and, within each system, across releases. We find an OR greater than 1 for the release 0.18; this indicates that classes with both design and lexical smells are more change-prone than classes with design smells only. For the rest of releases, the OR is less than 1 and in few cases close to 1, *i.e.*, the odd of experiencing a change is the same for classes with both lexical and design smells and classes with design smells only.

For Hibernate, we did not find any significant differences.

Overall, we could not find that classes having both design and lexical smells are more change-prone than classes containing design smells only across all systems and releases. We therefore conclude that lexical smells do not increase the odds of a class to experience a change, *i.e.*, they do not make classes with design smells more change-prone:

> This finding brings empirical evidence on the fact that lexical smells do not contribute to the change-proneness of design smells when both occur in classes of object-oriented systems.

2. Classes having design and lexical smells vs. classes containing lexical smells

Table 4 shows the difference in proportions between the change-proneness of classes with both design and lexical smells and classes with lexical smells only. As it can be noticed, results vary depending on the system. However, in several cases, Fisher's exact test show significant differences with OR greater than 1. For ANT, the OR ranges between 2.33 (ANT 192) and 5.66 (ANT 15 MAIN) while for Hibernate the OR is higher, it is between 13.24 (Hibernate 3.6.1) and 112.42 (Hibernate 3.6.2) which means that the difference, in terms of changes, is really high. For ArgoUML, the OR is between 1.95 (ArgoUML 0.30.1) and 3.76 (ArgoUML 0.26.2); theses results suggest that, in most cases, the odd of experiencing a change is higher for classes with both design and lexical smells than it is for classes with lexical smells only. We therefore conclude that classes with both design and lexical smells are changed in greater proportion than classes having only in lexical smells. When comparing the Odd ratios of (Design and Lexical vs. Design Smells) and (Design and Lexical vs. Lexical Smells), we observe that:

> The occurrence of design smell in a class that experienced a lexical smell seems to have a stronger relationship with change-proneness than the occurrence of lexical smell in a class that experienced a design smell.

A possible explanation to the low odds ratio for classes with lexical smells is the type of changes underwent by such classes. We manually checked the change logs of a set of such classes for the three analyzed systems and we

observed that they underwent changes mainly related to spelling, formatting (whitespace, etc.), checkstyle, imports organization, javadoc fixes, namespaces, spell checkers, as well as identifier naming. Classes with design smells underwent, in addition to such types of changes, others associated with code fixing, refactoring, design, optimization of performance, memory issues, static code analysis, etc. This is why more likely lexical smells do not boost change rates that much.

3. Classes containing design smells vs. classes with lexical smells

Table 5 reports on the proportion of changed classes in the groups of classes experiencing design smells only and classes experiencing lexical smells only.

As it can be noticed, in most of the system's releases, Fisher's exact test indicates that the difference between the change-proneness of classes with design smells only vs. classes with lexical smells only is statistically significant. Except for ANT, in which the OR is less than 1 indicating that the proportion of classes having design smells that changed, is lower than the proportion of classes with lexical smells that changed, the OR is greater than 1 for all other releases of the analyzed systems. It ranges between 2.47 (Hibernate 3.4.6) and 75.16 (Hibernate 3.6.1) for Hibernate and between 1.78 (ArgoUML 0.16) and 7.20 (ArgoUML 0.30) for ArgoUML. This findings is very likely due to the fact that design smells are, in general, well known and established than lexical smells, *i.e.*, they change more often in comparison with lexical smells as developers know they need to change them. We therefore conclude that:

> Design smells contribute more to the change-proneness of lexical smell classes than lexical smells do to the change-proneness of design smells classes.

Overall, we reject the hypothesis H_{01} since, in most of the analyzed systems, there is a significant difference between the proportion of classes undergoing at least one change between two releases, for classes belonging to different families of smells.

4.2 RQ2. Are classes having a particular family of smells more fault-prone than others?

In this section, we first present the results obtained using as a measure of fault-proneness the post-release defects. Then, we show our findings when defects are identified using SZZ.

4.2.1 Fault-Proneness using Post-Release Defects

1. Classes with design and lexical smells vs. classes having design smells

Table 6 summarises Fisher's exact test results and ORs. The differences in the proportions of classes that undergo fault-fixing changes is mostly significant for ANT with an OR greater than 1 varying from 2.18 to 2.76; indicating

Table 5 Change-Proneness Results: Design Smells vs. Lexical Smells.

