When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)

Michele Tufano[1], Fabio Palomba[2,3], Gabriele Bavota[4]
Rocco Oliveto[5], Massimiliano Di Penta[6], Andrea De Lucia[3], Denys Poshyvanyk[1]

[1]The College of William and Mary, Williamsburg, USA, [2]Delft University of Technology, The Netherlands
[3]University of Salerno, Fisciano (SA), Italy, [4]Università della Svizzera italiana (USI), Switzerland,
[5]University of Molise, Pesche (IS), Italy, [6]University of Sannio, Benevento (BN), Italy

mtufano@email.wm.edu, fpalomba@unisa.it, gabriele.bavota@usi.ch
rocco.oliveto@unimol.it, dipenta@unisannio.it, adelucia@unisa.it, denys@cs.wm.edu

Abstract—Technical debt is a metaphor introduced by Cunningham to indicate "not quite right code which we postpone making it right". One noticeable symptom of technical debt is represented by code smells, defined as symptoms of poor design and implementation choices. Previous studies showed the negative impact of code smells on the comprehensibility and maintainability of code. While the repercussions of smells on code quality have been empirically assessed, there is still only anecdotal evidence on *when* and *why* bad smells are introduced, what is their *survivability*, and *how* they are *removed* by developers. To empirically corroborate such anecdotal evidence, we conducted a large empirical study over the change history of 200 open source projects. This study required the development of a strategy to identify smell-introducing commits, the mining of over half a million of commits, and the manual analysis and classification of over 10K of them. Our findings mostly contradict common wisdom, showing that most of the smell instances are introduced when an artifact is created and not as a result of its evolution. At the same time, 80% of smells survive in the system. Also, among the 20% of removed instances, only 9% is removed as a direct consequence of refactoring operations.

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

1 Introduction

The technical debt metaphor introduced by Cunningham [23] well explains the trade-offs between delivering the most appropriate but still immature product, in the shortest time possible [14], [23], [43], [48], [71]. Bad code smells (shortly "code smells" or "smells"), *i.e.*, symptoms of poor design and implementation choices [28], represent one important factor contributing to technical debt, and possibly affecting the maintainability of a software system [43]. In the past and, most notably, in recent years, several studies investigated the relevance that code smells have for developers [61], [91], the extent to which code smells tend to remain in a software system for long periods of time [4], [18], [49], [65], as well as the side effects of code smells, such as an increase in change- and fault-proneness [38], [39] or decrease of software understandability [1] and maintainability [73], [90], [89]. While the repercussions of code smells on software quality have been empirically proven, there is still noticeable lack of empirical evidence related to how, when, and why they occur in software projects, as well as whether, after how long, and how they are removed [14]. This represents an obstacle for an effective and efficient management of technical debt. Also, understanding the typical life-cycle of code smells and the actions undertaken by developers to remove them is of paramount importance in the conception of recommender tools for developers' support. In other words, only a proper undertanding of the phenomenon would allow the creation of recommenders able to highlight the presence of code smells and suggesting refactorings only when appropriate, hence avoiding information overload for developers [54].

Common wisdom suggests that urgent maintenance activities and pressure to deliver features while prioritizing time-to-market over code quality are often the causes of such smells. Generally speaking, software evolution has always been considered as one of the reasons behind "software aging" [62] or "increasing complexity" [45], [56], [88]. Also, one of the common beliefs is that developers remove code smells from the system by performing refactoring operations. However, to the best of our knowledge, there is no comprehensive empirical investigation into *when* and *why* code smells are introduced in software projects, how long they *survive*, and *how they are removed*.

In this paper we fill the void in terms of our understanding of code smells, reporting the results of a large-scale empirical study conducted on the change history

This paper is an extension of "When and Why Your Code Starts to Smell Bad" that appeared in the Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015), Florence, Italy, pp. 403-414, 2015 [82].

of 200 open source projects belonging to three software ecosystems, namely Android, Apache and Eclipse. The study aims at investigating (i) *when* smells are introduced in software projects, (ii) *why* they are introduced (*i.e.,* under what circumstances smell introductions occur and who are the developers responsible for introducing smells), (iii) *how long they survive* in the system, and (iv) *how they are removed*. To address these research questions, we developed a metric-based methodology for analyzing the evolution of code entities in change histories of software projects to determine when code smells start manifesting themselves and whether this happens suddenly (*i.e.,* because of a pressure to quickly introduce a change), or gradually (*i.e.,* because of medium-to-long range design decisions). We mined over half a million of commits and we manually analyzed over 10K of them to understand how code smells are introduced and removed from software systems. We are unaware of any published technical debt, in general, and code smells study, in particular, of comparable size. The obtained results allowed us to report quantitative and qualitative evidence on when and why smells are introduced and removed from software projects as well as implications of these results, often contradicting common wisdom. In particular, our main findings show that (i) most of the code smells are introduced when the (smelly) code artifact is created in the first place, and not as the result of maintenance and evolution activities performed on such an artifact, (ii) 80% of code smells, once introduced, are not removed by developers, and (iii) the 20% of removed code smells are very rarely (in 9% of cases) removed as a direct consequence of refactoring activities.

The paper makes the following notable contributions:

1) *A methodology for identifying smell-introducing changes*, namely a technique able to analyze change history information in order to detect the commit, which introduced a code smell;
2) *A large-scale empirical study involving three popular software ecosystems* aimed at reporting quantitative and qualitative evidence on when and why smells are introduced in software projects, what is their survaivability, and how code smells are removed from the source code, as well as implications of these results, often contradicting common wisdom.
3) *A publicly available comprehensive dataset* [81] that enables others to conduct further similar or different empirical studies on code smells (as well as completely reproducing our results).

Implications of the study. From a purely empirical point of view, the study aims at confirming and/or contradicting the common wisdom about software evolution and manifestation of code smells. From a more practical point of view, the results of this study can help distinguish among different situations that can arise in software projects, and in particular in cases where:

- Smells are introduced when a (sub) system has been conceived. Certainly, in such cases smell detectors can help identify potential problems, although this situation can trigger even more serious alarms related to potentially poor design choices made in the system since its inception (*i.e.,* technical debt that smell detectors will not be able to identify from a system's snapshot only), that may require careful redesign in order to avoid worse problems in future.
- Smells occur suddenly in correspondence to a given change, pointing out cases for which recommender systems may warn developers of emergency maintenance activities being performed and the need to consider refactoring activities whenever possible.
- The symptom simply highlights—as also pointed out in a previous study [61], [91]—the intrinsic complexity, size (or any other smell-related characteristics) of a code entity, and there is little or nothing one can do about that. Often some situations that seem to fall in the two cases above should be considered in this category instead.
- Smells manifest themselves gradually. In such cases, smell detectors can identify smells only when they actually manifest themselves (*e.g.,* some metrics go above a given threshold) and suggest refactoring actions. Instead, in such circumstances, tools monitoring system evolution and identifying metric trends, combined with history-based smell detectors [59], should be used.

