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

On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation

Fabio Palomba · Gabriele Bavota · Massimiliano Di Penta · Fausto Fasano · Rocco Oliveto · Andrea De Lucia

Received: date / Accepted: date

Abstract Code smells are symptoms of poor design and implementation choices that may hinder code comprehensibility and maintainability. Despite the effort devoted by the research community in studying code smells, the extent to which code smells in software systems affect software maintainability remains still unclear. In this paper we present a large scale empirical investigation on the diffuseness of code smells and their impact on code change- and fault-proneness. The study was conducted across a total of 395 releases of 30 open source projects and considering 17,350 manually validated instances of 13 different code smell kinds. The results show that smells characterized by long and/or complex code (*e.g., Complex Class*) are highly diffused, and that smelly classes have a higher change- and fault-proneness than smell-free classes.

Keywords Code Smells · Empirical Studies · Mining Software Repositories

Fabio Palomba
Delft University of Technology, The Netherlands
E-mail: f.palomba@tudelft.nl

Gabriele Bavota
Università della Svizzera italiana (USI), Switzerland
E-mail: gabriele.bavota@usi.ch

Massimiliano Di Penta
University of Sannio, Italy
E-mail: dipenta@unisannio.it

Fausto Fasano, Rocco Oliveto
University of Molise, Italy
E-mail: fausto.fasano@unimol.it E-mail: rocco.oliveto@unimol.it

Andrea De Lucia
University of Salerno, Italy
E-mail: adelucia@unisa.it

1 Introduction

Bad code smells (also known as "code smells" or "smells") were defined as symptoms of poor design and implementation choices applied by programmers during the development of a software project (Fowler, 1999). As a form of technical debt (Cunningham, 1993), they could hinder the comprehensibility and maintainability of software systems (Kruchten et al, 2012). An example of code smell is the *God Class*, a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. *God Classes* can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

The research community has been studying code smells from different perspectives. On the one side, researchers developed methods and tools to detect code smells. Such tools exploit different types of approaches, including metrics-based detection (Lanza and Marinescu, 2010; Moha et al, 2010; Marinescu, 2004; Munro, 2005), graph-based techniques (Tsantalis and Chatzigeorgiou, 2009), mining of code changes (Palomba et al, 2015a), textual analysis of source code (Palomba et al, 2016b), or search-based optimization techniques (Kessentini et al, 2010; Sahin et al, 2014). On the other side, researchers investigated how relevant code smells are for developers (Yamashita and Moonen, 2013; Palomba et al, 2014), when and why they are introduced (Tufano et al, 2015), how they evolve over time (Arcoverde et al, 2011; Chatzigeorgiou and Manakos, 2010; Lozano et al, 2007; Ratiu et al, 2004; Tufano et al, 2017), and whether they impact on software quality properties, such as program comprehensibility (Abbes et al, 2011), fault- and change-proneness (Khomh et al, 2012, 2009a; D'Ambros et al, 2010), and code maintainability (Yamashita and Moonen, 2012, 2013; Deligiannis et al, 2004; Li and Shatnawi, 2007; Olbrich et al, 2010; Sjoberg et al, 2013).

Similarly to some previous work (Khomh et al, 2012; Li and Shatnawi, 2007; Olbrich et al, 2010; Gatrell and Counsell, 2015) this paper investigates the relationship existing between the occurrence of code smells in software projects and software change- and fault-proneness. Specifically, while previous work shows a significant correlation between smells and code change/fault-proneness, the empirical evidence provided so far is still limited because of:

- **Limited size of previous studies:** the study by Khomh et al (2012) was conducted on four open source systems, while the study by D'Ambros et al (2010) was performed on seven systems. Furthermore, the studies by Li and Shatnawi (2007), Olbrich et al (2010), and Gatrell and Counsell (2015) were conducted considering the change history of only one software project.
- **Detected smells *vs.* manually validated smells:** Previous work studying the impact of code smells on change- and fault-proneness, including the one by Khomh et al (2012), relied on data obtained from automatic smell detectors. Although such smell detectors are often able to achieve a good level of accuracy, it is still possible that their intrinsic imprecision affects the results of the study.

- **Lack of analysis of the magnitude of the observed phenomenon:** previous work indicated that some smells can be more harmful than others, but the analysis did not take into account the magnitude of the observed phenomenon. For example, even if a specific smell type may be considered harmful when analyzing its impact on maintainability, this may not be relevant in case the number of occurrences of such a smell type in software projects is limited.
- **Lack of analysis of the magnitude of the effect:** Previous work indicated that classes affected by code smells have more chances to exhibit defects (or to undergo changes) than other classes. However, no study has observed the magnitude of such changes and defects, *i.e.,* no study addressed the question: *How many defects would exhibit on average a class affected by a code smell as compared to another class affected by a different kind of smell, or not affected by any smell at all?*
- **Lack of within-artifact analysis:** sometimes, a class has intrinsically a very high change-proneness and/or fault-proneness, *e.g.,* because it plays a core role in the system or because it implements a very complex feature. Hence, the class may be intrinsically "smelly". Instead, there may be classes that become smelly during their lifetime because of maintenance activities (Tufano et al, 2017). Or else, classes where the smell was removed, possibly because of refactoring activities (Bavota et al, 2015). For such classes, it is of paramount importance to analyze the change- and fault-proneness of the class during its evolution, in order to better relate the cause (presence of smell) with the possible effect (change- or fault-proneness).
- **Lack of a temporal relation analysis between smell presence and fault introduction:** While previous work correlated the presence of code smells with high fault- and change-proneness, one may wonder whether the artifact was smelly when the fault was introduced, or whether the fault was introduced before the class became smelly.

To cope with the aforementioned issues, this paper aims at corroborating previous empirical research on the impact of code smells by analyzing their diffuseness and effect on change- and fault-proneness on a large set of software projects. In the context of this paper, the "diffuseness" of a code smell type (*e.g.,* God Class) refers to the percentage of code components in a system affected by at least one instance of the code smell type.

The study was conducted on a total of 395 releases of 30 open source systems, and considered 13 different kinds of code smells. More specifically, the study aims at investigating:

1. *the diffuseness of code smells in open source systems.* If the magnitude of the phenomenon is small—*i.e.,* code smells, or some specific kinds of code smells, are poorly diffused—then studying their impact on the code maintainability might not be worthwhile.
2. *the impact of code smells on maintenance properties and specifically on code change- and fault-proneness.* We intend to investigate to what extent the previous findings reported by Khomh et al (2012) and D'Ambros

et al (2010)—obtained on a smaller set of software systems and based on smell instances automatically identified using code smell detectors—are confirmed on a larger set of 395 software releases and considering manually validated smell instances.

To the best of our knowledge, this is to date the largest study investigating the relationship between the presence of code smells and source code change- and fault-proneness. In addition, and to cope with the other limitations of previous studies mentioned above, this paper (i) relies on a set of manually-validated code smells rather than just on the output of detection tools, (ii) analyzes the fault proneness magnitude in terms of number of code smell instances, (iii) performs an analysis of the evolution of classes in order to investigate how the change/fault-proneness changes when the smell was removed, and (iv) uses the SZZ algorithm (Sliwerski et al, 2005) to determine whether an artifact was already smelly when a fault was induced. The dataset used in this study is publicly available in our online appendix (Palomba et al, 2017).

Structure of the paper. Section 2 discusses the related literature about smell detection and about studies on the effect of code smells. Section 3 describes the design and planning of the empirical study. Section 4 presents and discusses the results of the study, while the threats that could affect their validity are discussed in Section 5. Finally, Section 6 concludes the paper, discussing the main findings of the work.

2 Related work

The research community has been highly active in the definition of code smell detection methods and tools, as well as in the investigation of the impact of code smells on software maintenance properties. In this section we report the literature related to (i) empirical studies aimed at understanding to what extent code smells are diffused in software systems and how they evolve over time, (ii) the impact of code smells on change- and fault-proneness, and (iii) user studies conducted in order to comprehend the phenomenon from a developer's perspective. A complete overview of code smell detection techniques can be found in related papers by Palomba et al (2015c) and Fernandes et al (2016).

2.1 Diffuseness and Evolution of Code Smells

Chatzigeorgiou and Manakos (2010) analyzed the evolution of code smells. Their results indicate that (i) the number of instances of code smells increases during time; and (ii) developers are reluctant to perform refactoring operations in order to remove them. Peters and Zaidman (2012) obtained similar results, showing that developers are often aware of the presence of code smells in the

source code, but they do not invest time in performing refactoring activities aimed at removing them. A partial explanation for this behavior is provided by Arcoverde et al (2011), who studied the longevity of code smells showing that they often survive for a long time in the source code. The authors point to the will of avoiding changes to API as one of the main reason behind this result (Arcoverde et al, 2011).

The evolution of code smells has also been studied by Olbrich et al (2009), who analyzed the evolution of *God Class* and *Shotgun Surgery*, showing that there are periods in which the number of smells increases and periods in which this number decreases. They also show that the increase/decrease of the number of instances does not depend on the size of the system.

Vaucher et al (2009) conducted a study on the evolution of the *God Class* smell, aimed at understanding whether they affect software systems for long periods of time or, instead, are refactored while the system evolves. Their goal was to define a method able to discriminate between *God Class* instances that are introduced by design and *God Class* instances that are introduced unintentionally. Recently, Tufano et al (2015) investigated when code smells are introduced by developers, and the circumstances and reasons behind their introduction. They showed that most of the times code artifacts are affected by smells since their creation and that developers introduce them not only when implementing new features or enhancing existing ones, but sometimes also during refactoring. A similar study was also conducted on test smells (Tufano et al, 2016). Furthermore, Tufano et al (2017) also found that almost 80% of the code smells are never removed from software systems, and the main cause for their removal is the removal of the smelly artifact, rather than refactoring activities. In a closely related field, Bavota et al (2012) and Palomba et al (2016a) provided evidence that test smells are also widely diffused in test code and impact the maintainability of JUnit test classes.

Historical information, in general, and the evolution of code smells, in particular, was also used in the past to identify components affected by code smells. Ratiu et al (2004) proposed an approach to detect smells based on evolutionary information of code components over their life-time. The aim is to analyze the persistence of the problem and the effort spent to maintain these components. Historical information has also been used by Lozano et al (2007) to assess the impact of code smells on software maintenance. Gîrba et al (2007) exploited formal concept analysis (FCA) to detect co-change patterns. In other words, they identified code components that change in the same way and at the same time. Palomba et al (2015b) use association rule discovery to detect some code smell types, showing that the evolutionary-based approach outperforms approaches based on static and dynamic analysis and could also successfully complement them.

Our investigation about the diffuseness of code smells ($\mathbf{RQ}_1$) is closely related to the empirical studies discussed above. However, our goal is to analyze whether the results achieved in previous work hold on the set of software systems used in this paper in order to (i) corroborate previous findings on a much larger dataset (both in terms of number of software systems and code

smells), and (ii) understand the confidence level for the generalizability of the results provided through the analysis of the impact of code smells on change- and fault-proneness.

2.2 Change- and Fault-proneness of Code Smells

The main goal of this paper is to analyze the change- and fault-proneness of classes affected (and not) by code smells. Such a relationship has already been investigated by previous research. In particular, Khomh et al (2009a) showed that the presence of code smells increases the code change proneness. Also, they showed that code components affected by code smells are more fault-prone than non-smelly components (Khomh et al, 2012). Our work confirms the results achieved by Khomh et al (2012) on a larger set of code smells and software systems, an provides some complementary hints about the phenomenon. In particular, other than studying the change- and fault-proneness of smelly and non-smelly classes, we analyzed how such indicators vary when the smells identified are removed. Also, we use the SZZ algorithm (Sliwerski et al, 2005) to better investigate the temporal relationship between the presence of code smells and fault introduction.

Gatrell and Counsell (2015) conducted an empirical study aimed at quantifying the effect of refactoring on class change- and fault-proneness. In particular, they monitored a commercial C# system for twelve months identifying the refactorings applied during the first four months. They examined the same classes for the second four months in order to determine whether the refactoring results in a decrease of change- and fault-proneness. They also compared such classes with the classes of the system that were not refactored in the same period. Results revealed that classes subject to refactoring have a lower change- and fault-proneness. It is worth noting that Gatrell and Counsell did not focus their attention on well known design problems (*i.e.,* code smells) but they analyzed if refactored classes regardless of the presence of a design problem. Instead, our study investigates the actual impact of code smells on change- and fault-proneness. Moreover, their study was conducted on a single software system, while we analyzed 395 software releases of 30 software systems.