Design Smells vs. Lexical Smells						
Release	#Design	#Lexical	#No-Design	#No-Lexical	Adj. p-val	OR
ANT 151	266	58	119	13	0.0328	0.50
ANT 152	269	57	119	13	0.044	0.51
ANT 154	244	51	57	2	**0.0044**	0.16
ANT 170	146	59	331	110	0.33	0.72
ANT 180	357	157	183	17	<**0.0001**	0.21
ANT 192	292	129	198	51	**0.0039**	0.58
ANT 15(MAIN)	162	38	220	38	0.25	0.73
Hibernate 3.6.1	736	157	22	354	<**0.0001**	75.16
Hibernate 3.6.2	589	131	149	385	<**0.0001**	11.58
Hibernate 3.6.3	538	209	181	309	<**0.0001**	4.38
Hibernate 3.6.4	452	208	274	312	<**0.0001**	2.47
Hibernate 3.6.7	304	63	420	461	<**0.0001**	5.28
Hibernate 3.6.8	315	60	455	468	<**0.0001**	5.39
Hibernate 4.2.5	512	29	504	1274	<**0.0001**	44.55
Hibernate 4.2.7	492	24	486	628	<**0.0001**	26.44
Hibernate 4.3.0	469	59	639	660	<**0.0001**	8.20
ArgoUML 0.14	365	26	471	58	0.027	1.72
ArgoUML 0.16	397	30	437	59	**0.0137**	1.78
ArgoUML 0.18	514	50	1077	84	0.25	0.80
ArgoUML 0.181	576	53	201	91	<**0.0001**	4.91
ArgoUML 0.20	459	43	364	104	<**0.0001**	3.04
ArgoUML 0.22	653	53	285	95	<**0.0001**	4.10
ArgoUML 0.24	496	62	483	131	<**0.0001**	2.16
ArgoUML 0.26	435	54	525	138	<**0.0001**	2.11
ArgoUML 0.262	606	69	328	219	<**0.0001**	5.85
ArgoUML 0.28	374	44	591	244	<**0.0001**	3.50
ArgoUML 0.281	540	53	418	228	<**0.0001**	5.54
ArgoUML 0.30	370	41	595	241	<**0.0001**	7.20
ArgoUML 0.30.1	520	51	445	231	<**0.0001**	5.28

that in ANT, the fault-proneness of classes with both design and lexical smells is higher than the fault-proneness of classes with design smells only. For the remaining systems, there is no statistically significant difference between the proportions of classes that underwent fault-fixing changes among the groups of classes with both design and lexical smells and classes with design smells, suggesting that in general:

> Lexical smells do not make classes with design smells more fault-prone (than they already are).

2. Classes with design and lexical smells vs. classes having lexical smells

Table 7 shows significant differences for some releases of the three systems. For three releases of ANT, OR values are greater than 1 ranging between 1.96 (ANT 170) and 3.67 (ANT 15 MAIN). For two releases of Hibernate the ORs are almost equal to 5. These findings suggest that, for the mentioned releases,

Table 6 Fault-Proneness Results: Design and Lexical Smells vs. Design (only).

Design and Lexical vs. Design Smells						
Release	#Design-Lexical	#Design	#No-Design-Lexical	#No-Design	Adj. p-value	OR
ANT 151	17	183	10	202	0.16	1.87
ANT 152	16	158	13	230	0.17	1.78
ANT 154	18	154	8	147	0.10	2.14
ANT 170	55	191	35	286	**0.00029**	2.34
ANT 180	50	174	48	366	**0.00051**	2.18
ANT 192	57	189	40	301	**0.00029**	2.86
ANT 15(MAIN)	20	194	7	188	0.02	2.76
Hibernate 3.6.1	6	28	96	730	0.278	1.62
Hibernate 3.6.2	6	29	88	709	0.271	1.66
Hibernate 3.6.3	0	28	0	691	1	0
Hibernate 3.6.4	0	33	0	693	1	0
Hibernate 3.6.7	0	33	0	698	1	0
Hibernate 3.6.8	0	32	0	692	1	0
Hibernate 4.2.5	0	32	0	738	1	0
Hibernate 4.2.7	0	43	0	973	1	0
Hibernate 4.3.0	0	41	0	937	1	0
ArgoUML 0.14	6	79	53	573	0.83	0.82
ArgoUML 0.16	5	100	55	736	0.53	0.66
ArgoUML 0.18	7	110	63	724	0.57	0.73
ArgoUML 0.18.1	13	141	72	1450	0.053	1.85
ArgoUML 0.20	12	143	60	634	0.87	0.88
ArgoUML 0.22	15	163	63	660	1	0.96
ArgoUML 0.24	12	165	93	773	0.13	0.60
ArgoUML 0.26	16	188	72	791	0.88	0.93
ArgoUML 0.26.2	17	190	61	770	0.65	1.12
ArgoUML 0.28	14	194	78	740	0.22	0.68
ArgoUML 0.28.1	13	188	78	777	0.26	0.68
ArgoUML 0.30	14	188	105	770	0.045	0.54
ArgoUML 0.30.1	15	178	111	817	0.10	0.62

the odd of experiencing a fault-fixing change is higher for classes with both design and lexical smells than for classes with lexical smells only. We therefore conclude that, in some cases:

> The occurrence of design smell in a class that experienced a lexical smell have a strong relationship with fault-proneness than the occurrence of lexical smell in a class that experienced a design smell.