In addition, our findings, which are related to the very limited refactoring actions undertaken by developers to remove code smells, call for further studies aimed at understanding the reasons behind this result. Indeed, it is crucial for the research community to study and understand whether:

- developers perceive (or don't) the code smells as harmful, and thus they simply do not care about removing them from the system; and/or
- developers consider the cost of refactoring code smells too high when considering possible side effects (*e.g.,* bug introduction [9]) and expected benefits; and/or
- the available tools for the identification/refactoring of code smells are not sufficient/effective/usable from the developers' perspective.

Paper structure. Section 2 describes the study design, while Section 3 and Section 4 report the study results and discuss the threats to validity, respectively. Following the related work (Section 5), Section 6 concludes the paper outlining lessons learned and promising directions for future work.

2 Study Design

The *goal* of the study is to analyze the change history of software projects with the *purpose* of investigating when code smells are introduced and fixed by developers and the circumstances and reasons behind smell appearances.

TABLE 1: Characteristics of ecosystems under analysis.

Ecosystem	#Proj.	#Classes	KLOC	#Commits	#Issues	Mean Story Length	Min-Max Story Length
Apache	100	4-5,052	1-1,031	207,997	3,486	6	1-15
Android	70	5-4,980	3-1,140	107,555	1,193	3	1-6
Eclipse	30	142-16,700	26-2,610	264,119	124	10	1-13
Overall	**200**	-	-	**579,671**	**4,803**	**6**	**1-15**

More specifically, the study aims at addressing the following four research questions (RQs):

- **RQ$_1$:** *When are code smells introduced?* This research question aims at investigating to what extent the common wisdom suggesting that "*code smells are introduced as a consequence of continuous maintenance and evolution activities performed on a code artifact*" [28] applies. Specifically, we study "when" code smells are introduced in software systems, to understand whether smells are introduced as soon as a code entity is created, whether smells are suddenly introduced in the context of specific maintenance activities, or whether, instead, smells appear "gradually" during software evolution. To this aim, we investigated the presence of possible trends in the history of code artifacts that characterize the introduction of specific types of smells.
- **RQ$_2$:** *Why are code smells introduced?* The second research question aims at empirically investigating under which circumstances developers are more prone to introduce code smells. We focus on factors that are indicated as possible causes for code smell introduction in the existing literature [28]: the *commit goal* (*e.g.,* is the developer implementing a new feature or fixing a bug?), the *project status* (*e.g.,* is the change performed in proximity to a major release deadline?), and the *developer status* (*e.g.,* a newcomer or a senior project member?).
- **RQ$_3$:** *What is the survivability of code smells?* In this research question we aim to investigate how long a smell remains in the code. In other words, we want to study the *survivability* of code smells, that is the probability that a code smell instance survives over time. To this aim, we employ a statistical method called survival analysis [67]. In this research question, we also investigate differences of survivability among different types of code smells.
- **RQ$_4$:** *How do developers remove code smells?* The fourth and last research question aims at empirically investigating whether and how developers remove code smells. In particular, we want to understand whether code smells are removed using the expected and suggested refactoring operations for each specific type of code smell (as suggested by Fowler [28]), whether they are removed using "unexpected refactorings", or whether such a removal is a side effect of other changes. To achieve this goal, we manually analyzed 979 commits removing code smells by following an open coding process inspired by grounded theory [22].

2.1 Context Selection

The *context* of the study consists of the change history of 200 projects belonging to three software ecosystems, namely Android, Apache, and Eclipse. Table 1 reports for each of them (i) the number of projects analyzed, (ii) size ranges in terms of the number of classes and KLOC, (iii) the overall number of commits and issues analyzed, and (iv) the average, minimum, and maximum length of the projects' history (in years) analyzed in each ecosystem. All the analyzed projects are hosted in *Git* repositories and have associated issue trackers.

The Android ecosystem contains a random selection of 70 open source apps mined from the F-Droid[1] forge. The Apache ecosystem consists of 100 Java projects randomly selected among those available[2]. Finally, the Eclipse ecosystem consists of 30 projects randomly mined from the list of GitHub repositories managed by the Eclipse Foundation[3]. The choice of the ecosystems to analyze is not random, but rather driven by the motivation to consider projects having (i) different sizes, *e.g.,* Android apps are by their nature smaller than projects in Apache's and Eclipse's ecosystems, (ii) different architectures, *e.g.,* we have Android mobile apps, Apache libraries, and plug-in based architectures in Eclipse projects, and (iii) different development bases, *e.g.,* Android apps are often developed by small teams whereas several Apache projects are carried out by dozens of developers [8]. Also, we limited our study to 200 projects since, as it will be shown later, the analysis we performed is not only computationally expensive, but also requires the manual analysis of thousands of data points. To sum up, we mined 579,671 commits and 4,803 issues.

We focus our study on the following types of smells:

1) *Blob Class*: a large class with different responsibilities that monopolizes most of the system's processing [15];
2) *Class Data Should be Private*: a class exposing its attributes, violating the information hiding principle [28];
3) *Complex Class*: a class having a high cyclomatic complexity [15];
4) *Functional Decomposition*: a class where inheritance and polymorphism are poorly used, declaring many private fields and implementing few methods [15];
5) *Spaghetti Code*: a class without structure that declares long methods without parameters [15].

1. https://f-droid.org/
2. https://projects.apache.org/indexes/quick.html
3. https://github.com/eclipse

While several other smells exist in the literature [15], [28], we need to limit our analysis to a subset due to computational constraints. However, we carefully keep a mix of smells related to complex/large code components (*e.g.*, Blob Class, Complex Class) as well as smells related to the lack of adoption of good Object-Oriented coding practices (*e.g.*, Class Data Should be Private, Functional Decomposition). Thus, the considered smells are representative of the categories of smells investigated in previous studies (see Section 5).

2.2 Data Extraction and Analysis

This subsection describes the data extraction and analysis process that we followed to answer our research questions.

2.2.1 When are code smells introduced?

To answer $\mathbf{RQ}_1$ we firstly clone the 200 *Git* repositories. Then, we analyze each repository r_i using a tool that we developed (named as `HistoryMiner`), with the purpose of identifying smell-introducing commits. Our tool mines the entire change history of r_i, checks out each commit in chronological order, and runs an implementation of the *DECOR* smell detector based on the original rules defined by Moha *et al.* [51]. *DECOR* identifies smells using detection rules based on the values of internal quality metrics[4]. The choice of using *DECOR* is driven by the fact that (i) it is a state-of-the-art smell detector having a high accuracy in detecting smells [51]; and (ii) it applies simple detection rules that allow it to be very efficient. Note that we ran *DECOR* on all source code files contained in r_i only for the first commit of r_i. In the subsequent commits *DECOR* has been executed only on code files added or modified in each specific commit to save computational time. As an output, our tool produces, for each source code file $f_j \in r_i$ the list of commits in which f_j has been involved, specifying if f_j has been added, deleted, or modified and if f_j was affected in that specific commit, by one of the five considered smells.

Starting from the data generated by the `HistoryMiner` we compute, for each type of smell ($smell_k$) and for each source code file (f_j), the number of commits performed on f_j since the first commit involving f_j and adding the file to the repository, up to the commit in which *DECOR* detects that f_j is affected by $smell_k$. Clearly, such numbers are only computed for files identified as affected by the specific $smell_k$.