Li and Shatnawi (2007) empirically evaluated the correlation between the presence of code smells and the probability that the class contains errors. They studied the post-release evolution process showing that many code smells are positively correlated with class errors. Olbrich et al (2010) conducted a study on the *God Class* and *Brain Class* code smells, reporting that these code smells were changed less frequently and had a fewer number of defects than non-smelly classes. D'Ambros et al (2010) also studied the correlation between the *Feature Envy* and *Shotgun Surgery* smells and the defects in a system, reporting no consistent correlation between them. In our empirical study, we do not consider correlation between the presence of smells and the number of defects, but we investigate the release history of software systems in order to

measure the actual change- and fault-proneness of classes affected (and not) by design flaws.

Finally, Saboury et al (2017) conducted an empirical investigation on the impact of code smells on the fault-proneness of JavaScript modules, confirming the negative effect smells have on the maintainability of source code. Similarly to our study, Saboury et al (2017) used of the SZZ algorithm to identify which bugs were introduced after the introduction of the smells.

2.3 Code Smells and User Studies

Abbes et al (2011) studied the impact of two code smell types, *i.e., Blob* and *Spaghetti Code*, on program comprehension. Their results show that the presence of a code smell in a class does not have an important impact on developers' ability to comprehend the code. Instead, a combination of more code smells affecting the same code components strongly decreases developers' ability to deal with comprehension tasks.

The interaction between different smell instances affecting the same code components was also been studied by Yamashita and Moonen (2013), who confirmed that developers experience more difficulties when working on classes affected by more than one code smell. The same authors also analyzed the impact of code smells on maintainability characteristics (Yamashita and Moonen, 2012). They identified 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.

Sjoberg et al (2013) investigated the impact of twelve code smells on the maintainability of software systems. In particular, the authors conducted a study with six industrial developers involved in three maintenance tasks on four Java systems. The amount of time spent by each developer in performing the required tasks whas been measured through an Eclipse plug-in, while a regression analysis whas been used to measure the maintenance effort on source code files having specific properties, including the number of smells affecting them. The achieved results show that smells do not always constitute a problem, and that often class size impacts maintainability more than the presence of smells.

Deligiannis et al (2004) also performed a controlled experiment showing that the presence of *God Class* smell negatively affects the maintainability of source code. Also, the authors highlight an influence played by these smells in the way developers apply the inheritance mechanism.

Recently, Palomba et al (2014) investigated how the developers perceive code smells, showing that smells characterized by long and complex code are those perceived more by developers as design problems. In this paper we provide a complementary contribution to the previous work by Palomba et al (2014). Rather than looking at developers' perception, this paper observes the possible effect of smells in terms of change- and fault-proneness.

Table 1: Code smells considered in the context of the study

Name	Description
Class Data Should Be Private (CDSBP)	A class exposing its fields, violating the principle of data hiding.
Complex Class	A class having at least one method having a high cyclomatic complexity.
Feature Envy	A method more interested in a class other than the one it actually is in.
God Class	A large class implementing different responsibilities and centralizing most of the system processing.
Inappropriate Intimacy	Two classes exhibiting a very high coupling between them.
Lazy Class	A class having very small dimension, few methods and low complexity.
Long Method	A method that is unduly long in terms of lines of code.
Long Parameter List (LPL)	A method having a long list of parameters, some of which avoidable.
Message Chain	A long chain of method invocations performed to implement a class functionality.
Middle Man	A class delegating to other classes most of the methods it implements.
Refused Bequest	A class redefining most of the inherited methods, thus signaling a wrong hierarchy.
Spaghetti Code	A class implementing complex methods interacting between them, with no parameters, using global variables.
Speculative Generality	A class declared as abstract having very few children classes using its methods.

3 Study Definition and Planning

The *goal* of this study is to analyze the diffuseness of 13 code smell types in real software applications and to assess their impact on code change- and fault-proneness. It is worth remarking that the term "diffuseness", when associated to a code smell type, refers to the percentage of code components in a system affected by at least one instance of the smell type. Analyzing the diffuseness of code smells is a preliminary analysis needed to better interpret their effect on change- and fault-proneness. Indeed, some smells might be highly correlated with fault-proneness but rarely diffused in software projects or *vice versa*. The 13 code smell types considered in this study are listed in Table 1 together with a short description.

3.1 Research Questions and Planning

We formulated the following three research questions:

- **RQ$_1$**: *What is the diffuseness of code smells in software systems?* This is a preliminary research question aiming at assessing to what extent software systems are affected by code smells.
- **RQ$_2$**: *To what extent do classes affected by code smells exhibit a different level of change- and fault-proneness with respect to non-smelly classes?* Previous work (Khomh et al, 2012) found that classes affected by at least one smell have a higher chance of being change- and fault-prone than non-smelly classes. In this work we are interested in measuring the change- and fault-proneness magnitude of such classes, in terms of number of changes and of bug fixes.
- **RQ$_3$**: *To what extent do change- and fault-proneness of classes vary as a consequence of code smell introduction and removal?* This research question

Table 2: Systems involved in the study

System	Description	#Releases	Classes	Methods	KLOCs
ArgoUML	UML Modeling Tool	16	777-1,415	6,618-10,450	147-249
Ant	Build System	22	83-813	769-8,540	20-204
aTunes	Player and Audio Manager	31	141-655	1,175-5,109	20-106
Cassandra	Database Management System	13	305-586	1,857-5,730	70-111
Derby	Relational Database Management System	9	1,440-1,929	20,517-28,119	558-734
Eclipse Core	Integrated Development Environment	29	744-1,181	9,006-18,234	167-441
Elastic Search	RESTful Search and Analytics Engine	8	1,651-2,265	10,944-17,095	192-316
FreeMind	Mind-mapping Tool	16	25-509	341-4,499	4-103
Hadoop	Tool for Distributed Computing	9	129-278	1,089-2,595	23-57
HSQLDB	HyperSQL Database Engine	17	54-444	876-8,808	26-260
Hbase	Distributed Database System	8	160-699	1,523-8148	49-271
Hibernate	Java Persistence Framework	11	5-5	15-18	0.4-0.5
Hive	Data Warehouse Software Facilitates	8	407-1,115	3,725-9,572	64-204
Incubating	Codebase	6	249-317	2,529-3,312	117-136
Ivy	Dependency Manager	11	278-349	2,816-3,775	43-58
Lucene	Search Manager	6	1,762-2,246	13,487-17,021	333-466
JEdit	Text Editor	23	228-520	1,073-5,411	39-166
JHotDraw	Java GUI Framework	16	159-679	1,473-6,687	18-135
JFreeChart	Java Chart Library	23	86-775	703-8,746	15-231
JBoss	Java Webserver	18	2,313-4,809	19,901-37,835	434-868
JVlt	Vocabulary Learning Tool	15	164-221	1,358-1,714	18-29
jSL	Java Service Launcher	15	5-10	26-43	0.5-1
Karaf	Standalone Container	5	247-470	1,371-2,678	30-56
Nutch	Web-search Software	7	183-259	1,131-1,937	33-51
Pig	Large Dataset Analyzer	8	258-922	1,755-7,619	34-184
Qpid	Messaging Tool	5	966-922	9,048-9,777	89-193
Sax	XML Parser	6	19-38	119-374	3-11
Struts	MVC Framework	7	619-1,002	4,059-7,506	69-152
Wicket	Java Application Framework	9	794-825	6,693-6,900	174-179
Xerces	XML Parser	16	162-736	1,790-7,342	62-201
Total	-	**395**	**5-4,809**	**15-37,835**	**0.4-868**

investigates whether the change- and fault-proneness of a class increases when a smell is introduced, and whether it decreases when the smell is removed. Such an analysis is of paramount importance because a class may be intrinsically change-prone (and also fault-prone) regardless of whether it is affected by code smells.

To answer our research questions we mined 395 releases of 30 open source systems searching for instances of the 13 code smells object of our study. Table 2 reports the analyzed systems, the number of releases considered for each of them, and their size ranges in terms of number of classes, number of methods, and KLOCs. The choice of the subject systems was driven by the will to consider systems having different size (ranging from 0.4 to 868 KLOCs), belonging to different application domains (modeling tools, parsers, IDEs, IR-engines, *etc.*), developed by different open source communities (Apache, Eclipse, *etc.*), and having different lifetime (from 1 to 19 years).

The need for analyzing smells in 395 project releases makes the manual detection of the code smells prohibitively expensive. For this reason, we developed a simple tool to perform smell detection. The tool outputs a list of candidate code components (*i.e.,* classes or methods) potentially exhibiting a smell. Then, we manually validated the candidate code smells suggested by the tool. The validation was performed by two of the authors who individually analyzed and classified as true or false positives all candidate code smells. Finally, they performed an open discussion to resolve possible conflicts and reach

Table 3: The rules used by our tool to detect candidate code smells

Name	Description
CDSBP	A class having at least one public field.
Complex Class	A class having at least one method for which McCabe cyclomatic complexity is higher than 10.
Feature Envy	All methods having more calls with another class than the one they are implemented in.
God Class	All classes having (i) cohesion lower than the average of the system AND (ii) LOCs > 500.
Inappropriate Intimacy	All pairs of classes having a number of method's calls between them higher than the average number of calls between all pairs of classes.
Lazy Class	All classes having LOCs lower than the first quartile of the distribution of LOCs for all system's classes.
Long Method	All methods having LOCs higher than the average of the system.
LPL	All methods having a number of parameters higher than the average of the system.
Message Chain	All chains of methods' calls longer than three.
Middle Man	All classes delegating more than half of the implemented methods.
Refused Bequest	All classes overriding more than half of the methods inherited by a superclass.
Spaghetti Code	A class implementing at least two long methods interacting between them through method calls or shared fields.
Speculative Generality	A class declared as abstract having less than three children classes using its methods.

a consensus on the detected code smells. To ensure high recall, our detection tool uses very simple rules that overestimate the presence of code smells.

The rules for the 13 smell types considered in the study are reported in Table 3 and are inspired to the rule cards proposed by Moha et al (2010) in *DECOR*. The metrics' thresholds used to discriminate whether a class/method is affected or not by a smell are lower than the thresholds used by Moha et al (2010). Again, this was done in order to detect as many code smell instances as possible. For example, in the case of the *Complex Class* smell we considered as candidates all the classes having a cyclomatic complexity higher than 10. Such a choice was driven by recent findings reported by Lopez and Habra (2015), which found that *"a threshold lower than 10 is not significant in Object-Oriented programming when interpreting the complexity of a method"*. As for the other smells we relied on (i) simple filters, *e.g.,* in the cases of *CDSBP* (where we discarded from the manual validation all the classes having no public attributes) and *Feature Envy* (we only considered the methods having more relationships toward another class than with the class they are contained in), (ii) the analysis of the metrics' distribution (like in the cases of *Lazy Class*, *Inappropriate Intimacy*, *Long Method*, and *Long Parameter List*), or (iii) very conservative thresholds (*e.g.,* a God Class should not have less than 500 LOCs).

We chose not to use existing detection tools (Marinescu, 2004; Khomh et al, 2009b; Sahin et al, 2014; Tsantalis and Chatzigeorgiou, 2009; Moha et al, 2010; Oliveto et al, 2010; Palomba et al, 2015a, 2016b) because (i) none of them has ever been applied to detect all the studied code smells and (ii) their detection rules are generally more restrictive to ensure a good compromise between recall and precision and thus may miss some smell instances. To verify this claim, we evaluated the behavior of three existing tools, *i.e.,* DECOR

(Moha et al, 2010), JDeodorant (Tsantalis and Chatzigeorgiou, 2009), and HIST (Palomba et al, 2015a) on one of the systems used in the empirical study, *i.e.,* Apache Cassandra 1.1. When considering the God Class smell none of the available tools is able to identify all the eight actual smell instances we found by manually analyzing the classes of this system. Indeed, DECOR identifies only one of the actual instances, while JDeodorant and HIST detect three of them. Therefore, the use of existing tools would have resulted in a less comprehensive analysis. Of course, using rules that overestimate the presence of code smells pays the higher recall with a lower precision with respect to other tools. However, this is not a threat for our study, because the manual validation of the instances detected by the tool aims at discarding the false positives, while keeping the true positive smell instances. A detailed overview of the results obtained by the tools on Apache Cassandra is available in our online appendix (Palomba et al, 2017).