This finding is likely due to the fact that fault-fixing changes related to classes with design smells cover (according to the fault-fixing change logs) a variety of types including implementation problems, features, API changes, bugs after modification, deployment, and FindBugs reported problems while lexical smells are mostly associated with formatting issues, identifier naming, data types, enumeration types, spelling, checkstyle, etc. This justifies why design smells boost faults rates that much.

3. Classes having design smells vs. classes with lexical smells

Table 8 reports on the proportion of classes that underwent fault-fixing changes in the groups of classes experiencing design smells only and classes experiencing lexical smells only. Except for ANT and ArgoUML 0.16 and 0.18, Fisher's exact test show significant differences with an OR greater than 1 in all studied cases. For Hibernate, the OR ranges between 2.75 and 8.75 while it varies between 1.78 and 7.20 for ArgoUML. This finding brings further

Table 7 Fault-Proneness Results: Design and Lexical Smells vs. Lexical Smells (only).

	Design and Lexical vs. Lexical Smells					
Release	#Design-Lexical	#Design	#No-Design-Lexical	#No-Design	Adj. p-value	OR
ANT 151	17	29	10	42	0.06	2.43
ANT 152	16	26	13	44	0.12	2.06
ANT 154	18	27	8	26	0.15	2.14
ANT 170	55	75	35	94	**0.01**	1.96
ANT 180	50	70	48	104	0.09	1.54
ANT 192	57	77	40	103	**0.01**	1.90
ANT 15(MAIN)	20	33	7	43	**0.007**	3.67
Hibernate 3.6.1	6	7	96	504	**0.011**	4.48
Hibernate 3.6.2	6	7	88	509	**0.007**	4.93
Hibernate 3.6.3	0	4	0	514	1	0
Hibernate 3.6.4	0	5	0	515	1	0
Hibernate 3.6.7	0	5	0	519	1	0
Hibernate 3.6.8	0	5	0	519	1	0
Hibernate 4.2.5	0	5	0	523	1	0
Hibernate 4.2.7	0	8	0	1295	1	0
Hibernate 4.3.0	0	8	0	644	1	0
ArgoUML 0.14	6	6	53	75	0.56	1.41
ArgoUML 0.16	5	6	55	78	1	1.18
ArgoUML 0.18	7	7	63	82	0.77	1.29
ArgoUML 0.18.1	13	17	72	117	0.68	1.24
ArgoUML 0.20	12	17	60	117	0.52	1.37
ArgoUML 0.22	15	17	63	117	0.23	1.63
ArgoUML 0.24	12	17	93	117	0.84	0.88
ArgoUML 0.26	16	17	72	117	0.33	1.52
ArgoUML 0.26.2	17	17	61	117	0.11	1.91
ArgoUML 0.28	14	19	78	269	0.017	2.53
ArgoUML 0.28.1	0.68	17	78	264	0.024	2.58
ArgoUML 0.30	14	17	105	264	0.06	2.06
ArgoUML 0.30.1	15	17	111	264	0.04	2.09

evidence to recent works [26] on the relationship between smells and faults. It suggests that:

> The occurrence of design smell in a class has a strong relationship with the class's fault-proneness than the occurrence of lexical smell.

We reject the hypothesis H_{02} since in most cases there is a significant difference between the proportion of classes undergoing at least one fault-fixing change between two releases, for classes belonging to different families of smells.

4.2.2 Fault-Proneness using SZZ

1. Classes with design and lexical smells vs. classes having design smells

Table 9 reports the results of Fisher's exact test and OR. It indicates the difference in proportions between the fault-proneness of classes with both design and lexical smells and classes with design smells only. As it can be noticed, results are mostly significant for ArgoUML. All results are statistically significant for all release except for 0.16, with an OR varying between 4.92 and 0. We also found statistically significant results for Hibernate in particular for the 3.6.1 and 3.6.2 releases with ORs equal to 2.74 and 2.48 respectively.

Table 8 Fault-Proneness Results: Design Smells vs. Lexical Smells.