When analyzing the number of commits needed for a smell to affect a code component, we can have two possible scenarios. In the first scenario, smell instances are introduced during the creation of source code artifacts, *i.e.*, in the first commit involving a source code file. In the second scenario, smell instances are introduced after several commits and, thus, as a result of multiple maintenance activities. For the latter scenario, besides running the *DECOR* smell detector for the project snapshot related to each commit, the `HistoryMiner` also computes, for each snapshot and for each source code artifact, a set of quality metrics (see Table 2). As done for *DECOR*, quality metrics are computed for all code artifacts only during the first commit, and updated at each subsequent commit for added and modified files. The purpose of this analysis is to understand whether the trend followed by such metrics differ between files affected by a specific type of smell and files not affected by such a smell. For example, we expect that classes becoming Blobs will exhibit a higher growth rate than classes that are not going to become Blobs.

In order to analyze the evolution of the quality metrics, we need to identify the function that best approximates the data distribution, *i.e.*, the values of the considered metrics computed in a sequence of commits. We found that the best model is the linear function (more details are available in our technical report [81]). Note that we only consider linear regression models using a single metric at a time (*i.e.*, we did not consider more than one metric in the same regression model) since our interest is to observe how a single metric in isolation describes the smell-introducing process. We consider the building of more complex regression models based on more than one metric as part of our future work.

Having identified the model to be used, we compute, for each file $f_j \in r_i$, the regression line of its quality metric values. If file f_j is affected by a specific $smell_k$, we compute the regression line considering the quality metric values computed for each commit involving f_j from the first commit (*i.e.*, where the file was added to the versioning system) to the commit where the instance of $smell_k$ was detected in f_j. Instead, if f_j is not affected by any smell, we consider only the first n^{th} commits involving the file f_j, where n is the average number of commits required by $smell_k$ to affect code instances. Then, for each metric reported in Table 2, we compare the distributions of regression line slopes for smell-free and smelly files. The comparison is performed using a two-tailed Mann-Whitney U test [21]. The results are intended as statistically significant at $\alpha = 0.05$. We also estimate the magnitude of the observed differences using the Cliff's Delta (or d), a non-parametric effect size measure [32] for ordinal data. We follow the guidelines in [32] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Overall, the data extraction for $\mathbf{RQ}_1$ (*i.e.*, the smells detection and metrics computation at each commit for the 200 systems) took eight weeks on a Linux server having 7 quad-core 2.67 GHz CPU (28 cores) and 24 Gb of RAM.

2.2.2 Why are code smells introduced?

One challenge arising when answering $\mathbf{RQ}_2$ is represented by the identification of the specific commit (or

4. An example of detection rule exploited to identify Blob classes can be found at http://tinyurl.com/paf9gp6.

TABLE 2: Quality metrics measured in the context of $\mathbf{RQ}_1$.

Metric	Description
Lines of Code (LOC)	The number of lines of code excluding white spaces and comments
Weighted Methods per Class (WMC) [19]	The complexity of a class as the sum of the McCabe's cyclomatic complexity of its methods
Response for a Class (RFC) [19]	The number of distinct methods and constructors invoked by a class
Coupling Between Object (CBO) [19]	The number of classes to which a class is coupled
Lack of COhesion of Methods (LCOM) [19]	The higher the pairs of methods in a class sharing at least a field, the higher its cohesion
Number of Attributes (NOA)	The number of attributes in a class
Number of Methods (NOM)	The number of methods in a class

TABLE 3: Tags assigned to the smell-introducing commits.

Tag	Description	Values
	COMMIT GOAL TAGS	
Bug fixing	The commit aimed at fixing a bug	[true,false]
Enhancement	The commit aimed at implementing an enhancement in the system	[true,false]
New feature	The commit aimed at implementing a new feature in the system	[true,false]
Refactoring	The commit aimed at performing refactoring operations	[true,false]
	PROJECT STATUS TAGS	
Working on release	The commit was performed [value] before the issuing of a major release	[one day, one week, one month, more than one month]
Project startup	The commit was performed [value] after the starting of the project	[one week, one month, one year, more than one year]
	DEVELOPER STATUS TAGS	
Workload	The developer had a [value] workload when the commit has been performed	[low,medium,high]
Ownership	The developer was the owner of the file in which the commit introduced the smell	[true,false]
Newcomer	The developer was a newcomer when the commit was performed	[true,false]

also possibly a set of commits) where the smell has been introduced (from now on referred to as a *smell-introducing commit*). Such information is crucial to explain under which circumstances these commits were performed. A trivial solution would have been to use the results of our $\mathbf{RQ}_1$ and consider the commit c_s in which *DECOR* detects for the first time a smell instance $smell_k$ in a source code file f_j as a commit-introducing smell in f_j. However, while this solution would work for smell instances that are introduced in the first commit involving f_j (there is no doubt on the commit that introduced the smell), it would not work for smell instances that are the consequence of several changes, performed in n different commits involving f_j. In such a situation, on one hand, we cannot simply assume that the first commit in which *DECOR* identifies the smell is the one introducing that smell, because the smell appearance might be the result of several small changes performed across the n commits. On the other hand, we cannot assume that all n commits performed on f_j are those (gradually) introducing the smell, since just some of them might have pushed f_j toward a smelly direction. Thus, to identify the smell-introducing commits for a file f_j affected by an instance of a smell ($smell_k$), we use the following heuristic:

- **if** $smell_k$ has been introduced in the commit c_1 where f_j has been added to the repository, **then** c_1 is the smell-introducing commit;
- **else** given $C = \{c_1, c_2, \ldots, c_n\}$ the set of commits involving f_j and leading to the detection of $smell_k$ in c_n we use the results of $\mathbf{RQ}_1$ to select the set of quality metrics M allowing to discriminate between the groups of files that are affected and not affected in their history by $smell_k$. These metrics are those for which we found statistically significant difference between the slope of the regression lines for the two groups of files accompanied by at least a medium effect size. Let s be the slope of the regression line for the metric $m \in M$ built when considering all commits leading f_j to become affected by a smell and s_i the slope of the regression line for the metric m built when considering just two subsequent commits, *i.e.*, c_{i-1} and c_i for each $i \in [2, ..., n]$. A commit $c_i \in C$ is considered as a smell-introducing commit if $|s_i| > |s|$, *i.e.*, the commit c_i significantly contributes to the increment (or decrement) of the metric m.

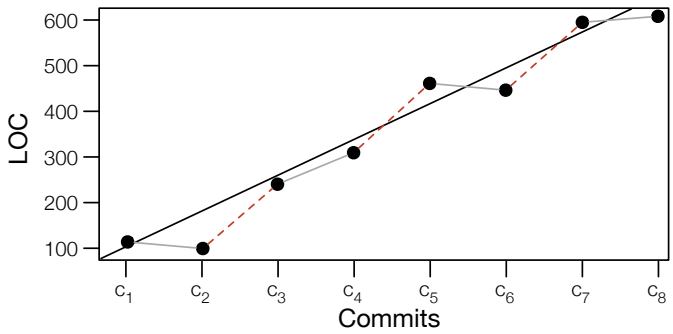


Fig. 1: Example of identifying smell-introducing commits.