We used the collected data to answer our research questions. Concerning $\mathbf{RQ}_1$ we verified what is the diffuseness of the considered code smells in the analyzed systems. We also verified whether there is a correlation between systems' characteristics (#Classes, #Methods, and KLOCs) and the presence of code smells. To compute the correlation on each analyzed system release we apply the Spearman rank correlation analysis (Student, 1921) between the different characteristics of the system release and the presence of code smells. Such an analysis measures the strength and direction of association between two ranked variables, and ranges between -1 and 1, where 1 represents a perfect positive linear relationship, -1 represents a perfect negative linear relationship, and values in between indicate the degree of linear dependence between the considered distributions. Cohen (1988) provided a set of guidelines for the interpretation of the correlation coefficient. It is assumed that there is no correlation when $0 \leq \rho < 0.1$, small correlation when $0.1 \leq \rho < 0.3$, medium correlation when $0.3 \leq \rho < 0.5$, and strong correlation when $0.5 \leq \rho \leq 1$. Similar intervals also apply for negative correlations.

To answer $\mathbf{RQ}_2$ we mined the change history of the 30 systems subject of our study. In particular, to compute the class change-proneness, we extracted the change logs from their versioning systems in order to identify the set of classes modified in each commit. Then, we computed the change-proneness of a class C_i in a release r_j as:

$$change\_proneness(C_i, r_j) = \#Changes(C_i)_{r_{j-1} \rightarrow r_j}$$

where $\#Changes(C_i)_{r_{j-1} \rightarrow r_j}$ is the number of changes performed on C_i by developers during the evolution of the system between the r_{j-1}'s and the r_j's release dates.

As for the fault-proneness, we developed a mining tool to extract the bugs fixed over the change history of the subject systems. All considered systems exploit Bugzilla[1] or Jira[2] as issue tracker. Firstly, we identified bug fixing

[1] `http://www.bugzilla.org`

[2] `https://www.atlassian.com/software/jira`

commits by mining regular expressions containing issue IDs in the change log of the versioning system, *e.g.,"fixed issue #ID"* or *"issue ID"*. Secondly, for each issue ID related to a commit, we downloaded the corresponding issue reports from their issue tracking system and extracted the following information from them: (i) *product name*; (ii) *issue type*, *i.e.,* whether an issue is a bug, enhancement request, *etc.*; (iii) *issue status*, *i.e.,* whether an issue was closed or not; (iv) *issue resolution*, *i.e.,* whether an issue was resolved by fixing it, or whether it was a duplicate bug report, or a "works for me" case; (v) *issue opening date*; (vi) *issue closing date*, if available.

Then, we checked each issue report to be correctly downloaded (*e.g.,* the issue ID identified from the versioning system commit note could be a false positive). After that, we used the issue type field to classify the issue and distinguish bug fixes from other issue types (*e.g.,* enhancements). Finally, we only considered bugs having *Closed* status and *Fixed* resolution. In this way, we restricted our attention to (i) issues that were related to bugs, and (ii) issues that were neither duplicate reports nor false alarms. Having bugs linked to the commits fixing them allowed us to identify which classes were modified to fix each bug. Thus, we computed the fault-proneness of a class C_i in a release r_j as the number of bug fixing activities involving the class in the period of time between the r_{j-1} and the r_j release dates.

Once extracted all the required information, we compare the distribution of change- and fault-proneness of classes affected and not by code smells. In particular, we present boxplots of change- and fault- proneness distributions of the two sets of classes and we also compare them through the Mann-Whitney statistical test (Conover, 1998). The latter is a non-parametric test used to evaluate the null hypothesis stating that it is equally likely that a randomly selected value from one sample will be less than or greater than a randomly selected value from a second sample. The results are intended as statistically significant at $\alpha = 0.05$. We estimated the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure (Grissom and Kim, 2005) for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ (Grissom and Kim, 2005).

It is important to note that the analysis of the fault-proneness might be biased by the fact that a bug might have been introduced before the introduction of the code smell. This would lead to an overestimation of the actual number of bug fixing activities performed on smelly classes in the time period between the releases r_{j-1} and r_j. For this reason, we also analyzed the fault-proneness of smelly classes only considering bug fixing activities related to bugs introduced after the smell introduction. More formally, we computed the fault-proneness of a smelly class C_i in a release r_j as the number of changes to C_i aimed at fixing a bug introduced after the code smell introduction in the period between r_{j-1} and r_j.

To estimate the date in which a bug was likely introduced[3], we exploited the SZZ algorithm[4] (Sliwerski et al, 2005), which is based on the annotation/blame feature of versioning systems. In summary, given a bug-fix identified by the bug ID, k, the approach works as follows:

1. For each file f_i, $i = 1 \dots m_k$ involved in the bug-fix k (m_k is the number of files changed in the bug-fix k), and fixed in its revision *rel-fix*$_{i,k}$, we extract the file revision just *before* the bug fixing (*rel-fix*$_{i,k} - 1$).
2. Starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the bug k the *blame* feature of *Git* is used to identify the file revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island grammar parser (Moonen, 2001). This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions *rel-bug*$_{i,j,k}$, $j = 1 \dots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for inducing a bug.

By adopting the process described above we were able to approximate the time periods in which each class was affected by one or more bugs. We excluded from our analysis all the bugs occurring in a class C_i before it became smelly. Note that we also excluded bug-introducing changes that were recorded after the bug was reported, since they represent false positives.

It is worth noting that in the context of $\mathbf{RQ}_2$ we considered all the classes of the analyzed systems: if a class was smelly in some releases and non-smelly in other releases, it contributes to both sets of smelly and non-smelly classes. Also, in this research question we did not discriminate the specific kind of smell affecting a class (*i.e.,* a class is considered smelly if it contains any kind of code smell). A fine-grained analysis of the impact of the different smell types on class change- and fault-proneness is presented in the next research question.

In $\mathbf{RQ}_3$ we exploited the code smells' oracle we built (*i.e.,* the one reporting the code smells affecting each class in each of the 395 considered releases) to identify in which releases of each system a class was smelly or not smelly. Then, we focused only on classes affected by at least one smell instance in at least one of the analyzed software releases but not in all of them. In this way, we could compare their change- and fault-proneness when they were affected and not affected by smells. To effectively investigate the effect of smell removal on maintainability, we considered each smell type in isolation, *i.e.,* we took into account only the classes affected by a single smell rather than considering classes affected by more smells. For example, suppose that a class C was firstly affected by the God Class smell between releases r_i and r_{i+1}. Then, the smell was not detected between releases r_{i+1} and r_{i+2}. Finally, the smell re-appeared between releases r_{i+2} and r_{i+3}. We compute the change-proneness of C when it was smelly by summing up the change-proneness of C in the periods between

[3] The right terminology is "when the bug induced the fix" because of the intrinsic limitations of the SZZ algorithm, which cannot precisely identify whether a change actually introduced the bug.

[4] SZZ stays for the last name initials of the three algorithm's authors.

r_i and r_{i+1} and between r_{i+2} and r_{i+3}. Similarly, we computed the change-proneness of C when it was non-smelly by computing the change-proneness of C in the period between r_{i+1} and r_{i+2}. Following the same procedure, we compare the fault-proneness of classes when they were affected and not by a code smell. As done for $\mathbf{RQ}_2$, the comparison is performed by using boxplots and statistical tests for significance (Mann-Whitney test) and effect size (Cliff's Delta).

4 Analysis of the Results

In this section we answer our three research questions.

4.1 Diffuseness of code smells ($\mathbf{RQ}_1$)

Fig. 1 shows the boxplot reporting (i) the absolute number of code smell instances, (ii) the percentage of affected code components (*i.e.,* percentage of affected classes/methods[5]), and (iii) the code smell density (*i.e.,* number of code smells per KLOC) affecting the software systems considered in our study. For sake of clarity, we aggregated the results considering all the systems as a single dataset. Detailed results are reported in the appendix at the end of the paper.

The boxplots highlight significant differences in the diffuseness of code smells. The first thing that leaps to the eyes is that code smells like *Feature Envy*, *Message Chain*, and *Middle Man* are poorly diffused in the analyzed systems. For instance, across the 395 system releases the highest number of *Feature Envy* instances in a single release (a Xerces release) is 17, leading to a percentage of affected methods of only 2.3%. We found instances of *Feature Envy* in 50% of the analyzed 395 releases.

The *Message Chain* smell is also poorly diffused. It affects 13% of the analyzed releases and in the most affected release (a release of `HSQLDB`) only four out of the 427 classes (0.9%) are instances of this smell. It is worth noting that in previous work *Message Chain* resulted to be the smell having the highest correlation with fault-proneness (Khomh et al, 2012). Therefore, the observed results indicate that although the *Message Chain* smell is potentially harmful its diffusion is fairly limited.

Finally, the last poorly diffused code smell is the *Middle Man*. Only 30% of the 395 analyzed releases are affected by this smell type and the highest number of instances of this smell type in a single release (a release of `Cassandra`) is eight. In particular, the classes affected by the *Middle Man* in Cassandra 0.6 were 8 out of 261 (3%). In this case, all identified *Middle Man* instances affect classes belonging to the `org.apache.cassandra.utils` package, grouping together classes delegating most of their work to classes

[5] Depending on the code smell granularity, we report the percentage of affected classes or methods.

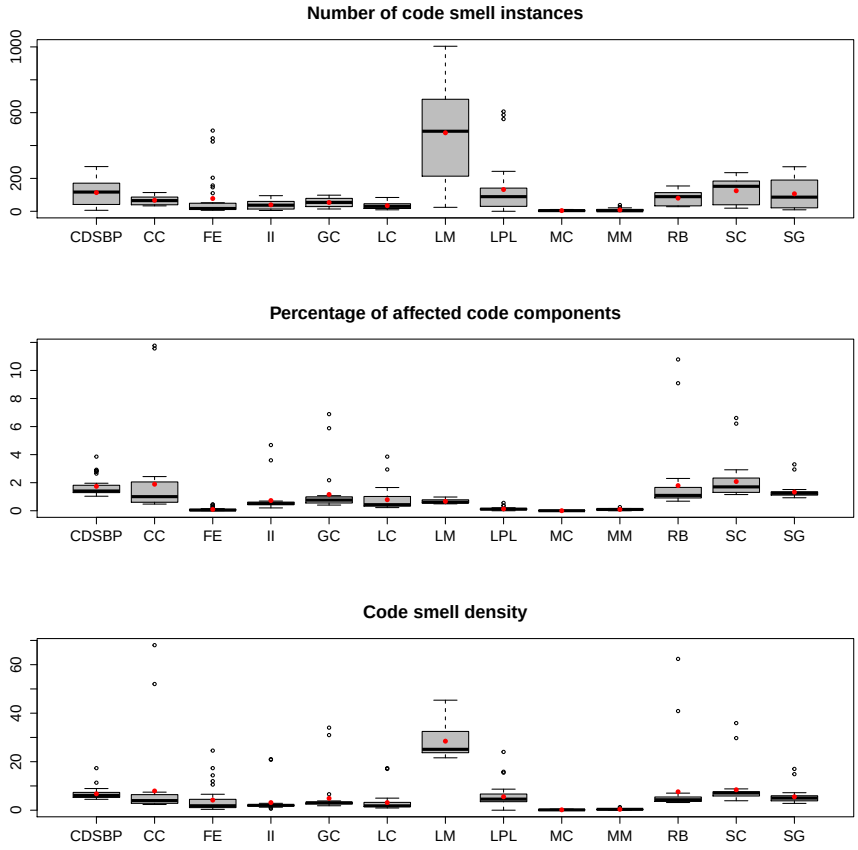

Fig. 1: Absolute number, percentage, and density of code smell instances in the analyzed systems.

in other packages. For example, the `HintedHandOffManager` class delegates eleven out of the twelve methods it contains to the `StorageService` class from the `org.apache.cassandra.service` package.

Other code smells are instead quite diffused. For example, we found at least one instance of *Long Method* in 84% of the analyzed releases (331 out of 395). In particular, on average each of these 331 releases is affected by 44 *Long Method* instances with the peak of 212 in an `Apache Derby` release. We manually analyzed that release (*i.e.,* `10.1`) to understand the reasons behind the presence of so many *Long Method* instances. Most of the instances are in the `org.apache.derby.impl.sql.compile` package, grouping together classes implementing methods responsible for parsing code statements written by using the SQL language. Such parsing methods are in general very com-

Table 4: Correlation between code smell instances and system size.