	Design Smells vs. Lexical Smells					
Release	#Design	#Lexical	#No-Design	#No-Lexical	Adj. *p*-val	*OR*
ANT 151	183	29	202	42	0.36	1.31
ANT 152	158	26	230	44	0.59	1.16
ANT 154	154	27	147	26	1	1.08
ANT 170	191	75	286	94	0.36	0.83
ANT 180	174	70	366	104	0.05	0.70
ANT 192	189	77	301	103	0.32	2.84
ANT 15(MAIN)	194	33	188	43	0.25	1.34
Hibernate 3.6.1	28	7	730	504	**0.01**	2.75
Hibernate 3.6.2	29	7	709	509	**0.0089**	2.97
Hibernate 3.6.3	28	4	691	514	**0.00041**	5.20
Hibernate 3.6.4	33	5	693	515	**0.000169**	4.89
Hibernate 3.6.7	33	5	698	519	**0.00016**	4.90
Hibernate 3.6.8	32	5	692	519	**0.0002**	4.79
Hibernate 4.2.5	43	8	973	1295	**<0.0001**	7.14
Hibernate 4.2.7	41	8	937	644	**0.00052**	8.75
Hibernate 4.3.0	40	8	1068	711	**0.00084**	3.32
ArgoUML 0.14	365	26	471	58	0.027	1.72
ArgoUML 0.16	397	30	437	59	**0.0137**	1.78
ArgoUML 0.18	514	50	1077	84	0.25	0.80
ArgoUML 0.18.1	576	53	201	91	**<0.0001**	4.91
ArgoUML 0.20	459	43	364	104	**<0.0001**	3.04
ArgoUML 0.22	653	53	285	95	**<0.0001**	4.10
ArgoUML 0.24	496	62	483	131	**<0.0001**	2.16
ArgoUML 0.26	435	54	525	138	**<0.0001**	2.11
ArgoUML 0.26.2	606	69	328	219	**<0.0001**	5.85
ArgoUML 0.28	374	44	591	244	**<0.0001**	3.50
ArgoUML 0.28.1	540	53	418	228	**<0.0001**	5.54
ArgoUML 0.30	370	41	595	241	**<0.0001**	7.20
ArgoUML 0.30.1	520	51	445	231	**<0.0001**	5.28

However, we did not find any statistically significant results for ANT. Unlike the findings obtained by leveraging post-release defects, these results show that in some cases:

> The occurrence of lexical smells can make classes with design smells more fault-prone.

2. Classes with design and lexical smells vs. classes having lexical smells

Table 10 summarizes the results obtained using Fisher's exact test and *OR* for what concerns the differences in terms of the proportions of fault-proneness of classes with both design and lexical smells and classes with lexical smells only. As it can be noticed, results are all statistically significant for Hibernate, with an *OR* between 6.34 and 2.29. For ArgoUML, we found statistically significant results for three releases only, *i.e.*, 0.10.1, 0.14, and 0.12 with *ORs* equal to 3.69, 4.0, and 3.96 respectively. For ANT, only the release 170 yields statistically significant results with an *OR* equals to 2.04. These findings suggest

Table 9 Fault-Proneness Results using SZZ: Design and Lexical Smells vs. Design (only).

Design and Lexical Smells vs. Design Smells							
Project	Release	#Design	#Lexical	#No-Design	#No-Lexical	Adj. p-val	OR
Ant	151	8	92	19	292	0.49	1.34
	152	8	93	20	294	0.64	1.26
	15	8	91	19	290	0.49	1.34
	154	8	90	20	210	1.0	0.93
	192	21	108	65	381	0.67	1.14
	180	21	108	64	431	0.31	1.31
	170	20	111	54	365	0.46	1.22
ArgoUML	0.10.1	24	316	5	324	**0.000437**	4.92
	0.28	0	365	89	568	**0.0**	0.0
	0.24	0	395	73	542	**0.0**	0.0
	0.26	0	393	88	585	**0.0**	0.0
	0.20	0	407	67	369	**0.0**	0.0
	0.22	0	408	70	414	**0.0**	0.0
	0.30	0	364	88	593	**0.0**	0.0
	0.14	21	387	7	448	**0.003365**	3.47
	0.16	22	412	20	421	0.75	1.12
	0.26.2	0	360	88	599	**0.0**	0.0
	0.12	21	329	6	322	**0.00564**	3.43
	0.18.1	22	416	60	379	**<0.0001**	0.33
	0.30.1	0	358	89	636	**0.0**	0.0
	0.28.1	0	376	89	588	**0.0**	0.0
Hibernate	4.2.5	12	84	110	931	0.60	1.21
	3.6.1	11	71	43	686	**0.017058**	2.47
	3.6.2	11	69	43	668	**0.016986**	2.48
	3.6.3	8	70	43	648	0.22	1.72
	3.6.4	8	70	43	655	0.22	1.74
	3.6.6	8	69	43	661	0.14	1.78
	3.6.7	8	72	43	651	0.22	1.68
	3.6.8	8	71	43	698	0.13	1.83
	4.2.7	12	83	110	894	0.60	1.18
	4.3.0	11	91	139	1016	0.87	0.88

that, for the mentioned releases, the odd of experiencing a fault is higher for classes with both design and lexical smells than for classes with lexical smells only. We therefore confirm the results obtained using post-release defects as a measure of fault-proneness and can conclude that:

> The occurrence of design smell in a class having a lexical smell has a strong relationship with fault-proneness than the occurrence of lexical smell in a class that experienced a design smell.