Fig. 1 reports an example aimed at illustrating the smell-introducing commits identification for a file f_j. Suppose that f_j has been involved in eight commits (from c_1 to c_8), and that in c_8 a Blob instance has been identified by *DECOR* in f_j. Also, suppose that the results of our $\mathbf{RQ}_1$ showed that the LOC metric is the only one

"characterizing" the Blob introduction, *i.e.*, the slope of the LOC regression line for Blobs is significantly different than the one of the regression line built for classes which are not affected by the Blob smell. The black line in Fig. 1 represents the LOC regression line computed among all the involved commits, having a slope of 1.3. The gray lines represent the regression lines between pairs of commits (c_{i-1}, c_i), where c_i is not classified as a smell-introducing commit (their slope is lower than 1.3). Finally, the red-dashed lines represent the regression lines between pairs of commits (c_{i-1}, c_i), where c_i is classified as a smell-introducing commit (their slope is higher than 1.3). Thus, the smell-introducing commits in the example depicted in Fig. 1 are: c_3, c_5, and c_7. Overall, we obtained 9,164 smell-introducing commits in 200 systems, that we used to answer $\mathbf{RQ}_2$.

After having identified smell-introducing commits, with the purpose of understanding *why* a smell was introduced in a project, we classify them by assigning to each commit one or more tags among those reported in Table 3. The first set of tags (*i.e.*, commit goal tags) aims at explaining *what the developer was doing when introducing the smell*. To assign such tags we firstly download the issues for all 200 projects from their JIRA or BUGZILLA issue trackers. Then, we check whether any of the 9,164 smell-introducing commits were related to any of the collected issues. To link issues to commits we used (and complemented) two existing approaches. The first one is the regular expression-based approach by Fischer *et al.* [26] matching the issue ID in the commit note. The second one is a re-implementation of the *ReLink* approach proposed by Wu *et al.* [87], which considers the following constraints: (i) matching the committer/authors with issue tracking contributor name/email; (ii) the time interval between the commit and the last comment posted by the same author/contributor on the issue tracker must be less than seven days; and (iii) the Vector Space Model (VSM) [6] cosine similarity between the commit note and the last comment referred above must be greater than 0.7. RELINK has been shown to accurately link issues and commits (89% for precision and 78% for recall) [87]. When it was possible to identify a link between one of the smell-introducing commits and an issue, and the issue type was one of the goal-tags in our design (*i.e.*, bug, enhancement, or new feature), such tag was automatically assigned to the commit and its correctness was double-checked by one of the authors, which verified the correctness of the issue category (*e.g.*, that an issue classified as a bug was actually a bug). We were able to automatically assign a tag with this process in 471 cases, *i.e.*, for a small percentage (5%) of the commits, which is not surprising and in agreement with previous findings [5]. Of these 471 automatically assigned tags, 126 were corrected during the manual double-check, most of them (96) due to a misclassification between enhancement and new feature. In the remaining 8,693 cases, two of the authors manually analyzed the commits, assigning one or more of the goal-tags by relying on the analysis of the commit messages and of the unix diffs between the commit under analysis and its predecessor.

Concerning the project-status tags (see Table 3), the *Working on release* tag can assume as possible values *one day*, *one week*, *one month*, or *more than one month* before the issuing of a major release. The aim of such a tag is to indicate whether, *when introducing the smell, the developer was close to a project's deadline*. We just consider major releases since those are the ones generally representing a real deadline for developers, while minor releases are sometimes issued just due to a single bug fix. To assign such tags, one of the authors identified the dates in which the major releases were issued by exploiting the GIT tags (often used to tag releases), and the commit messages left by developers. Concerning the *Project startup* tag, it can assume as values *one week*, *one month*, *one year*, or *more than one year* after the project's start date. This tag can be easily assigned by comparing the commit date with the date in which the project started (*i.e.*, the date of the first commit). This tag can be useful to verify whether *during the project's startup, when the project design might not be fully clear, developers are more prone to introduce smells*. Clearly, considering the date of the first commit in the repository as the project's startup date can introduce imprecisions in our data in case of projects migrated to *git* in a later stage of their history. For this reason, we verify whether the first release of each project in our dataset was tagged with 0.1 or 1.0 (*i.e.*, a version number likely indicating the first release of a project). As a result, we exclude from the *Project startup* analysis 31 projects having a partial change history in the mined *git* repository, for a total of 552 smell-introducing commits excluded. While we acknowledge that also this heuristic might introduce imprecisions (*e.g.*, a project starting from release 1.0 could still have a previous $0.x$ release), we are confident that it helps in eliminating most of the problematic projects from our dataset.

Finally, we assign developer-status tags to smell-introducing commits. The *Workload* tag measures how busy a developer was when introducing the bad smell. In particular, we measure the *Workload* of each developer involved in a project using time windows of one month, starting from the date in which the developer joined the project (*i.e.*, performed the first commit). The *Workload* of a developer during one month is measured in terms of the number of commits she performed in that month. We are aware that such a measure (i) is an approximation because different commits can require different amount of work; and (ii) a developer could also work on other projects. When analyzing a smell-introducing commit performed by a developer d during a month m, we compute the workload distribution for all developers of the project at m. Then, given Q_1 and Q_3, the first and the third quartile of such distribution, respectively, we assign: *low* as *Workload* tag if the developer performing the commit had a workload lower than Q_1, *medium* if $Q_1 \leq workload < Q_3$, *high* if the workload was higher than Q_3.

The *Ownership* tag is assigned if the developer performing the smell-introducing commit is the owner of the file on which the smell has been detected. As defined by Bird *et al.* [12], a file owner is a developer responsible for more than 75% of the commits performed on the file. Lastly, the *Newcomer* tag is assigned if the smell-introducing commit falls among the first three commits in the project for the developer responsible for it.

After assigning all the described tags to each of the 9,164 smell-introducing commits, we analyzed the results by reporting descriptive statistics of the number of commits to which each tag type has been assigned. Also, we discuss several qualitative examples helping to explain our findings.

2.2.3 What is the survivability of code smells?

To address RQ_3, we need to determine when a smell has been introduced and when a smell disappears from the system. To this aim, given a file f, we formally define two types of commits:

1) *last-smell-introducing commit*: A commit c_i modifying a file f such that, f is affected by a code smell $smell_k$ after commit c_i while it was not affected by $smell_k$ before c_i. Even if an artifact can become smelly as consequence of several modifications (see RQ_2), in this analysis we are interested in finding a specific date in which an artifact can actually be considered smelly. To this aim we consider the latest possible commit before f actually becomes smelly. Clearly, when a smell is introduced gradually, this commit is not the only responsible for the smell introduction, but, rather, it represents the "turning point" of the smell introduction process.
2) *smell-removing commit*: A commit c_i modifying a file f such that f is not affected by a code smell $smell_k$ after c_i while it was affected by $smell_k$ before c_i. Also, in this case, it may happen that the smell can be gradually removed, though we take the first commit in which the code smell detector does not spot the smell anymore.