Code smell	ρ with #Classes	ρ with #Methods	ρ with LOCs
Class Data Should Be Private	**0.72**	**0.82**	**0.82**
Complex Class	*0.49*	**0.71**	**0.73**
Feature Envy	-0.07	-0.02	0.01
God Class	**0.50**	**0.76**	**0.82**
Inappropriate Intimacy	-0.02	0.02	0.08
Lazy Class	0.20	*0.32*	*0.32*
Long Method	*0.47*	**0.72**	**0.79**
Long Parameter List	-0.12	-0.09	-0.05
Message Chain	-0.10	-0.03	0.03
Middle Man	0.07	0.19	0.18
Refused Bequest	**0.74**	**0.82**	**0.81**
Spaghetti Code	**0.69**	**0.74**	**0.75**
Speculative Generality	**0.85**	**0.78**	**0.77**

In *Italic* the medium correlations, in **bold** the strong correlations

plex and long (on average, 259 LOC). For a similar reason, we found several instances of *Long Method* in `Eclipse Core`. Indeed, it contains a high number of classes implementing methods dealing with code parsing in the IDE. While we cannot draw any clear conclusion based on the manual analysis of these two systems, our feeling is that the inherent complexity of such parsing methods makes it difficult for developers to (i) write the code in a more concise way to avoid *Long Method* code smells, or (ii) remove the smell, for instance by applying extract method refactoring.

Another quite diffused code smell is the *Spaghetti Code*, that affects 83% of the analyzed releases (327 out of 395) with the highest number of instances (54) found in a `JBoss`'s release. Other diffused code smells are *Speculative Generality* (80% of affected releases), *Class Data Should Be Private* (77%), *Inappropriate Intimacy* (71%), and *God Class* (65%).

Interestingly, the three smallest systems considered in our study (*i.e.,* `Hibernate`, `jSL`, and `Sax`) do not present any instance of code smell in any of the 31 analyzed releases. This result might indicate that in small systems software developers are generally able to better keep under control the code quality, thus avoiding the introduction of code smells. To further investigate this point we computed the correlation between system size (in terms of # Classes, #Methods, and LOCs) and the number of instances of each code smell (see Table 4). As expected, some code smells have a positive correlation with the size attributes, meaning that the larger the system the higher the number of code smell instances in it. There are also several code smells for which this correlation does not hold (*i.e., Feature Envy*, *Inappropriate Intimacy*, *Long Parameter List*, *Message Chain*, and *Middle Man*). With the exception of *Long Parameter List*, all these smells are related to "suspicious" interactions between the classes of the system (*e.g.,* the high coupling represented by the *Inappropriate Intimacy* smell). It is reasonable to assume that

Table 5: **RQ**$_1$: Diffuseness of the studied code smells.

Code smell	% affected releases	avg. number of instances	max number of instances	Diffuseness
Long Method	84%	44	212	High
Spaghetti Code	83%	12	54	High
Speculative Generality	80%	11	65	High
Class Data Should Be Private	76%	12	65	High
Inappropriate Intimacy	71%	4	34	High
God Class	65%	5	26	Medium
Refused Bequest	58%	11	55	Medium
Complex Class	56%	9	35	Medium
Long Parameter List	47%	16	77	Medium
Feature Envy	50%	3	17	Low
Lazy Class	47%	5	21	Low
Middle Man	30%	2	8	Low
Message Chain	13%	2	4	Low

the interactions of such classes is independent from the system size and mainly related to correct/wrong design decisions.

We also compute the code smell density as the number of smell instances per KLOC in each of the 395 analyzed releases (see bottom part of Fig. 1). The results confirm that the *Long Method* is the most diffused smell, having the highest average density (*i.e.,* 28 instances per KLOC). Also *Refused Bequest* and *Complex Class* smells, *i.e.,* the code smells having the highest percentage of affected code components, are confirmed to be quite diffused in the studied systems. All the other smells seem to have diffuseness trends similar to the ones previously discussed.

Table 5 classifies the studied code smells on the basis of their diffuseness in the releases subject of our study. The "% of affected releases" column reports the percentage of analyzed releases in which we found at least one instance of a specific smell type. For example, a smell like *Long Method* affects 84% of releases, *i.e.,* 395*0.84=332 releases.

Summary for RQ$_1$. Most of the analyzed smells are quite diffused, especially the ones characterized by long and/or complex code (*e.g., Long Method*, *Complex Class*). On the contrary, *Feature Envy*, *Lazy Class*, *Message Chain*, and *Middle Man* are poorly diffused.

4.2 Change- and fault-proneness of classes affected/not affected by code smells (**RQ**$_2$)

Fig. 2 shows the boxplots of change-proneness for classes affected/not affected by code smells. Our results confirm the findings reported by Khomh et al (2012), showing that classes affected by code smells have a higher change-proneness than other classes. Indeed, the median change-proneness for classes

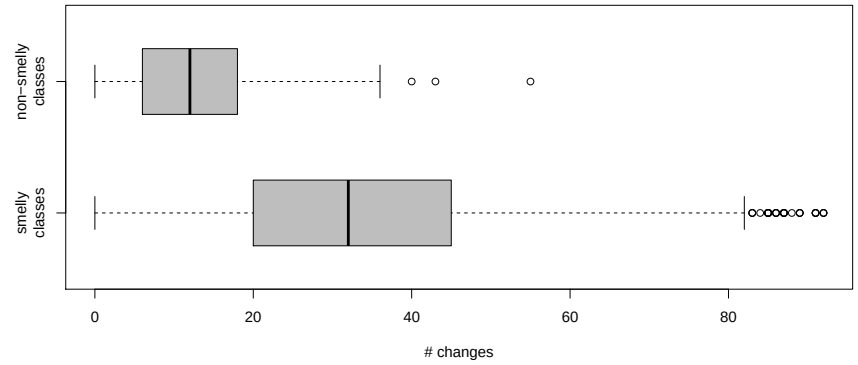

Fig. 2: Change-proneness of classes affected and not by code smells

affected by code smells (32) is almost three times higher with respect to the median change-proneness of the other classes (12). As an example, the `Eclipse` class `IndexAllProject` affected by the *Long Method* smell (in its method `execute`) was modified 77 times during the time period between the release 8 (2.1.3) and 9 (3.0), while the median value of changes for classes not affected by any code smell is 12. Moreover, during the change history of the system the number of lines of code of the method `execute` of this class varied between 671 and 968 due to the addition of several features. The results of the Mann-Whitney and Cliff tests highlight a statistically significant difference in the change-proneness of classes affected and not affected by code smell (p-value$<$0.001) with a large effect size (d=0.68).

Concerning the fault-proneness, the results also show important differences between classes affected and not affected by code smells, even if such differences are less marked than those observed for the change-proneness (see Fig. 3). The median value of the number of bugs fixed on classes not affected by smells is 3 (third quartile=5), while the median for classes affected by code smells is 9 (third quartile=12). The results confirm what already observed by Khomh et al (2012). The observed difference is statistically significant (p-value$<$0.001) with a medium effect size (d=0.41).

When considering only the bugs induced after the smell introduction, the results still confirm previous findings. Indeed, as shown in Fig. 4, smelly classes still have a much higher fault-proneness with respect to non-smelly classes. In particular, the median value of the number of bugs fixed in non-smelly classes is 2 (third quartile=5), while it is 9 for smelly classes (third quartile=12). The difference is statistically significant (p-value$<$0.001) with a large effect size (d=0.82).

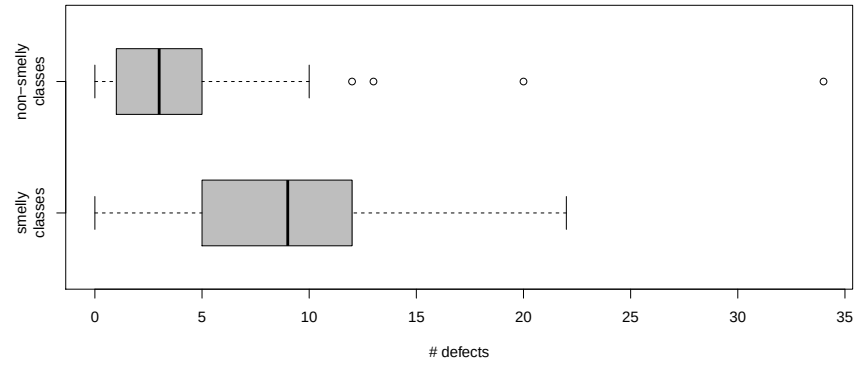

Fig. 3: Fault-proneness of classes affected and not affected by code smells.

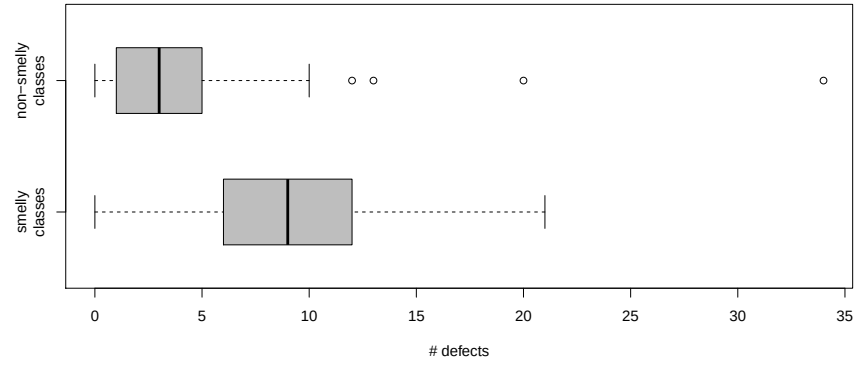

Fig. 4: Fault-proneness of classes affected and not affected by code smells when considering the bugs introduced after the smell introduction only.

This result can be explained by the findings reported in the work by Tufano et al (2017), where the authors showed that most of the smells are introduced during the very first commit involving the affected class (*i.e.,* when the class is added for the first time to the repository). As a consequence, most of the bugs are introduced after the code smell appearance. This conclusion is also supported by the fact that in our dataset only 21% of the bugs related to smelly classes are introduced before the smell introduction.

While the analysis carried out until now clearly highlighted a trend in terms of change- and fault- proneness of smelly and non-smelly classes, it is

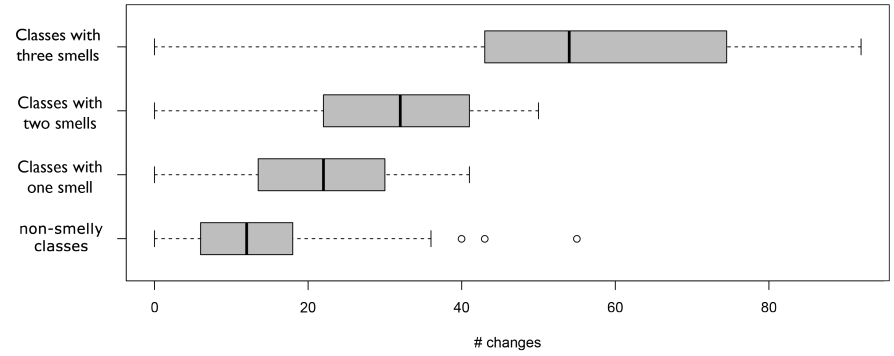

Fig. 5: Change-proneness of classes affected by different number of code smells.

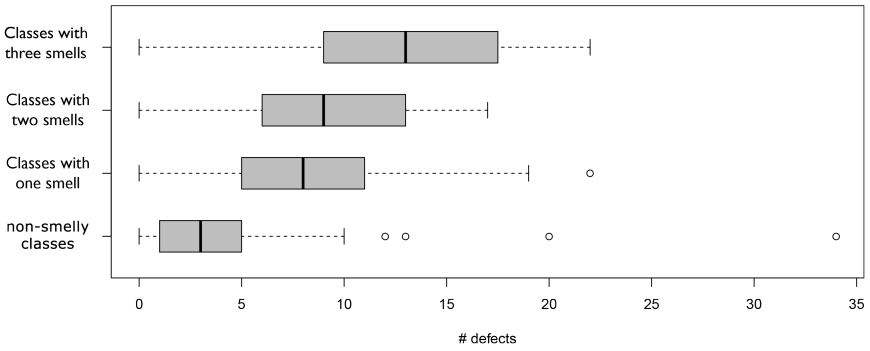

Fig. 6: Fault-proneness of classes affected by different number of code smells.

important to note that a smelly class could be affected by **one or more** smells. For this reason, we performed an additional analysis to verify how change- and fault-proneness of classes very when considering classes affected by zero, one, two, and three code smells. In our dataset there are no classes affected by more than three smells in the same system release. Moreover, if a class was affected by two code smells in release r_{j-1} and by three code smells in release r_j, its change- (fault-) proneness between releases r_{j-1} and r_j contributed to the distribution representing the change- (fault-) proneness of classes affected by two smells while its change- (fault-) proneness between releases r_j and r_{j+1} contributed to the distribution representing the change- (fault-) proneness of classes affected by three smells. Fig. 5 reports the change-proneness of the four considered sets of classes, while Fig. 6 and Fig. 7 depict the results achieved for fault-proneness.