3. Classes having design smells vs. classes with lexical smells

Table 11 reports on the proportion of classes that underwent fault-fixing changes in the groups of classes experiencing design smells only and classes experiencing lexical smells only. As it can be noticed, the results for Hibernate are all statistically significant, with an OR varying between 3.48 and 1.94. For ArgoUML, out of the 14 studied releases, ten show statistically significant

Table 10 Fault-Proneness Results using SZZ: Design and Lexical Smells vs. Lexical Smells (only).

Design and Lexical Smells vs. Lexical Smells							
Project	Release	#Design	#Lexical	#No-Design	#No-Lexical	Adj. *p*-val	*OR*
Ant	151	8	12	19	59	0.17	2.07
	152	8	11	20	60	0.16	2.18
	15	8	12	19	64	0.15	2.25
	154	8	11	20	42	0.58	1.53
	192	21	26	65	154	0.05	1.91
	180	21	26	64	148	0.06	1.87
	170	20	26	54	143	**0.049164**	2.04
ArgoUML	0.10.1	24	39	5	30	**0.019952**	3.69
	0.28	0	0	89	288	1.0	-
	0.24	0	0	73	148	1.0	-
	0.26	0	0	88	193	1.0	-
	0.20	0	0	67	144	1.0	-
	0.22	0	0	70	147	1.0	-
	0.30	0	0	88	281	1.0	-
	0.14	21	36	7	48	**0.004267**	4.0
	0.16	22	36	20	53	0.25	1.62
	0.26.2	0	0	88	192	1.0	-
	0.12	21	38	6	43	**0.006988**	3.96
	0.18.1	22	26	60	108	0.23	1.52
	0.30.1	0	0	89	282	1.0	-
	0.28.1	0	0	89	288	1.0	-
Hibernate	4.2.5	12	29	110	622	**0.024941**	2.34
	3.6.1	11	20	43	491	**<0.0001**	6.28
	3.6.2	11	20	43	496	**<0.0001**	6.34
	3.6.3	8	17	43	501	**0.000795**	5.48
	3.6.4	8	17	43	503	**0.000777**	5.5
	3.6.6	8	17	43	507	**0.000741**	5.55
	3.6.7	8	17	43	507	**0.000741**	5.55
	3.6.8	8	17	43	511	**0.000707**	5.59
	4.2.7	12	29	110	623	**0.024821**	2.34
	4.3.0	11	24	139	695	**0.03655**	2.29

results with an *OR* up to 4.56. Additionally, we found statistically significant results for two releases 192 and 170, with *OR*s equal to 1.68 and 1.67 respectively. While these results bring further evidence on the fact that design smells have an impact on fault-proneness and confirm our results obtained using post-release defects, they are different from the findings of a recent study by Hall *et al.* [26] who have examined the relationship between faults and five smells of Fowler (*i.e.*, Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man). In fact, the results of their empirical investigation have demonstrated that Switch Statements do not have any effect on faults, while Message Chains and Data Clumps for example increased faults in some cases and reduced them in others. Results also indicated that in cases where smells have significantly affected faults, the size of that effect was small. Overall, we conclude that:

The occurrence of design smell in a class has a strong relationship with the class's fault-proneness than the occurrence of lexical smell.

Table 11 Fault-Proneness Results using SZZ: Design Smells vs. Lexical Smells.

Design Smells vs. Lexical Smells							
Project	Release	#Design	#Lexical	#No-Design	#No-Lexical	Adj. p-val	OR
Ant	151	92	12	292	59	0.22	1.55
	152	93	11	294	60	0.12	1.73
	15	91	12	290	64	0.13	1.67
	154	90	11	210	42	0.19	1.64
	192	108	26	381	154	**0.02948**	1.68
	180	108	26	431	148	0.14	1.43
	170	111	26	365	143	**0.037109**	1.67
ArgoUML	0.10.1	316	39	324	30	0.31	0.75
	0.28	365	0	568	288	**0.0**	-
	0.24	395	0	542	148	**0.0**	-
	0.26	393	0	585	193	**0.0**	-
	0.20	407	0	369	144	**0.0**	-
	0.22	408	0	414	147	**0.0**	-
	0.30	364	0	593	281	**0.0**	-
	0.14	387	36	448	48	0.56	1.15
	0.16	412	36	421	53	0.11	1.44
	0.26.2	360	0	599	192	**0.0**	-
	0.12	329	38	322	43	0.55	1.16
	0.18.1	416	26	379	108	**0.0**	4.56
	0.30.1	358	0	636	282	**0.0**	-
	0.28.1	376	0	588	288	**0.0**	-
Hibernate	4.2.5	84	29	931	622	**0.002605**	1.94
	3.6.1	71	20	686	491	**0.000216**	2.54
	3.6.2	69	20	668	496	**0.000193**	2.56
	3.6.3	70	17	648	501	**<0.0001**	3.18
	3.6.4	70	17	655	503	**<0.0001**	3.16
	3.6.6	69	17	661	507	**<0.0001**	3.11
	3.6.7	72	17	651	507	**<0.0001**	3.3
	3.6.8	71	17	698	511	**<0.0001**	3.06
	4.2.7	83	29	894	623	**0.001353**	1.99
	4.3.0	91	24	1016	695	**<0.0001**	2.59