Based on what has been discussed above, given a code smell $smell_k$, the time interval between the *last-smell-introducing commit* and the *smell-removing commit* is defined as *smelly interval*, and determines the longevity of $smell_k$. Given a smelly interval for a code smell affecting the file f and bounded by the *last-smell-introducing commit* c_i and the *smell-removing commit* c_j, we compute as proxies for the smell longevity:

- #*days*: the number of days between the introduction of the smell ($c_i.time$) and its fix ($c_j.time$);
- #*commits*: the number of commits between c_i and c_j that modified the artifact f.

These two proxies provide different and complementary views about the survivability of code smells. Indeed, considering only the #*days* (or any other time-based proxy) could lead to misleading interpretations in cases in which a project is mostly inactive (*i.e.,* no commits are performed) in a given time period. For example, suppose that a smell instance $smell_k$ is refactored 10 months after its introduction in the system. The #*days* proxy will indicate a very high survivability (∼300 days) for $smell_k$. However, we do not know whether the project was active in such a time period (and thus, if developers actually had the chance to fix $smell_k$). The #*commits* will provide us with such information: If the project was active, it will concur with the #*days* proxy in indicating a high survivability for $smell_k$, otherwise it will "contradict", showing a low survivability in terms of #*commits*.

Since we are analyzing a finite change history for a given repository, it could happen that for a specific file and a smell affecting it we are able to detect the *last-smell-introducing commit* but not the *smell-removing commit*, due to the fact that the file is still affected by the code smell in the last commit we analyzed. In other words, we can discriminate two different types of smelly intervals in our dataset:

1) *Closed Smelly Intervals*: intervals delimited by a *last-smell-introducing commit* as well as by a *smell-removing commit*;
2) *Censored Smelly Intervals*: intervals delimited by a *last-smell-introducing commit* and by the end of the change history (*i.e.,* the date of the last commit we analyzed).

In total, we identified 1,426 closed smelly intervals and 9,197 censored smelly intervals. After having collected this data, we answer RQ_3 by relying on *survival analysis* [67], a statistical method that aims at analyzing and modeling the time duration until one or more events happen. Such time duration is modeled as a random variable and typically it has been used to represent the time to the failure of a physical component (mechanical or electrical) or the time to the death of a biological unit (patient, animal, cell, *etc.*) [67]. The survival function $S(t) = Pr(T > t)$ indicates the probability that a subject (in our case the code smell) survives longer than some specified time t. The survival function never increases as t increases; also, it is assumed $S(0) = 1$ at the beginning of the observation period, and, for time $t \to \infty$, $S(\infty) \to 0$. The goal of the survival analysis is to estimate such a survival function from data and assess the relationship of explanatory variables (covariates) to survival time. Time duration data can be of two types:

1) *Complete data*: the value of each sample unit is observed or known. For example, the time to the failure of a mechanical component has been observed and reported. In our case, the code smell disappearance has been observed.
2) *Censored Data*: The event of interest in the analysis has not been observed yet (so it is considered as unknown). For example, a patient cured with a particular treatment has been alive till the end of the observation window. In our case, the smell remains in the system until the end of the observed project

history. For this sample, the time-to-death observed is a censored value, because the event (death) has not occurred during the observation.

Both complete and censored data can be used, if properly marked, to generate a survival model. The model can be visualized as a survival curve that shows the survival probability as a function of the time. In the context of our analysis, the population is represented by the code smell instances while the event of interest is its fix. Therefore, the "time-to-death" is represented by the observed time from the introduction of the code smell instance, till its fix (if observed in the available change history). We refer to such a time period as "the lifetime" of a code smell instance. Complete data is represented by those instances for which the event (fix) has been observed, while censored data refers to those instances which have not been fixed in the observable window. We generate survival models using both the #days and #commits in the smelly intervals as time variables. We analyzed the survivability of code smells by ecosystem. That is, for each ecosystem, we generated a survival model for each type of code smell by using R and the `survival` package[5]. In particular, we used the `Surv` type to generate a survival object and the `survfit` function to compute an estimate of a survival curve, which uses Kaplan-Meier estimator [34] for censored data. In the latter, we use the `conf.type=''none''` argument to specify that we do not want to include any confidence interval for the survival function. Also, we decided to use the Kaplan-Meier estimator, a non-parametric survival analysis method, since we cannot assume a particular distribution of survival times. Such an estimator has been widely used in the literature, for example to study the longevity of Debian packages [69] or to analyze when source code becomes dead code (unused code) [20].

We report the survival function for each type of code smell grouped by ecosystem. In addition, we compare the survival curve of artifacts that are born smelly (*i.e.,* those in which the code smell appears in the commit creating the artifact) with the survival curve of artifacts that became smelly during maintenance and evolution activities.

It is important to highlight that, while the survival analysis is designed to deal with censored data, we perform a cleaning of our dataset aimed at reducing possible biases caused by censored intervals before running the analysis. In particular, code smell instances introduced too close to the end of the observed change history can potentially influence our results, since in these cases the period of time needed for their removal is too small for being analyzed. Thus, we excluded from our survival analysis all censored intervals for which the *last-smell-introducing commit* was "too close" to the last commit we analyzed in the system's change history (*i.e.,* for which the developers did not have "enough time" to fix them). To determine a threshold suitable to remove only the subset of smell instances actually too close to the end of the analyzed change history, we study the distribution of the number of days needed to fix the code smell instance (*i.e.,* the length of the closed smelly interval) in our dataset and, then, we choose an appropriate threshold (see Section 3.3).

5. https://cran.r-project.org/package=survival

2.2.4 How do developers remove code smells?

In order to understand how code smells disappear from the system, we manually analyzed a randomly selected set of 979 *smell-removing commits*. Such a set represents a 95% statistically significant stratified sample with a 5% confidence interval of the 1,426 smell-removing commits in our dataset. The *strata* of such a sample are represented by (i) the three ecosystems analyzed (*i.e.,* we make sure to consider a statistically significant sample for each of the three subject ecosystems), and (ii) the five different code smells considered in our study, *i.e.,* the higher the number of fixing commits involving a smell type (*e.g.,* Blob), the higher the number of *smell-removing commits* involving such a smell type in our manually evaluated sample. In other words, we determined a sample size (for the desired confidence interval and significance level) for each combination of smell type and ecosystem, sampled and manually analyzed accordingly.

To analyze and categorize the type of action performed by the developers that caused the smell to disappear (*e.g.,* refactoring, code deletion, *etc.*), we followed an open coding process. In particular, we randomly distributed the 979 commits among three of the authors (∼326 commits each). Each of the involved authors independently analyzed the commits assigned to him by relying on the commit note and the unix diff as shown by GitHub (all subject systems are hosted on GitHub). The output of this phase was the assignment of each *smell-removing commit* to a given category explaining why the smell disappeared from the system (*e.g.,* the smell has been refactored, the code affected by the smell has been deleted, *etc.*). Then, the three authors involved in the classification discussed their codings in order to (i) double-check the consistency of their individual categorization, and (ii) refine the identified categories by merging similar categories they identified or splitting when it was the case.

The output of our open coding procedure is the assignment of the 979 commits to a category explaining the reason *why* a specific smell disappeared in a given commit. We quantitatively and qualitatively discuss such data in our results section.