In terms of change-proneness, the trend depicted in Fig. 5 shows that the higher the number of smells affecting a class the higher its change-proneness. In particular, the median number of changes goes from 12 for non-smelly classes,

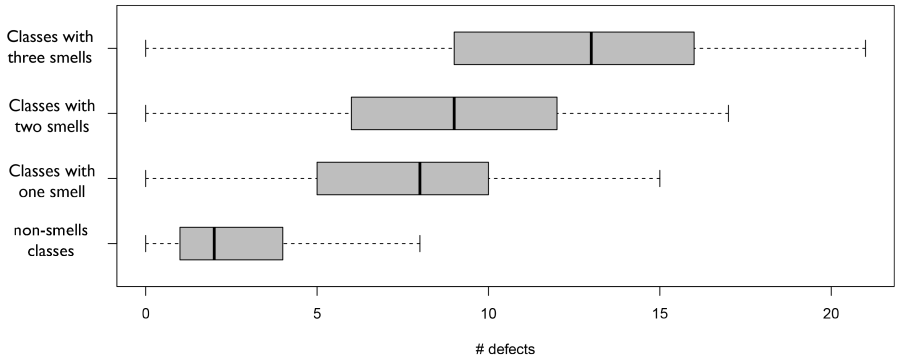

Fig. 7: Fault-proneness of classes affected by different number of code smells when considering only the bugs induced after the smell introduction.

Table 6: Change-proneness of classes affected by a different number of code smells: Mann-Whitney test (adj. p-value) and Cliff's Delta (d).

Test	adj. p-value	d
zero smells *vs* one smell	<0.001	0.53 (Large)
zero smells *vs* two smells	<0.001	0.80 (Large)
zero smells *vs* three smells	<0.001	0.89 (Large)
one smell *vs* two smells	<0.001	0.42 (Medium)
one smell *vs* three smells	<0.001	0.84 (Large)
two smells *vs* three smells	<0.001	0.72 (Large)

to 22 for classes affected by one smell (+83%), 32 for classes affected by two smells (+167%), and up to 54 for classes affected by three smells (+350%). Table 6 reports the results of the Mann-Whitney test and of the Cliff's delta obtained when comparing the change-proneness of these four categories of classes. Since we performed multiple tests, we adjusted our p-values using the Holm's correction procedure (Holm, 1979). This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n-1$, and so on.

The achieved results show that (i) classes affected by a lower number of code smells always exhibit a statistically significant lower change-proneness than classes affected by a higher number of code smells and (ii) the effect size is always large with the only exception of the comparison between classes affected by one smell and classes affected by two smells, for which the effect size is medium.

Similar observations can be made for what concerns the fault-proneness. Fig. 6 depicts the boxplots reporting the fault-proneness of classes affected by zero, one, two, and three code smells. When increasing the number of code smells, the median fault-proneness of the classes grows from 3 for the non-smelly classes up to 12 (+300%) for the classes affected by three code smells.

Table 7: Fault-proneness of classes affected by a different number of code smells: Mann-Whitney test (adj. p-value) and Cliff's Delta (d).

Test	adj. p-value	d
zero smells *vs* one smell	<0.001	0.74 (Large)
zero smells *vs* two smells	<0.001	0.74 (Large)
zero smells *vs* three smells	<0.001	0.89 (Large)
one smell *vs* two smells	<0.001	0.14 (Small)
one smell *vs* three smells	<0.001	0.53 (Large)
two smells *vs* three smells	<0.001	0.40 (Medium)

Table 8: Fault-proneness of classes affected by a different number of code smells when considering only bugs induced after the smell introduction: Mann-Whitney test (adj. p-value) and Cliff's Delta (d).

Test	adj. p-value	d
zero smells *vs* one smell	<0.001	0.75 (Large)
zero smells *vs* two smells	<0.001	0.71 (Large)
zero smells *vs* three smells	<0.001	0.95 (Large)
one smell *vs* two smells	<0.001	0.19 (Small)
one smell *vs* three smells	<0.001	0.61 (Large)
two smells *vs* three smells	<0.001	0.43 (Medium)

The results of the statistical analysis reported in Table 7 confirm the significant difference in the fault-proneness of classes affected by a different number of code smells, with a large effect size in most of the comparisons.

Previous findings are also confirmed when looking at the boxplots of Fig. 7, which refers to the analysis of the fault-proneness performed considering only the bugs introduced after the smell introduction. Indeed, the higher the number of code smells affecting a class the higher its fault-proneness. The significant differences are also confirmed by the statistical tests reported in Table 8.

Summary for RQ$_2$. Our results confirm the findings by Khomh et al (2012): Classes affected by code smells have a statistically significant higher change- (large effect size) and fault- (medium effect size) proneness with respect to classes not affected by code smells. Also, we observed a very clear trend indicating that the higher the number of smells affecting a class the higher its change- and fault-proneness.

4.3 Change- and fault-proneness of classes when code smells are introduced and removed (**RQ$_3$**)

For each considered code smell type, Fig. 8 shows a pair of boxplots reporting the change-proneness of the same set of classes during the time period in which they were affected (*S* in Fig. 8) and not affected (*NS* in Fig. 8) by that specific code smell.

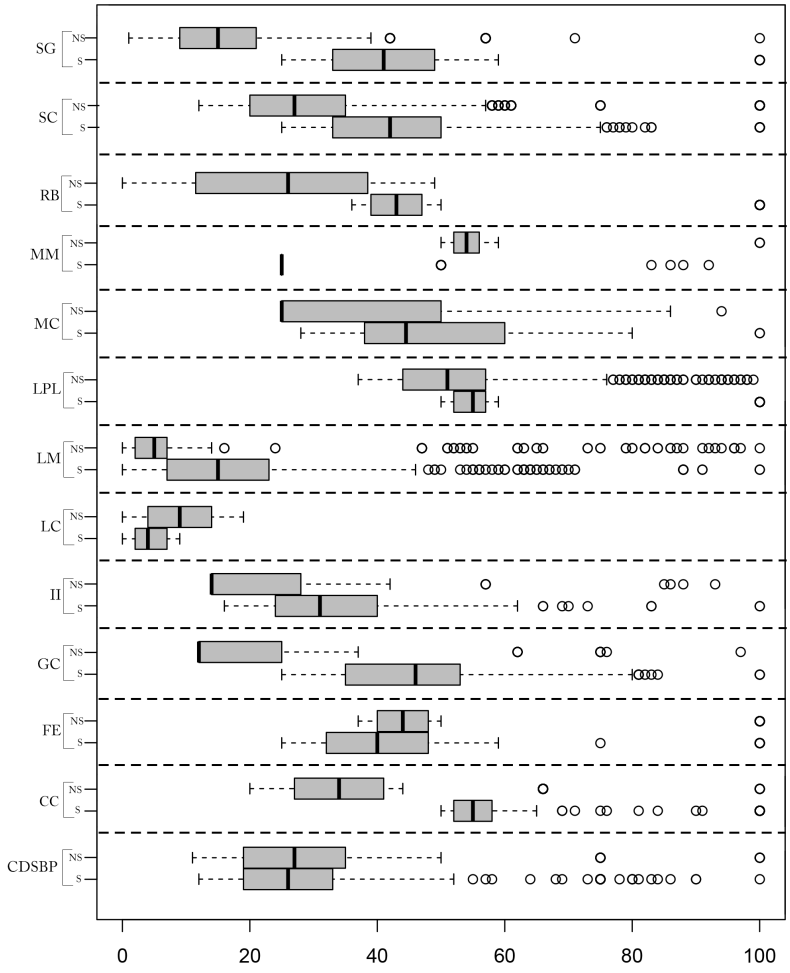

Fig. 8: Change-proneness of classes affected by a code smell (S) compared to the change-proneness of the same classes during the time period in which they were not affected by a code smell (NS).

In all pairs of boxplots a recurring pattern can be observed: when the classes are affected by the code smell they generally have a higher change-proneness than when they are not affected. This result holds for all code smells but *Middle Man* (MM), *Lazy Class* (LC), *Feature Envy* (FE), and *Class Data Should Be Private* (CDSBP).

For classes affected by a *God Class* (GC) smell we can observe an increase of +283% of the change-proneness median value (46 *vs* 12). The case of the `Base64` class belonging to the *Elastic Search* system is particularly represen-

tative: when affected by the *God Class* smell, the developers modified it 87 times on average (the average is computed across the 5 releases in which this class was smelly); instead, when the class was not affected by the code smell, the developers modified it only 10 times on average (the class was not smelly in 3 releases).

Similar results can be observed for the *Complex Class* (CC) smell: the median change-proneness of classes is 55 in the time period in which they are affected by this smell, while it is 34 when they are non-smelly. For example, when the `Scanner` class of the *Eclipse Core* project was affected by this smell, it was modified 95 times on average (across the 18 releases in which the class was smelly), as opposed to the 27 changes observed on average across the 11 releases in which it was not smelly.

The discussion is quite similar for code smells related to errors in the applications of Object Oriented principles. For example, for classes affected by *Refused Bequest* (RB) the median change-proneness goes from 43 (in the presence of the smell) down to 26 (in the absence of the smell). The case of the class `ScriptWriterBase` of the *HSQLDB* project is particularly interesting. On average this class was involved in 52 changes during the time period in which it was affected by RB (13 releases), while the average number of changes decreased to 9 during the time period in which it was not smelly (4 releases).

It is also interesting to understand why some code smells reduce the change-proneness. For the *Lazy Class* smell this result is quite expected. Indeed, by definition this smell arises when a class has small size, few methods, low complexity, and it is used rarely from the other classes; in other words, as stated by Fowler *"the class isn't doing enough to pay for itself"* (Fowler, 1999). Removing this smell could mean increasing the usefulness of the class, for example by implementing new features in it. This likely increases the class change-proneness. Also, the removal of a *Middle Man* (a class delegating most of its responsibilities) is expected to increase the change-proneness of classes, since the non-smelly class will implement (without delegation) a set of responsibilities that are likely to be maintained by developers, thus triggering new changes.