5 Threats to Validity

Construct Validity threats concern the relation between theory and observation. A main threat is related to the techniques used to detect design and lexical smells. We applied DECOR [16] for the identification of design smells since it has been widely used in previous studies on design smells, while we applied LADP to detect lexical smells because it is the most novel and recent approach [8]. Other possible design smells detection techniques (*e.g.*, inFusion,

JDeodorant or PMD) can be used to confirm our findings. Another threats relate to our method for detecting post-release bugs. In effect, we have used a method that is widely-applied in the literature [18, 20, 21]. Yet, We are aware that this accuracy is not perfect since it includes its authors' subjective understanding of the code smells [16]. Additionally, DECOR accuracy may have an impact on our results since we may have classified a class without smells as a class involving smells and vice-versa. In the future, we intend to apply other techniques and tools to confirm our findings [16].

Internal Validity threats deal with alternative explanations of our results. It is important to mention that we do not claim causation but we bring empirical evidence of the relationship between the presence of a particular family of smells and the occurrences of changes, and faults. Another threat is related to errors related to faut-fixing changes. We mitigated such a threat by not computing only the post-release defects but also defects using the SZZ algorithm [23].

Conclusion validity threats concern the relation between the treatment and the outcome. Proper tests were performed to statistically reject the null hypotheses. In particular, we used non-parametric tests, which do not make any assumption on the underlying distributions of the data, and, specifically, Fisher's exact test. Also, we based our conclusions not only on the presence of significant differences but also on the presence of a practically relevant difference, estimated by means of Odds ratio measures. Last, but not least, we dealt with problems related to performing multiple Fisher tests using the Bonferroni correction procedure.

Reliability validity threats concern the possibility of replicating this study. We make publicly available all information and necessary details to replicate our study. Moreover, the source code repositories and issue-tracking systems are publicly available to obtain the same data. The raw data used to compute the statistics presented in this paper is available on-line[10].

External validity threats concern the possibility of generalizing our results. We studied three systems having their corresponding control version system from where we extracted changes and fault-fixes. It is true that three projects is not a large number. However, we analyzed a large number of releases, *i.e.*, 30 releases in total. The investigated systems have different sizes and belong to different domains. Such a number of systems and releases may not be representative of all systems and thus we cannot guarantee that similar findings will be obtained when applying our approach to other open or closed source systems. Additionally, further validation on a larger set of systems from different domains is recommended to make sure our results are generalizable. Finally, we used a specific yet representative family of lexical and design smells. Different smells could be investigated in future work and could lead to different results.

[10] `http://swat.polymtl.ca/data/Replication-Package-Smells-SQJ-2015.zip`

6 Related Work

6.1 Design Smells

6.1.1 Design Smells Definition and Detection

Webster [27] was the first who wrote about smells in object-oriented development. A taxonomy of 22 code smells was introduced by Fowler *et al.* [2]. They pointed out to the fact that such smells are indicators about design or implementation issues which can be addressed using refactoring. Recently, Suryanarayana *et al.* [28] provided a catalog of 25 structural design smells that contribute to technical debt in software projects. Other works focused on the detection of smells [16]. Palomba *et al.* [29,30] suggested an approach called HIST (Historical Information for Smell deTection) to detect five different code smells, namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy, by exploiting change history information mined from versioning systems. The results indicate that HIST's precision ranges between 61% and 80%, and its recall ranges between 61% and 100%. More importantly, the results confirm that HIST is able to identify code smells that cannot be detected by approaches solely based on code analysis. More precisely, the authors analyzed HIST accuracy in two different scenarios. In the first scenario, they applied HIST on 20 different open-source projects and showed that in comparison with previous work, there is an improvement in precision while the recall was nearly the same. They concluded that HIST can detect code smells which were not detected by the competitive algorithm. The second scenario investigated to what extent developers can trust and consider the smells detected by HIST. The findings showed that more than 75% of these detected code smells are real design or implementation problems [29,30]. Recently, researchers [31] have applied machine learning algorithms to detect code smells. They investigated 16 different machine-learning algorithms on four code smells (*i.e.*, Data Class, Large Class, Feature Envy, Long Method) and 74 software systems. Their findings show that machine learning can help achieve a high accuracy (*i.e.*, > 96 %) when detecting code smells, and that only a hundred training examples are sufficient to reach at least 95% accuracy [31].

6.1.2 Design Smells and Software Evolution

Other researchers have analyzed the relation between smells and software quality. For example, Khomh *et al.* [5] have discovered the relation between smells and change- and fault- proneness. Li and Shatnawi [32] show relationships between six code smells and probability of class error in three different versions of Eclipse. Their investigation showed that classes with smells, such as God Class, God Method and Shotgun Surgery is strongly related to class error.