3 Analysis of the Results

This section reports the analysis of the results achieved in our study and aims at answering the four research questions formulated in Section 2.

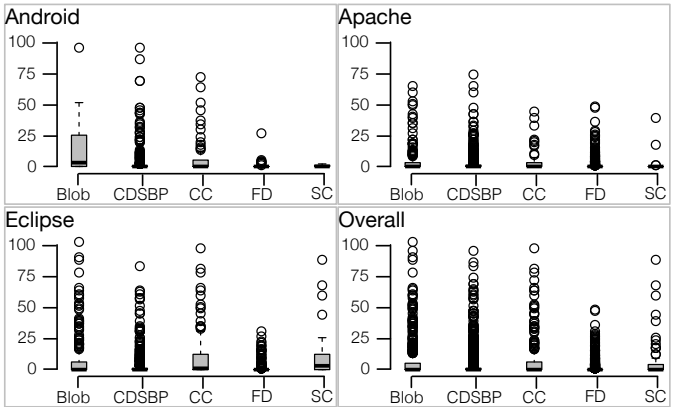


Fig. 2: The number of commits required by a smell to manifest itself.

3.1 When are code smells introduced?

Fig. 2 shows the distribution of the number of commits required by each type of smell to manifest itself. The results are grouped by ecosystems; also, we report the *Overall* results (all ecosystems together).

As we can observe in Fig. 2, in almost all the cases the median number of commits needed by a smell to affect code components is zero, except for Blob on Android (median=3) and Complex Class on Eclipse (median=1). In other words, most of the smell instances (at least half of them) are introduced when a code entity is added to the versioning system. This is quite surprising finding, considering the common wisdom that *smells are generally the result of continuous maintenance activities performed on a code component* [28].

However, the box plots also indicate (i) the presence of several outliers; and that (ii) for some smells, in particular Blob and Complex Class, the distribution is quite skewed. This means that besides smell instances introduced in the first commit, there are also several smell instances that are introduced as a result of several changes performed on the file during its evolution. In order to better understand such phenomenon, we analyzed how the values of some quality metrics change during the evolution of such files.

Table 4 presents the descriptive statistics (mean and median) of the slope of the regression line computed, for each metric, for both smelly and clean files. Also, Table 4 reports the results of the Mann-Whitney test and Cliff's d effect size (**L**arge, **M**edium, or **S**mall) obtained when analyzing the difference between the slope of regression lines for clean and smelly files. Column *cmp* of Table 4 shows a $\uparrow$ ($\downarrow$) if for the metric m there is a statistically significant difference in the m's slope between the two groups of files (*i.e., clean* and *smelly*), with the smelly ones exhibiting a higher (lower) slope; a "$-$" is shown when the difference is not statistically significant.

The analysis of the results reveals that for all the smells, but Functional Decomposition, the files affected by smells show a higher slope than clean files. This suggests that the files that will be affected by a smell exhibit a steeper growth in terms of metric values than files that are not becoming smelly. In other words, when a smell is going to appear, its operational indicators (metric value increases) occur very fast (not gradually). For example, considering the Apache ecosystem, we can see a clear difference between the growth of LOC in Blob and clean classes. Indeed, this latter have a mean growth in terms of LOC characterized by a slope of 0.40, while the slope for Blobs is, on average, 91.82. To make clear the interpretation of such data, let us suppose we plot both regression lines on the Cartesian plane. The regression line for Blobs will have an inclination of 89.38°, indicating an abrupt growth of LOC, while the inclination of the regression line for clean classes will be 21.8°, indicating less steep increase of LOC. The same happens when considering the LCOM cohesion metric (the higher the LCOM, the lower the class cohesion). For the overall dataset, the slope for classes that will become Blobs is 849.90 as compared to the 0.25 of clean classes. Thus, while the cohesion of classes generally decreases over time, classes destined to become Blobs exhibit cohesion metric loss orders of magnitude faster than clean classes. In general, the results in Table 4 show strong differences in the metrics' slope between clean and smelly files, indicating that it could be possible to create recommenders warning developers when the changes performed on a specific code component show a dangerous trend potentially leading to the introduction of a bad smell.

The Functional Decomposition (FD) smell deserves a separate discussion. As we can see in Table 4, the slope of the regression line for files affected by such a smell is negative. This means that during the evolution of files affected by Functional Decomposition we can observe a decrement (rather than an increment) of the metric values. The rationale behind such a result is intrinsic in the definition of this smell. Specifically, one of the symptoms of such a smell is represented by a class with a single action, such as a function. Thus, the changes that could introduce a Functional Decomposition might be the removal of responsibilities (*i.e.,* methods). This clearly results in the decrease of some metrics, such as NOM, LOC and WMC. As an example, let us consider the class `DisplayKMeans` of Apache Mahout. The class implements the K-means clustering algorithm in its original form. However, after three commits the only operation performed by the class was the visualization of the clusters. Indeed, developers moved the actual implementation of the clustering algorithm in the class `Job` of the package `kmeans`, introducing a Functional Decomposition in `DisplayKMeans`.

Overall, by analyzing Table 4 we can conclude that (i) LOC characterizes the introduction of all the smells; (ii) LCOM, WMC, RFC and NOM characterize all the smells but Class Data Should be Private; (iii) CBO does not characterize the introduction of any smell; and (iv) the only metrics characterizing the introduction of Class

TABLE 4: $\mathbf{RQ}_1$: slope affected *vs* slope not affected - Mann-Whitney test (adj. p-value) and Cliff's Delta (*d*).