Results of the fault-proneness are shown in Fig. 9. Here, the differences between the time periods the classes are affected and not by code smells are less evident, but still present, especially for *Refused Bequest* (RB), *Inappropriate Intimacy* (II), *God Class* (GC), and *Feature Envy* (FE). The most interesting case is the FE, for which we observed that the fault-proneness increases by a factor of 8 when this code smell affects the classes. A representative example is represented by the method `internalGetRowKeyAtOrBefore` of the class `Memcache` of the project *Apache HBase*. This method did not present faults when it was not affected by any smell (*i.e.,* the method was not affected by smells in 4 releases of the system). However, when the method started to be too coupled with the class `HStoreKey`, it was affected by up to 7 faults. The reason for this growth is due to the increasing coupling of the method with the class `HStoreKey`. Indeed, a HBase developer commented on the evolution

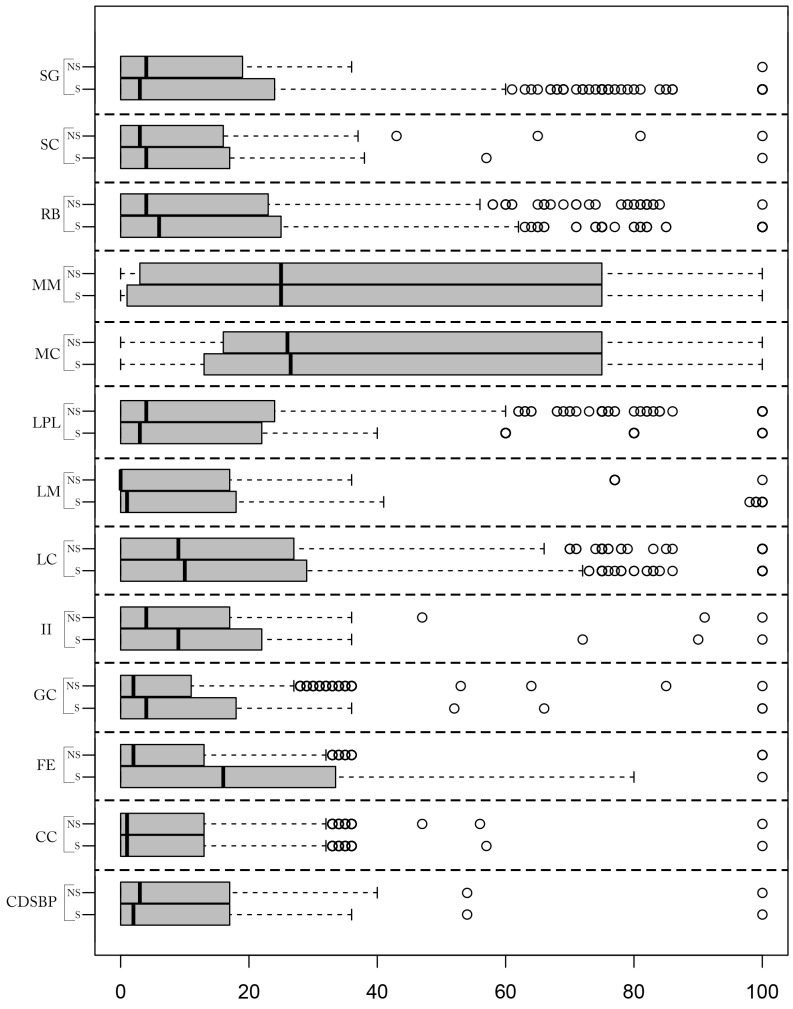

Fig. 9: Fault-proneness of classes affected by a code smell compared to the fault-proneness of the same classes during the time period in which they were not affected by a code smell.

of this method in the issue tracker[6]: "*Here's a go at it. The logic is much more complicated, though it shouldn't be too impossible to follow*".

For all other smells we did not observe any strong difference in the fault-proneness of the classes when comparing the time periods during which they were affected and not affected by code smells. While this result might seem

[6] https://issues.apache.org/jira/browse/HBASE-514

Table 9: ORs of independent factors when building logistic model. Statistically significant ORs are reported in bold face.

Dependent Variable	Smell Presence	Size	Their Interaction
Change-proneness	**4.46**	**1.7**	**8.41**
Defect-proneness	1.74	0.93	**2.11**

a contradiction with respect to what observed in RQ_2 and in the previous study by Khomh et al (2012), our interpretation is that classes that were fault-prone in the past will still continue to be fault-prone, even if a smell was removed. Moreover, since a smell removal requires a change to the code, it can have side effects like any other change, thus possibly affecting the fault-proneness independently of the smell. This is also in agreement with previous studies that used the past fault-proneness history of classes to predict their future faults (Ostrand et al, 2005). In essence, there seems to be no direct cause-effect relationships between the presence of code smells and the class fault-proneness. Rather, those are two different negative phenomena that tend to occur in some classes of a software project.

When analyzing only the bugs introduced after the smell appearance (Fig. 10), we can observe that also in this case the results are in line with those reported above. Indeed, there are no relevant changes between the findings achieved using or not such a filtering (based on the SZZ algorithm). As explained before, this is simply due to the fact that most of the code smells are introduced during the first commit of a class in the repository (Tufano et al, 2017).

Finally, it is important to point out that our analyses might be influenced by several confounding factors. For instance, it is likely that larger classes are more likely to change over time and to be subject to bug-fix activities. To verify the influence of the size attribute on the results achieved in the context of $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$ we built logistic regression models (Hosmer Jr and Lemeshow, 2004) relating the two phenomena, *i.e.,* change- and fault-proneness, with independent variables represented by the presence of a smell, the size of the component, and their interaction. Table 9 reports the ORs achieved from such an analysis. Statistically significant values, *i.e.,* those for which the p-value is lower than 0.05, are reported in bold face. From this analysis, we can notice that the presence of code smells is significantly related to the increase of change-proneness. The size of code components also affects change-proneness, although at a lower extent, while the interaction of smell presence and size has a strong impact on the change-proneness. In terms of fault-proneness, only the interaction between the independent variables is statistically significant. This confirms what we observed in $\mathbf{RQ}_3$: code smells are not necessarily the direct cause of the class fault-proneness.

Moreover, to be sure that the results achieved in the context of $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$ were not simply due to a reflection of code size, we re-ran our analysis by considering the change- and the fault-proneness of smelly and non-smelly classes having different size. In particular:

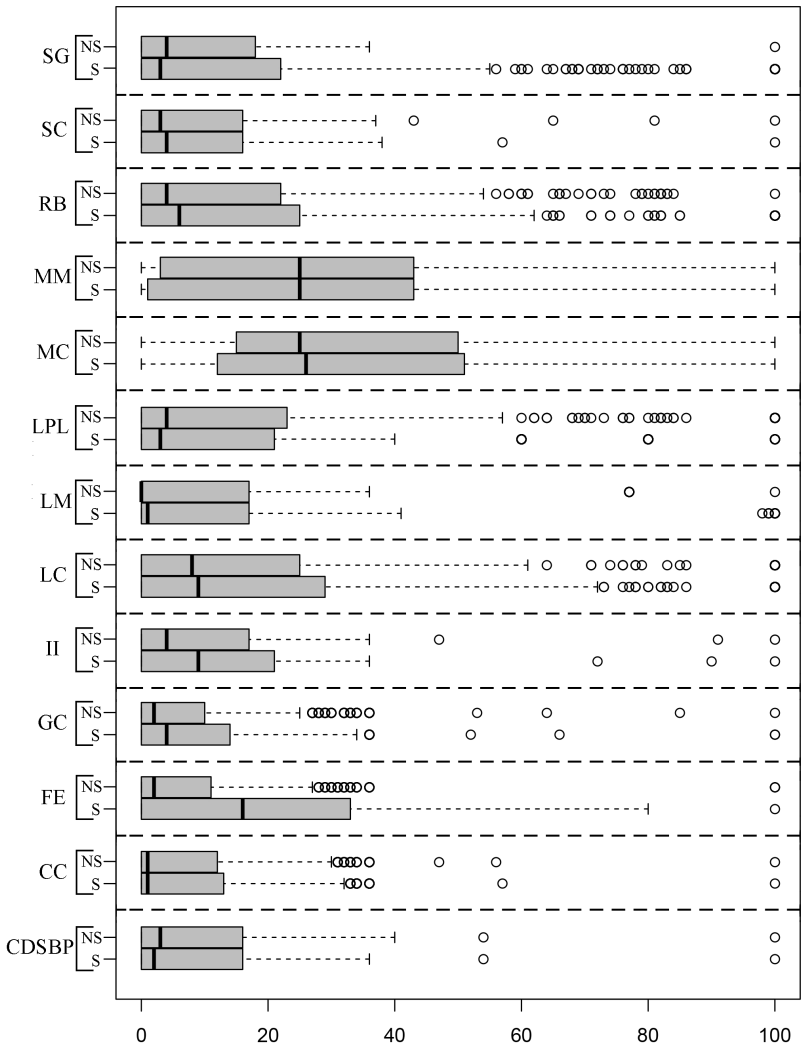

Fig. 10: SZZ Analysis: Fault-proneness of classes affected by a code smell compared to the fault-proneness of the same classes during the time period in which they were not affected by a code smell.

1. we grouped together *smelly* classes with similar size by considering their distribution in terms of size. Specifically, we compute the distribution of the lines of code of classes affected by code smells. This first step results in the construction of (i) the group composed by all the classes having a size lower than the first quartile of the distribution of the size of the classes, *i.e.*, *small* size; (ii) the group composed by all the smelly classes having a size

between the first and the third quartile of the distribution, *i.e., medium* size; and (iii) the group composed by the smelly classes having a size larger than the third quartile of the distribution of the size of the classes, *i.e., large* size;
2. we applied the same strategy for grouping *small*, *medium*, and *large* non-smelly classes; and
3. we computed the change- and the fault-proneness for each class belonging to the six groups, in order to investigate whether smelly-classes are more change- and fault-prone regardless of their size.

The obtained results are consistent with those discussed above. The interested reader can find them in our online appendix (Palomba et al, 2017).

Summary for RQ$_3$. While the class change-proneness can benefit from code smell removal, the presence of code smells in many cases is not necessarily the direct cause of the class fault-proneness, but rather a co-occurring phenomenon.

5 Threats to Validity

This section discusses the threats that might affect the validity of our study.

The main threats related to the relationship between theory and observation (*construct validity*) are due to imprecisions/errors in the measurements we performed. Above all, we relied on a tool we built and made publicly available in our online appendix (Palomba et al, 2017) to detect candidate code smell instances. Our tool exploits conservative detection rules aimed at ensuring high recall at the expense of low precision. Then, two of the authors manually validated the identified code smells to discard false positives. Still, we cannot exclude the presence of false positives/negatives in our dataset.

We assessed the change- and fault-proneness of a class C_i in a release r_j as the number of changes and the number of bug fixes C_i was subject to in the time period t between the r_j and the r_{j+1} release dates. This implies that the length of t could play a role in the change- and fault-proneness of classes (*i.e.,* the longer t the higher the class change- and fault-proneness). However, it is worth noting that:

1. This holds for both smelly and non-smelly classes, thus reducing the bias of t as a confounding factor.
2. To mitigate such a threat we completely re-run our analyses by considering a normalized version of class change- and fault-proneness. In particular, we computed the change-proneness of a class C_i in a release r_j as:

$$change\_proneness(C_i, r_j) = \frac{\#Changes(C_i)_{r_{j-1} \rightarrow r_j}}{\#Changes(r_{j-1} \rightarrow r_j)}$$

where $\#Changes(C_i)_{r_{j-1} \rightarrow r_j}$ is the number of changes performed to C_i by developers during the evolution of the system between the r_{j-1}'s and the

r_j's release dates and $\#Changes(r_{j-1} \rightarrow r_j)$ is the total number of changes performed on the whole system during the same time period. In a similar way, we computed the fault-proneness of a class C_i in a release r_j as:

$$fault\_proneness(C_i, r_j) = \frac{NOBF(C_i)_{r_{j-1} \rightarrow r_j}}{NOBF(r_{j-1} \rightarrow r_j)}$$

where $NOBF(C_i)_{r_{j-1} \rightarrow r_j}$ is the number of bug fixing activities performed on C_i by developers between the r_{j-1}'s and the r_j's release dates and $NOBF(r_{j-1} \rightarrow r_j)$ is the total number of bugs fixed in the whole system during the same time period.

The achieved results are reported in our online appendix (Palomba et al, 2017) and are consistent with those reported in Section 4.

In addition, we cannot exclude imprecisions in the measurement of the fault-proneness of classes due to misclassification of issues (*e.g.,* an enhancement classified as a bug) in the issue-tracking systems (Antoniol et al, 2008). At least, the systems we consider use an explicit classification of bugs, distinguishing them from other issues.

We relied on the SZZ algorithm (Sliwerski et al, 2005) to investigate whether there is a temporal relationship between the occurrence of a code smell and a bug induction. We are aware that such an algorithm only gives a rough approximation of the set of commits inducing a fix, because (i) the line-based differencing of git has intrinsic limitations, and (ii) in some cases a bug can be fixed without modifying the lines inducing it, *e.g.,* by adding a workaround or in general changing the control-flow elsewhere.

The main threats related to the relationship between the treatment and the outcome (*conclusion validity*) might be represented by the analysis method exploited in our study. We discussed our results by presenting descriptive statistics and using proper non-parametric correlation tests (p-values were properly adjusted when multiple comparisons were performed by applying the Holms correction procedure previously described). In addition, the practical relevance of the differences observed in terms of change- and fault-proneness is highlighted by effect size measures.