Hall *et al.* [26] have examined the relationship between faults and five smells of Fowler (*i.e.*, Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man) using Negative Binomial regression

models. They analyzed three open-source systems: Eclipse, ArgoUML, and Apache Commons. Their findings have shown that Switch Statements do not have any effect on faults in any of the three systems; Message Chains increased faults in two systems; Message Chains which occurred in larger files reduced faults; Data Clumps reduced faults in Apache and Eclipse but increased faults in ArgoUML; Middle Man reduced faults only in ArgoUML, and Speculative Generality reduced faults only in Eclipse. Results also indicated that in cases where smells did significantly affect faults, the size of that effect was small. While some smells do indicate fault-prone code in some circumstances their effect on faults is small. The authors concluded that arbitrary refactoring is unlikely to significantly reduce fault-proneness and in some cases may increase fault-proneness.

Cardoso *et al.* [33] have investigated co-occurrences between design patterns and bad smells on five systems such as AsoectJ, Hibernate, JHotDraw, Velocity and WebMail. The results of their study indicated co-occurrences between Command and GodClass, as well as between Template Method and Duplicated Code. They concluded that some of design pattern misuse may increase the possibility of arising of bad smells

Olbrich *et al.* [34] analyzed the evolution of two different code smells, *i.e.*, Shotgun surgery and God class over time in the development process of two software systems. They showed that components containing such code smells do not decrease over time given the fact that refactoring activities were not actively performed on these systems.

Peters *et al.* [35] studied the lifespan of five different code smells over different releases. They revealed that long-lived code smells increase over time given the low number of refactorings performed by developers on the considered systems. Such investigation confirm that code smells mostly remain in systems. Recently, Taba *et al.* have [6] suggested multiple metrics based on smells to improve fault prediction. Also, Palomba *et al.* [36] have conducted a study where developers with code entities of three systems affected and not by bad smells, and they asked them to tell whether the code have potential design problem, and if any, the nature and severity of the problem. The results of such study provide insights on the characteristics of bad smells yet unexplored in depth. Also, Yamashita and Moonen [7] have demonstrated through a user study with professional developers that the majority of developers are concerned about code smells.

6.2 Lexical Smells

6.2.1 Lexical Smells Definition and Detection

De Lucia *et al.* suggested COCONUT to verify consistency between the lexicon of high-level artifacts and of source code based on the textual similarity between the two artifacts [37].

Abebe and Tonella built an ontology to assist developers in the choice of identifiers consistent with the concepts already used in the system [38].

A more recent work proposed an approach to identify inconsistencies among identifiers, source code, and comments; this technique handles generic naming and comments issues in object-oriented programs, and specifically in the lexicon and comments of methods and attributes [15].

Tan *et al.* proposed several approaches to identify inconsistencies between code and comments. The first called, @iComment, detects lock- and call- related inconsistencies [39]. The second approach, @aComment, detects synchronization inconsistencies related to interrupt context [40]. A third approach, @tComment, automatically infers properties form Javadoc related to null values and exceptions; it performs test case generation by considering violations of the inferred properties [41].

6.2.2 Lexical Smells and Software Evolution

Abebe *et al.* [13] investigated whether using Lexicon Bad Smells (LBS) in addition to structural metrics improves fault prediction. They assessed the capability of their predictive models using i) only structural metrics, and ii) structural metrics and LBS. The results of their study conducted on three open-source systems, ArgoUML, Rhino, and Eclipse, indicate that there is an improvement in the majority of the cases.

The same authors investigated to what extent lexicon bad smells can hinder the execution of maintenance tasks. The results indicate that lexicon bad smells negatively affect concept location when using IR-based techniques [14].

We agree with the above-mentioned works that design smells are indicators about poor code quality and that lexical bad smells can hinder the execution of program understanding and maintenance tasks as well as decreasing the quality of programs. In our work, we empirically investigate the additional relationship that lexical smells can have with change- and fault-proneness.

7 Conclusion and Future Work

We provide further empirical evidence that design and lexical bad smells relate to change- and fault- proneness. Our investigation consists of the analysis of 30 releases of three different open-source systems: ArgoUML, Hibernate and ANT. We detected 29 smells in each release, *i.e.*, 13 design smells using the DECOR approach and 16 lexical smells using the novel LDAP approach. To study the relation between the detected families of smells and change- and fault- proneness, we leveraged the change history of the studied systems using information from their Git/SVN versioning systems. We also mined their bug repositories.

Interestingly, our findings show that lexical smells can make, in some cases, classes with design smells more fault-prone when both occur in classes of object-oriented systems. In addition, they indicate that, in a lot of cases, classes

containing design smells are more change- and fault-prone than classes with lexical smells. The occurrence of design smell in a class that experienced a lexical smell has a strong relationship with change- and fault-proneness than the occurrence of lexical smell in a class that experienced a design smell.