Ecosys.	Smell	Affected	LOC mean	LOC med	cmp	LCOM mean	LCOM med	cmp	WMC mean	WMC med	cmp	RFC mean	RFC med	cmp	CBO mean	CBO med	cmp	NOM mean	NOM med	cmp	NOA mean	NOA med	cmp
Android	Blob	NO	0.68	0		0.55	0		0.17	0		0.13	0		0.15	0		0.07	0		0.09	0	
		YES	32.90	12.51	↑	13.80	2.61	↑	3.78	1.81	↑	5.39	3.47	↑	1.34	0.69	↑	1.15	0.57	↑	0.49	0.13	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's *d*	0.65 (L)			0.38 (M)			0.53 (L)			0.64 (L)			0.66 (L)			0.51 (L)			0.56 (L)		
	CDSP	NO	0.42	0		0.12	0		0.12	0		0.05	0		0.09	0		0.05	0		0.06	0	
		YES	4.43	1.68	↑	0.83	0	—	0.33	0	—	0.27	0	—	0.36	0.18	↑	0.17	0	—	2.60	0.69	↑
		p-value	<0.01			0.26			0.88			0.86			<0.01			0.71			<0.01		
		Cliff's *d*	0.45 (M)			0.06 (N)			-0.01 (N)			-0.01 (N)			0.26 (S)			0.02 (N)			0.78 (L)		
	CC	NO	0.67	0		0.48	0		0.19	0		0.14	0		0.15	0		0.08	0		0.09	0	
		YES	7.71	6.81	↑	11.16	4.12	↑	2.61	2.20	↑	2.42	1.01	↑	0.33	0.28	↑	0.67	0.50	↑	0.18	0.10	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's *d*	0.63 (L)			0.45 (M)			0.76 (L)			0.64 (L)			0.32 (S)			0.67 (L)			0.39 (M)		
	FD	NO	0.99	0		0.62	0		0.29	0		0.31	0		0.40	0		0.11	0		0.11	0	
		YES	-10.56	-1.00	↓	-2.65	0	↓	-2.74	-0.60	↓	-3.49	0	↓	0.78	0.49	—	-1.13	-0.30	↓	-0.91	0	↓
		p-value	<0.01			<0.01			<0.01			0.02			0.09			<0.01			0.01		
		Cliff's *d*	-0.55 (L)			-0.49 (L)			-0.59 (L)			-0.42 (M)			0.32 (S)			-0.76 (L)			-0.45 (M)		
	SC	NO	1.42	0		0.96	0		0.31	0		0.42	0		0.29	0		0.11	0		0.13	0	
		YES	144.2	31.0	↑	69.17	100.00	↑	10.17	10.00	↑	6.33	5.00	↑	0.67	1.00	—	3	3	↑	0.16	0	↑
		p-value	<0.01			<0.01			<0.01			<0.01			0.50			<0.01			0.04		
		Cliff's *d*	0.99 (L)			0.98 (L)			0.99 (L)			0.95 (L)			0.22 (S)			0.99 (L)			0.68 (L)		
Apache	Blob	NO	0.40	0		0.42	0		0.13	0		0.13	0		0.05	0		0.05	0		0.03	0	
		YES	91.82	33.58	↑	384.70	12.40	↑	17.79	4.92	↑	27.61	7.09	↑	2.17	0.50	↑	7.64	1.72	↑	0.77	0.05	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's *d*	0.92 (L)			0.52 (L)			0.77 (L)			0.74 (L)			0.44 (M)			0.82 (L)			0.22 (S)		
	CDSP	NO	0.43	0		0.54	0		0.12	0		0.12	0		0.10	0		0.05	0		0.03	0	
		YES	8.69	2.03	↑	2.44	0	—	0.61	0	—	0.59	0	—	0.55	0.06	↑	0.23	0	—	3.28	1.07	↑
		p-value	<0.01			0.28			0.46			0.45			<0.01			0.37			<0.01		
		Cliff's *d*	0.63 (L)			-0.04 (N)			-0.03 (N)			0.03 (N)			0.25 (S)			-0.03 (N)			0.86 (L)		
	CC	NO	0.36	0		0.47	0		0.12	0		0.13	0		0.09	0		0.05	0		0.04	0	
		YES	121.80	25.86	↑	886.50	152.40	↑	31.87	10.36	↑	39.81	7.21	↑	3.45	0.53	↑	13.99	3.56	↑	0.17	0	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			0.02		
		Cliff's *d*	0.81 (L)			0.70 (L)			0.83 (L)			0.74 (L)			0.53 (L)			0.82 (L)			0.23 (S)		
	FD	NO	0.52	0		0.812	0		0.16	0		0.14	0		0.10	0		0.07	0		0.030	0	
		YES	-13.78	-3.32	↓	-5.98	-0.30	↓	-6.16	-1.00	↓	-4.81	-0.52	↓	-0.28	0	↓	-2.82	-0.53	↓	-0.40	0	↓
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's *d*	-0.72 (L)			-0.46 (M)			-0.66 (L)			-0.49 (L)			-0.14 (N)			-0.67 (L)			-0.35 (M)		
	SC	NO	0.54	0		0.11	0		0.11	0		0.12	0		0.14	0		0.04	0		0.04	0	
		YES	273.00	129.90	↑	232.30	4.50	—	7.09	6.50	↑	10.81	10.15	↑	0.96	0.92	—	3.41	3.00	↑	2.29	2.08	↑
		p-value	<0.01			0.52			<0.01			<0.01			0.12			<0.01			0.02		
		Cliff's *d*	0.94 (L)			0.17 (S)			0.91 (L)			0.95 (L)			0.44 (M)			0.94 (L)			0.63 (L)		
Eclipse	Blob	NO	0.02	0		0.02	0		-0.01	0		-0.03	0		0.13	0		-0.01	0		0.01	0	
		YES	69.51	28.15	↑	1208.00	14.71	↑	17.10	2.92	↑	18.15	2.44	↑	0.58	0.01	↑	7.11	1.09	↑	3.11	0.09	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's *d*	0.86 (L)			0.54 (L)			0.76 (L)			0.65 (L)			0.20 (S)			0.75 (L)			0.50 (L)		
	CDSP	NO	0.01	0		0.34	0		<-0.01	0		-0.02	0		0.13	0		<-0.01	0		0.01	0	
		YES	12.58	2.50	↑	749.1	0	↑	2.77	0	↑	0.70	0	↑	0.37	0	—	2.10	0	↑	4.01	1	↑
		p-value	<0.01			<0.01			<0.01			<0.01			0.53			<0.01			<0.01		
		Cliff's *d*	0.65 (L)			0.13 (N)			0.16 (S)			0.12 (N)			0.03 (N)			0.18 (S)			0.90 (L)		
	CC	NO	0.02	0		0.21	0		-0.01	0		-0.05	0		0.11	0		-0.01	0		0.02	0	
		YES	57.72	18.00	↑	2349.00	141.70	↑	19.86	4.86	↑	10.46	0.82	↑	0.68	0.01	↑	10.23	1.94	↑	3.10	<0.01	↑
		p-value	<0.01			<0.01			<0.01			<0.01			0.02			<0.01			<0.01		
		Cliff's *d*	0.82 (L)			0.75 (L)			0.84 (L)			0.54 (L)			0.15 (S)			0.83 (L)			0.42 (M)		
	FD	NO	-0.02	0		0.67	0		-0.02	0		-0.02	0		0.13	0		-0.01	0		0.02	0	
		YES	-15.09	-5.40	↓	-5.23	-0.95	↓	-5.15	-1.71	↓	-4.06	-0.60	↓	-0.16	0.16	↑	-2.39	-0.60	↓	-0.35	0	—
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			0.88		
		Cliff's *d*	-0.72 (L)			-0.61 (L)			-0.71 (L)			-0.51 (L)			0.23 (S)			-0.74 (L)			-0.01 (N)		
	SC	NO	0.07	0		1.19	0		0.02	0		-0.06	0		0.15	0		-0.01	0		0.02	0	
		YES	114.40	42.74	↑	698.4	137.3	↑	16.65	4.03	↑	9.47	0.03	↑	1.37	0	—	6.44	2.39	↑	9.30	1.17	↑
		p-value	<0.01			<0.01			<0.01			<0.01			0.97			<0.01			<0.01		
		Cliff's *d*	0.92 (L)			0.52 (L)			0.61 (L)			0.44 (M)			0.01 (N)			0.51 (L)			0.65 (L)		
Overall	Blob	NO	0.25	0		0.25	0		0.07	0		0.06	0		0.09	0		0.02	0		0.02	0	
		YES	73.76	29.14	↑	849.90	9.57	↑	16.26	3.30	↑	20.17	3.04	↑	1.15	0.20	↑	6.81	1.12	↑	2.15	0.08	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's *d*	0.87 (L)			0.52 (L)			0.74 (L)			0.67 (L)			0.32 (S)			0.75 (L)			0.42 (M)		
	CDSP	NO	0.26	0		0.43	0		0.07	0		0.06	0		0.11	0		0.03	0		0.02	0	
		YES	9.36	2.10	↑	290.50	0	—	1.39	0	↑	0.57	0	↑	0.44	0	↑	0.94	0	↑	3.42	1.00	↑
		p-value	<0.01			0.3			0.04			0.02			<0.01			0.01			<0.01		
		Cliff's *d*	0.61 (L)			0.05 (N)			0.05 (N)			0.05 (N)			0.17 (S)			0.06 (N)			0.87 (L)		
	CC	NO	0.21	0		0.34	0		0.06	0		0.04	0		0.10	0		0.02	0		0.03	0	
		YES	63.00	12.60	↑	1573.00	46.81	↑	19.36	3.81	↑	15.68	1.93	↑	1.25	0.18	↑	9.29	1.40	↑	1.88	0.01	↑
		p-value	<0.01			<0.01			<0.01			<0.01			<0.01			<0.01			<0.01		
		Cliff's *d*	0.79 (L)			0.69 (L)			0.82 (L)			0.61 (L)			0.30 (S)			0.81 (L)			0.39 (M)		
	FD	NO	0.29	0		0.75	0		0.08	0		0.07	0		0.12	0		0.03	0		0.02	0	
		YES	-14.09	-4.00	↓	-5.59	-0.50	↓	-5.67	-1.37	↓	-4.50	-0.54	↓	-0.19	0	—	-2.60	-0.57	↓	-0.40	0	↓
		p-value	<0.01			<0.01			<0.01			<0.01			0.75			<0.01			<0.01		
		Cliff's *d*	-0.71 (L)			-0.51 (L)			-0.67 (L)			-0.49 (L)			0.01 (N)			-0.69 (L)			-0.22 (S)		
	SC	NO	0.17	0		1.02	0		0.04	0		-0.02	0		0.15	0		0.01	0		0.03	0	
		YES	134.00	36.29	↑	597.0	100.0	↑	15.09	6.34	↑	9.36	1.00	↑	1.27	0	—	5.84	3.00	↑	7.80	0.57	↑
		p-value	<0.01			<0.01			<0.01			<0.01			0.49			<0.01			<0.01		
		Cliff's *d*	0.93 (L)			0.52 (L)			0.66 (L)			0.52 (L)			0.06 (N)			0.59 (L)			0.67 (L)		