Threats to *internal validity* concern factors that could influence our observations. The fact that code smells disappear, may or may not be related to refactoring activities occurred between the observed releases. In other words, other changes might have produced such effects. We are aware that we cannot claim a direct cause-effect relation between the presence of code smells and fault- and change-proneness of classes, which can be influenced by several other factors. In particular, our observations may be influenced by the different development phases encountered over the change history as well as by developer-related factors (*e.g.,* experience and workload). Also, we acknowledge that such measures could simply reflect the "importance" of classes in the analyzed systems and in particular their central role in the software evolution process. For example, we expect classes controlling the business logic of a system to also be the ones more frequently modified by developers (high change-proneness) and

then possibly subject to the introduction of bugs (high fault-proneness). It is possible that such classes are also the ones more frequently affected by code smells, thus implying high change- and fault-proneness of smelly classes. An in-depth analysis of how such factors influence change- and fault-proneness of classes is part of our future agenda.

Finally, regarding the generalization of our findings (*external validity*) to the best of our knowledge this is the largest study—in terms of number of software releases (395), and considered code smell types (13)—on the diffuseness of code smells and their impact on maintainability properties. However, we are aware that we limited our attention only to Java systems, due to limitations of the infrastructure we used (*e.g.,* the code smell detection tool only works on Java code). Further studies aiming at replicating our work on systems written in other programming languages are desirable. Moreover, we focused on open-source systems only, and we cannot speculate about how the results would be different when analyzing industrial systems. Replications of the study in the context of industrial systems may be worthwhile in order to corroborate our findings.

6 Discussion and Conclusion

This paper reported a large study conducted on 395 releases of 30 Java open source projects, aimed at understanding the diffuseness of code smells in Java open source projects and their relation with source code change- and fault-proneness. The study considered 17,350 instances of 13 different code smell types, firstly detected using a metric-based approach and then manually validated.

The results highlighted the following findings:

- **Diffuseness of smells.** The most diffused smells are the one related to size and complexity such as *Long Method*, *Spaghetti Code*, and to some extent *Complex Class* or *God Class*. This seems to suggests that a simple metric-based monitoring of code quality could already give enough indications about the presence of poor design decisions or in general of poor code quality. Smells not related to size like *Message Chains* and *Lazy Class* are less diffused, although there are also cases of such smells with high diffuseness, see for example *Class Data Should Be Private* and *Speculative Generality*.
- **Relation with change- and fault-proneness.** Generally speaking, our results confirm the results of the previous study by Khomh et al (2012), *i.e.,* classes affected by code smells tend to be more change- and fault-prone than others, and that this is even more evident when classes are affected by multiple smells. At the same time, if we analyze the fault-proneness results for specific types of smells, we can also notice that high fault-proneness is particularly evident for smells such as *Message Chain* that are not highly diffused.

Table 10: Summary of the results achieved

Code Smell	Diffuseness	Removal Effect on Change-Proneness	Removal Effect on Fault-Proneness
Inappropriate Intimacy	**High**	**High**	Medium
Long Method	**High**	**High**	Limited
Spaghetti Code	**High**	**High**	Limited
Speculative Generality	**High**	**High**	Limited
God Class	Medium	**High**	Limited
Complex Class	Medium	**High**	Limited
Refused Bequest	Medium	**High**	Limited
Message Chain	Low	Medium	Limited
Feature Envy	Low	Limited	Medium
CDSBP	**High**	Limited	Limited
LPL	Medium	Limited	Limited
Lazy Class	Low	Limited	Limited
Middle Man	Low	Limited	Limited

– **Effect of smell removal on change- and fault-proneness.** Removing code smells is beneficial most of the times for the code change-proneness. On the other side, we found no substantial differences between the fault-proneness of classes in the periods when they were affected by smells and when they were not (*e.g.,* before the smell introduction, or after the smell removal). This partially contrast the results of previous studies (Khomh et al, 2012) and seems to indicate that the smell is not the direct cause of fault-proneness but rather a co-occurring phenomenon in some parts of the system that are intrinsically fault-prone for various reasons. This also confirms the principle that a class exhibiting faults in the past is still likely to exhibit faults in the future (Ostrand et al, 2005).

Our findings clearly show that code smells should be carefully monitored by programmers, since all of them are related to maintainability aspects such as change- and fault-proneness. Table 10 shows a summary of our findings, where we ranked the code smells based on the effect of their removal on change- and fault-proneness. Looking at the table we can see that the removal of seven highly diffused smells, *i.e., Inappropriate Intimacy*, *Long Method*, *Spaghetti Code*, *Speculative Generality*, *God Class*, *Complex Class*, and *Refused Bequest* provide a high benefit in terms of change-proneness: thus, on the one hand practitioners should carefully monitor these smells and plan refactoring actions to improve the overall maintainability of the code; on the other hand, researchers should focus on the construction of automatic tools able to identify and remove these smells.

The removal of other smells seems to be less relevant from a practical perspective, since it does not substantially help in improving the maintainability of the source code. Our results also suggest that developers might use code smell detectors as a way to locate portions of source code that need more testing activities.

As for our future research agenda, we will focus on the definition of recommenders able to alert developers about the presence of potential problematic classes based on their (evolution of) change- and fault-proneness and rank them based on the potential benefits provided by their removal. Moreover,

we plan to further analyze other factors influencing the change- and fault-proneness of classes.

References

Abbes M, Khomh F, Gueheneuc YG, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, CSMR '11, pp 181–190

Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2008), October 27-30, 2008, Richmond Hill, Ontario, Canada, p 23

Arcoverde R, Garcia A, Figueiredo E (2011) Understanding the longevity of code smells: preliminary results of an explanatory survey. In: Proceedings of the International Workshop on Refactoring Tools, ACM, pp 33–36

Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D (2012) An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012, pp 56–65

Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F (2015) An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software 107:1–14

Chatzigeorgiou A, Manakos A (2010) Investigating the evolution of bad smells in object-oriented code. In: Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology, IEEE Computer Society, QUATIC '10, pp 106–115

Cohen J (1988) Statistical power analysis for the behavioral sciences, 2nd edn. Lawrence Earlbaum Associates

Conover WJ (1998) Practical Nonparametric Statistics, 3rd edn. Wiley

Cunningham W (1993) The WyCash portfolio management system. OOPS Messenger 4(2):29–30, DOI 10.1145/157710.157715

D'Ambros M, Bacchelli A, Lanza M (2010) On the impact of design flaws on software defects. In: Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010, pp 23–31

Deligiannis I, Stamelos I, Angelis L, Roumeliotis M, Shepperd M (2004) A controlled experiment investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software 72(2):129 – 143

Fernandes E, Oliveira J, Vale G, Paiva T, Figueiredo E (2016) A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, ACM, New York, NY, USA, EASE '16, pp 18:1–18:12, DOI

10.1145/2915970.2915984, URL http://doi.acm.org/10.1145/2915970.2915984

Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley

Gatrell M, Counsell S (2015) The effect of refactoring on change and fault-proneness in commercial c# software. Science of Computer Programming 102(0):44 – 56

Gîrba T, Ducasse S, Kuhn A, Marinescu R, Daniel R (2007) Using concept analysis to detect co-change patterns. In: Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting, ACM, IWPSE '07, pp 83–89

Grissom RJ, Kim JJ (2005) Effect sizes for research: A broad practical approach, 2nd edn. Lawrence Earlbaum Associates

Holm S (1979) A simple sequentially rejective Bonferroni test procedure. Scandinavian Journal on Statistics 6:65–70

Hosmer Jr DW, Lemeshow S (2004) Applied logistic regression. John Wiley & Sons

Kessentini M, Vaucher S, Sahraoui H (2010) Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ACM, ASE '10, pp 113–122

Khomh F, Di Penta M, Guéhéneuc YG (2009a) An exploratory study of the impact of code smells on software change-proneness. In: 16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France, IEEE Computer Society, pp 75–84

Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2009b) A bayesian approach for the detection of code and design smells. In: Proceedings of the 9th International Conference on Quality Software, IEEE CS Press, Hong Kong, China, pp 305–314

Khomh F, Di Penta M, Guéhéneuc YG, Antoniol G (2012) An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empirical Software Engineering 17(3):243–275

Kruchten P, Nord RL, Ozkaya I (2012) Technical debt: From metaphor to theory and practice. IEEE Software 29(6):18–21, DOI 10.1109/MS.2012.167

Lanza M, Marinescu R (2010) Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer

Li W, Shatnawi R (2007) 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

Lopez M, Habra N (2015) Relevance of the cyclomatic complexity threshold for the java programming language. Software Measurement European Forum

Lozano A, Wermelinger M, Nuseibeh B (2007) Assessing the impact of bad smells using historical information. In: Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, ACM, New York, NY, USA, IWPSE '07, pp 31–34

Marinescu R (2004) Detection strategies: Metrics-based rules for detecting design flaws. In: 20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA, IEEE Computer Society, pp 350–359

Moha N, Guéhéneuc YG, Duchien L, Meur AFL (2010) Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering 36(1):20–36

Moonen L (2001) Generating robust parsers using island grammars. In: Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001, p 13

Munro MJ (2005) Product metrics for automatic identification of "bad smell" design problems in java source-code. In: Proceedings of the 11^{th} International Software Metrics Symposium, IEEE Computer Society Press

Olbrich S, Cruzes DS, Basili V, Zazworka N (2009) The evolution and impact of code smells: A case study of two open source systems. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM '09, pp 390–400

Olbrich SM, Cruzes D, Sjøberg DIK (2010) Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. In: 26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania, pp 1–10

Oliveto R, Khomh F, Antoniol G, Guéhéneuc YG (2010) Numerical signatures of antipatterns: An approach based on b-splines. In: Capilla R, Ferenc R, Dueas JC (eds) Proceedings of the 14^{th} Conference on Software Maintenance and Reengineering, IEEE Computer Society Press

Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Software Eng 31(4):340–355

Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A (2014) Do they really smell bad? a study on developers' perception of bad code smells. In: In Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14), Victoria, Canada, pp 101–110

Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A (2015a) Mining version histories for detecting code smells. Software Engineering, IEEE Transactions on 41(5):462–489, DOI 10.1109/TSE.2014.2372760

Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A (2015b) Mining version histories for detecting code smells. IEEE Trans Software Eng 41(5):462–489, DOI 10.1109/TSE.2014.2372760, URL `http://dx.doi.org/10.1109/TSE.2014.2372760`

Palomba F, De Lucia A, Bavota G, Oliveto R (2015c) Anti-pattern detection: Methods, challenges, and open issues. Advances in Computers 95:201–238, DOI 10.1016/B978-0-12-800160-8.00004-8

Palomba F, Di Nucci D, Panichella A, Oliveto R, De Lucia A (2016a) On the diffusion of test smells in automatically generated test code: An empirical study. In: Proceedings of the 9th International Workshop on Search-based Software Testing, SBST 2016

Palomba F, Panichella A, Lucia AD, Oliveto R, Zaidman A (2016b) A textual-based technique for smell detection. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp 1–10, DOI 10.1109/ICPC.2016.7503704

Palomba F, Bavota G, Oliveto R, Fasano F, Di Penta M, De Lucia A (2017) Bad code smells study - online appendix. URL `https://dibt.unimol.it/fpalomba/reports/badSmell-analysis/index.html`

Peters R, Zaidman A (2012) Evaluating the lifespan of code smells using software repository mining. In: 16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012, pp 411–416

Ratiu D, Ducasse S, Gîrba T, Marinescu R (2004) Using history information to improve design flaws detection. In: 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), 24-26 March 2004, Tampere, Finland, Proceeding, IEEE Computer Society, pp 223–232

Saboury A, Musavi P, Khomh F, Antoniol G (2017) An empirical study of code smells in javascript projects. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 294–305, DOI 10.1109/SANER.2017.7884630

Sahin D, Kessentini M, Bechikh S, Deb K (2014) Code-smell detection as a bilevel problem. ACM Trans Softw Eng Methodol 24(1):6:1–6:44, DOI 10.1145/2675067, URL `http://doi.acm.org/10.1145/2675067`

Sjoberg D, Yamashita A, Anda B, Mockus A, Dyba T (2013) Quantifying the effect of code smells on maintenance effort. Software Engineering, IEEE Transactions on 39(8):1144–1156, DOI 10.1109/TSE.2012.89

Sliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, ACM

Student (1921) An experimental determination of the probable error of dr spearman's correlation coefficients. Biometrika 13(2/3):263–282, URL `http://www.jstor.org/stable/2331754`

Tsantalis N, Chatzigeorgiou A (2009) Identification of move method refactoring opportunities. IEEE Transactions on Software Engineering 35(3):347–367

Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2015) When and why your code starts to smell bad. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, pp 403–414

Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2016) An empirical investigation into the nature of test smells. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE 2016, pp 4–15

Tufano M, Palomba F, Bavota G, Oliveto R, Penta MD, Lucia AD, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering PP(99):1–1,

DOI 10.1109/TSE.2017.2653105
Vaucher S, Khomh F, Moha N, Gueheneuc YG (2009) Tracking design smells: Lessons from a study of god classes. In: Proceedings of the 2009 16th Working Conference on Reverse Engineering (WCRE'09), pp 145–158
Yamashita AF, Moonen L (2012) Do code smells reflect important maintainability aspects? In: 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012, pp 306–315
Yamashita AF, Moonen L (2013) Exploring the impact of inter-smell relations on software maintainability: an empirical study. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pp 682–691

Appendix

Table 11 shows the diffuseness of the analyzed code smells in the subject systems.