We believe such results could guide development and quality assurance teams to better focus their refactoring efforts on components with design smells (while not neglecting lexical smells) to assure good quality for their systems. As future work, we intend to conduct a user study involving professional developers both internal, *i.e.*, contributors to the development of the systems as well as external ones from industry to better understand the interaction between design and lexical smells, and identify which specific type of design smells (*e.g.*, Spaghetti Code) and lexical smells (*e.g.*, Attribute signature and comment are opposite) should be given higher priority during refactoring.

References

1. W. J. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis: Refactoring Software, Architecture and Projects in Crisis*, 1st ed. John Wiley & Sons, 1998.
2. M. Fowler, *Refactoring: Improving the Design of Existing Code.* Boston, MA, USA: Addison-Wesley, 1999.
3. A. Marwen, K. Foutse, Y. Guéhéneuc, and A. Giuliano, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension." IEEE Computer Society, 2011, pp. 181–190.
4. F. Khomh, M. D. Penta, and Y. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness." in *WCRE.* IEEE Computer Society, 2009, pp. 75–84.
5. F. Khomh, M. D. Penta, Y. 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.
6. S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns." in *ICSM.* IEEE, 2013, pp. 270–279.
7. A. F. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey." in *WCRE.* IEEE, 2013, pp. 242–251.
8. V. Arnaoudova, M. D. Penta, G. Antoniol, and Y. Guéhéneuc., "A new family of software anti-patterns: Linguistic anti-patterns," March 2013, pp. 187–196.
9. E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive strategies and looping constructs: an empirical study," *Commun. ACM*, vol. 26, no. 11, pp. 853–860, 1983.
10. A. Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *IEEE Computer*, pp. 44–55, 1995.
11. A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experiential study," *Journal of Program Languages*, vol. 4, no. 3, pp. 143–167, 1996.
12. D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.
13. A. S. Lemma, A. Venera, T. Paolo, A. Giuliano, and Y. Guéhéneuc, "Can lexicon bad smells improve fault prediction?" in *WCRE*, 2012, pp. 235–244.
14. S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code." in *SCAM.* IEEE, 2011, pp. 125–134.
15. V. Arnaoudova, M. D. Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering (EMSE)*, p. In press, 2015.

16. N. Moha, Y. Guéhéneuc, D. Laurence, and L. M. Anne-Franccoise, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 1, pp. 20–36, 2010.
17. K. Toutanova and C. D. Manning, "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger," October 2000, pp. 63–70.
18. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Software Eng.*, vol. 39, no. 6, pp. 757–773, 2013.
19. S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Software Eng.*, vol. 34, no. 2, pp. 181–196, 2008.
20. S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, 2014, pp. 192–201.
21. ——, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, 2015, to appear.
22. G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study." in *SCAM*, 2012, pp. 104–113.
23. J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.
24. M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *In Proceedings of the International Conference on Software Maintenance*, 2003, pp. 23–32.
25. D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.
26. 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, p. 33, 2014.
27. B. F. Webster, *Pitfalls of object-oriented development*. M & T, 1995.
28. G. Suryanarayana, *Refactoring for Software Design Smells: Managing Technical Debt 1st Edition*. Morgan Kaufmann, 2014.
29. F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information." in *ASE*, 2013, pp. 268–278.
30. F. Palomba, G. Bavota, M. D. Penta, R. Oliverto, D. Poshyvanyk, and A. D. Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.
31. M. M. V. Z. M. M. A. Arcelli Fontana, F., "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, 2015.
32. W. L. 0014 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*, vol. 80, no. 7, pp. 1120–1128, 2007.
33. B. Cardoso and E. Figueiredo, "Co-occurrence of design patterns and bad smells in software systems: An exploratory study," in *Proceedings of the annual conference on Brazilian Symposium on Information Systems: Information Systems: A Computer Socio-Technical Perspective*. Brazilian Computer Society, 2015, pp. 347–354.
34. S. M. Olbrich, D. Cruzes, V. R. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems." in *ESEM*, 2009, pp. 390–400.
35. R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining." in *CSMR*. IEEE, 2012, pp. 411–416.
36. F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *ICSME'14*, 2014, pp. 101–110.
37. A. De Lucia, M. Di Penta, and R. Oliveto, "Improving source code lexicon via traceability and information retrieval," *IEEE Trans. Software Eng.*, no. to appear, 2010.
38. S. L. Abebe and P. Tonella, "Automated identifier completion and replacement," in *CSMR'13*, 2013, pp. 263–272.

39. L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* iComment: Bugs or bad comments? */," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.
40. L. Tan, Y. Zhou, and Y. Padioleau, "aComment: Mining annotations from comments and code to detect interrupt-related concurrency bugs," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE11)*, May 2011.
41. S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing javadoc comments to detect comment-code inconsistencies," in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST)*, April 2012.