Table 11: Code smell diffuseness in the subject systems: min-max instances in the systems' releases.

System	CDSBP	Complex Class	Feature Envy	God Class	Inappropriate Intimacy	Lazy Class	Long Method	LPL	Message Chain	Middle Man	Refused Bequest	Spaghetti Code	Speculative Generality
ArgoUML	5-19 (0.6-1.3)	4-10 (0.5-0.7)	1-2 (0.1-0.1)	2-6 (0.2-0.4)	0-8 (0.0-0.5)	0-0 (0.0-0.0)	18-30 (2.3-2.1)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	1-2 (0.1-0.1)	0-4 (0.0-0.3)	14-21 (1.8-1.5)	12-29 (1.5-2.0)
Ant	1-7 (1.2-0.9)	0-4 (0.0-0.5)	0-6 (0.0-0.9)	0-6 (0.0-0.7)	2-22 (2.4-2.7)	0-0 (0.0-0.0)	6-38 (7.2-4.6)	0-17 (0.0-2.0)	0-3 (0.0-0.4)	0-2 (0.0-0.2)	0-14 (0.0-1.7)	4-30 (5.0-3.7)	1-4 (1.2-0.5)
aTunes	3-12 (2.0-1.8)	0-1 (0.0-0.2)	0-6 (0.0-0.9)	0-1 (0.0-0.2)	0-8 (0.0-1.2)	1-9 (0.7-1.4)	6-31 (4.3-3.2)	0-11 (0.0-1.7)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	1-8 (0.7-1.3)	6-21 (4.3-3.4)	0-10 (0.0-1.6)
Cassandra	5-14 (1.6-2.4)	0-4 (0.0-0.6)	0-2 (0.0-0.3)	0-2 (0.0-0.4)	2-24 (0.0-4.0)	0-1 (0.0-0.2)	3-22 (1.0-4.0)	0-16 (0.0-2.7)	0-2 (0.0-0.3)	2-8 (0.7-1.4)	0-2 (0.0-0.3)	0-6 (0.0-0.9)	0-16 (0.0-2.9)
Derby	20-40 (1.3-1.0)	21-25 (1.5-1.3)	1-1 (0.6-0.5)	20-26 (1.3-1.4)	0-0 (0.0-0.0)	1-1 (0.4-0.5)	176-212 (0.8-0.8)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	2-2 (0.1-0.1)	10-19 (0.7-0.9)	12-16 (0.8-0.8)	19-26 (1.3-1.4)
Eclipse Core	15-32 (2.1-2.7)	8-35 (1.1-2.9)	0-6 (0.0-0.5)	3-15 (0.4-1.3)	0-16 (0.0-1.4)	2-17 (0.2-1.4)	36-180 (4.8-15.2)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	2-2 (0.3-0.2)	7-31 (0.9-2.6)	12-25 (1.6-2.1)	4-15 (0.5-1.8)
Elastic Search	3-5 (0.2-0.2)	0-5 (0.0-0.2)	0-0 (0.0-0.0)	1-3 (0.1-0.1)	4-4 (0.2-0.1)	4-7 (0.2-0.3)	11-27 (0.7-1.9)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-3 (0.0-0.1)	3-8 (0.2-0.4)	3-9 (0.2-0.4)
FreeMind	0-5 (0.0-0.9)	0-6 (0.0-1.2)	0-1 (0.0-0.2)	0-2 (0.0-0.4)	0-6 (0.0-0.7)	0-3 (0.0-0.6)	0-13 (0.0-2.6)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-1 (0.0-0.1)	0-3 (0.0-0.6)	0-9 (0.0-1.9)	0-2 (0.0-0.4)
Hadoop	0-3 (0.0-0.1)	0-2 (0.0-0.1)	0-4 (0.0-0.1)	0-2 (0.0-0.1)	2-10 (1.6-0.1)	3-9 (2.3-0.1)	5-17 (3.8-0.1)	0-12 (0.0-0.1)	0-0 (0.0-0.1)	0-1 (0.0-0.1)	0-0 (0.0-0.1)	6-7 (4.6-0.1)	3-5 (2.5-0.1)
HSQLDB	0-7 (0.0-1.5)	0-5 (0.0-1.1)	0-3 (0.0-0.7)	0-11 (0.0-1.5)	8-24 (14.8-5.4)	0-9 (0.0-2.1)	10-124 (1.4-14.1)	0-13 (0.0-2.9)	0-4 (0.0-0.9)	0-0 (0.0-0.0)	0-13 (0.0-0.2.8)	2-29 (3.7-6.5)	0-3 (0.0-0.7)
Hbase	5-16 (3.1-2.3)	2-7 (1.2-1.0)	1-9 (0.7-1.3)	1-8 (0.7-1.1)	2-14 (1.2-2.0)	2-21 (1.4-3.1)	12-42 (7.5-6.4)	3-45 (1.9-6.5)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-22 (0.0-3.2)	3-5 (1.8-0.7)	2-10 (1.3-1.4)
Hibernate	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)
Hive	3-7 (0.7-0.6)	1-3 (0.1-0.3)	4-11 (0.9-0.9)	0-2 (0.0-0.2)	0-8 (0.0-0.7)	1-1 (0.1-0.1)	11-83 (2.7-7.4)	14-77 (3.4-6.9)	0-1 (0.0-0.1)	0-1 (0.0-0.1)	4-6 (0.9-0.5)	2-4 (0.5-0.4)	6-31 (1.5-2.8)
Incubating	12-16 (4.8-5.0)	6-6 (2.5-1.9)	3-10 (1.1-3.1)	6-6 (2.8-1.7)	12-30 (4.8-9.5)	1-4 (0.4-1.3)	89-110 (3.5-3.5)	27-35 (10.8-11.1)	0-0 (0.0-0.0)	0-1 (0.0-0.3)	10-17 (4.0-5.3)	6-8 (2.4-2.5)	6-8 (2.3-2.5)
Ivy	1-1 (0.4-0.3)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-2 (0.0-0.6)	0-6 (0.0-1.7)	0-0 (0.0-0.0)	2-22 (0.7-6.3)	13-21 (4.6-6.0)	0-0 (0.0-0.0)	1-1 (0.3-0.3)	0-0 (0.0-0.0)	1-4 (0.4-1.2)	4-5 (1.4-1.4)
Lucene	39-47 (2.2-2.0)	3-5 (0.2-0.2)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	8-14 (0.5-0.6)	0-10 (0.0-0.5)	61-74 (3.5-3.2)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	7-9 (0.4-0.4)	10-16 (0.6-0.7)	29-38 (1.6-1.7)
JEdit	0-7 (0.0-1.3)	4-21 (1.8-4.0)	0-2 (0.0-0.4)	0-6 (0.0-1.2)	0-8 (0.0-1.5)	0-0 (0.0-0.0)	8-33 (3.5-6.4)	0-9 (0.0-1.7)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-3 (0.0-0.6)	3-18 (1.3-3.5)	4-14 (1.8-2.7)
JHotDraw	0-0 (0.0-0.0)	0-4 (0.0-0.6)	0-0 (0.0-0.0)	0-2 (0.0-0.3)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)
JFreeChart	0-9 (0.0-1.2)	0-3 (0.0-0.4)	0-0 (0.0-0.0)	0-9 (0.0-1.4)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	2-63 (2.3-8.1)	8-67 (9.3-8.6)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	1-3 (1.1-0.4)	2-5 (2.3-0.6)
JBoss	18-65 (0.8-1.4)	9-23 (0.4-0.5)	0-1 (0.0-0.1)	1-16 (0.1-0.4)	0-4 (0.0-0.1)	0-6 (0.1-0.0)	45-135 (1.9-2.8)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-3 (0.0-0.1)	24-55 (1.0-1.1)	23-54 (1.0-1.1)	23-65 (1.0-1.4)
JVlt	0-2 (0.0-0.9)	0-0 (0.0-0.0)	0-4 (0.0-1.8)	0-1 (0.0-0.0)	2-4 (1.2-0.5)	1-5 (0.7-2.2)	5-7 (3.0-3.1)	0-2 (0.0-0.9)	0-0 (0.0-0.0)	0-1 (0.0-0.5)	1-3 (0.7-1.4)	3-4 (1.8-1.8)	1-3 (0.7-1.3)
jSL	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)
Karaf	0-2 (0.0-0.4)	0-0 (0.0-0.0)	1-1 (0.4-0.2)	0-2 (0.0-0.3)	0-2 (0.0-0.3)	0-0 (0.0-0.0)	1-5 (0.5-1.1)	0-2 (0.0-0.3)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	5-9 (2.0-1.9)	2-2 (2.1-0.4)
Nutch	3-12 (1.6-4.6)	0-0 (0.0-0.0)	1-5 (0.5-1.9)	0-0 (0.0-0.0)	4-12 (2.1-4.9)	1-9 (0.6-3.5)	7-17 (3.8-6.6)	0-16 (0.0-6.1)	0-0 (0.0-0.0)	0-6 (0.0-6.3)	0-2 (0.0-0.7)	0-4 (0.0-1.5)	0-3 (0.0-1.2)
Pig	3-5 (1.2-0.5)	0-7 (0.0-0.8)	0-1 (0.0-0.12)	0-3 (0.0-0.3)	4-10 (1.6-1.1)	0-1 (0.0-0.1)	0-43 (0.0-4.7)	0-3 (0.0-0.3)	0-1 (0.0-0.1)	0-2 (0.0-0.2)	0-13 (0.0-1.4)	0-7 (0.0-0.8)	4-20 (1.7-2.2)
Qpid	11-18 (1.1-1.9)	4-10 (0.4-1.0)	0-2 (0.0-0.2)	4-6 (0.6-0.7)	4-10 (0.4-1.1)	0-1 (0.0-0.1)	21-33 (2.1-3.6)	22-39 (2.2-4.2)	0-1 (0.0-0.1)	0-1 (0.0-0.1)	1-8 (0.1-0.9)	3-8 (0.4-0.8)	32-38 (3.3-4.1)
Sax	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)
Struts	7-12 (1.1-1.1)	0-4 (0.0-0.4)	0-1 (0.0-0.1)	0-2 (0.0-0.3)	6-12 (0.9-1.0)	0-0 (0.0-0.0)	3-14 (0.5-1.4)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	1-2 (0.2-0.2)	3-9 (0.5-0.9)	0-6 (0.0-0.6)
Wicket	0-0 (0.0-0.0)	2-2 (0.3-0.2)	0-0 (0.0-0.0)	4-4 (0.5-0.5)	4-6 (0.5-0.7)	0-0 (0.0-0.0)	4-4 (0.5-0.5)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	0-0 (0.0-0.0)	3-7 (0.4-0.8)	11-11 (1.4-1.3)	18-18 (2.7-2.2)
Xerces	10-42 (6.0-5.7)	4-10 (2.4-1.4)	0-17 (0.0-2.3)	5-11 (3.0-1.5)	2-34 (1.2-4.6)	0-4 (0.0-0.5)	48-123 (2.7-6.9)	4-29 (2.5-4.0)	2-3 (1.3-0.4)	0-0 (0.0-0.0)	0-15 (0.0-2.0)	4-9 (2.4-1.3)	2-11 (1.2-1.5)
Overall	**0-65 (0.0-5.0)**	**0-35 (0.0-4.0)**	**0-17 (0.0-3.1)**	**0-26 (0.0-1.7)**	**0-34 (0.0-9.5)**	**0-21 (0.0-3.5)**	**0-212 (0.0-15.2)**	**0-77 (0.0-11.0)**	**0-4 (0.0-0.9)**	**0-8 (0.0-6.3)**	**0-55 (0.0-3.2)**	**0-54 (0.0-6.5)**	**0-65 (0.0-4.1)**