Data Should be Private are LOC and NOA.

Summary for $\mathbf{RQ}_1$. Most of the smell instances are introduced when the files are created. However, there are also cases, especially for Blob and Complex Class, where the smells manifest themselves after several changes performed on the file. In these cases, the files that will become smelly exhibit specific trends for some quality metric values that are significantly different than those of clean files.

3.2 Why are code smells introduced?

To answer $\mathbf{RQ}_2$, we analyzed the percentage of smell-introducing commits classified according to the category of tags, *i.e.*, *commit goal*, *project status*, and *developer status*.

Commit-Goal: Table 5 reports the percentage of smell-introducing commits assigned to each tag of the category *commit-goal*. Among the three different ecosystems analyzed, results show that smell instances are mainly introduced when developers perform enhancement operations on the system. When analyzing the three ecosystems altogether, for all the considered types of smells the percentage of smell-introducing commits tagged as *enhancement* ranges between 60% and 66%. Note that by *enhancement* we mean changes applied by developers on existing features aimed at improving them. For example, a Functional Decomposition was introduced in the class `CreateProjectFromArchetypeMojo` of Apache Maven when the developer performed the "*first pass at implementing the feature of being able to specify additional goals that can be run after the creation of a project from an archetype*" (as reported in the commit log).

Note that when considering *enhancement* or *new feature* all together, the percentage of smell-introducing commits exceeds, on average, 80%. This indicates, as expected,

TABLE 5: **RQ$_2$**: Commit-goal tags to smell-introducing commits. BF: Bug Fixing; E: Enhancement; NF: New Feature; R: Refactoring.

Smell	Android				Apache				Eclipse				Overall			
	BF	E	NF	R	BF	E	NF	R	BF	E	NF	R	BF	E	NF	R
Blob	15	59	23	3	5	83	10	2	19	55	19	7	14	65	17	4
CDSP	11	52	30	7	6	63	30	1	14	64	18	4	10	60	26	4
CC	0	44	56	0	3	89	8	0	17	52	24	7	13	66	16	5
FD	8	48	39	5	16	67	14	3	18	52	24	6	16	60	20	4
SC	0	0	100	0	0	81	4	15	8	61	22	9	6	66	17	11

that the most smell-prone activities are performed by developers when adding new features or improving existing features. However, there is also a non-negligible number of smell-introducing commits tagged as *bug fixing* (between 6% and 16%). This means that also during corrective maintenance developers might introduce a smell, especially when the bug fixing is complex and requires changes to several code entities. For example, the class `SecuredModel` of Apache Jena builds the security model when a semantic Web operation is requested by the user. In order to fix a bug that did not allow the user to perform a safe authentication, the developer had to update the model, implementing more security controls. This required changing several methods present in the class (10 out of 34). Such changes increase the whole complexity of the class (the WMC metric increased from 29 to 73) making `SecuredModel` a Complex Class.

Another interesting observation from the results reported in Table 5 is related to the number of smell-introducing commits tagged as *refactoring* (between 4% and 11%). While refactoring is the principal treatment to remove smells, we found 394 cases in which developers introduced new smells when performing refactoring operations. For example, the class `EC2ImageExtension` of Apache jClouds implements the `ImageExtension` interface, which provides the methods for creating an image. During the evolution, developers added methods for building a new image template as well as a method for managing image layout options (*e.g.*, its alignment) in the `EC2ImageExtension` class. Subsequently, a developer performed an Extract Class refactoring operation aimed at reorganizing the responsibility of the class. Indeed, the developer split the original class into two new classes, *i.e.*, `ImageTemplateImpl` and `CreateImageOptions`. However, the developer also introduced a Functional Decomposition in the class `CreateImageOptions` since such a class, after the refactoring, contains just one method, *i.e.,* the one in charge of managing the image options. This result shows that refactoring can sometimes lead to unexpected side effects; besides the risk of introducing faults [9], when performing refactoring operations, there is also the risk of introducing design problems.

Looking into the ecosystems, the general trend discussed so far holds for Apache and Eclipse. Regarding Android, we notice something different for Complex Class and Spaghetti Code smells. In these cases, the smell-introducing commits are mainly due to the introduction of new features. Such a difference could be due to the particular development model used for Android apps. Specifically, we manually analyzed the instances of smells identified in 70 Android apps, and we observed that in the majority of the cases classes affected by a smell are those extending the Android `Activity` class, *i.e.*, a class extended by developers to provide features to the app's users. Specifically, we observed that quite often developers introduce a Complex Class or a Spaghetti Code smell when adding a new feature to their apps by extending the `Activity` class. For example, the class `ArticleViewActivity` of the Aard[6] app became a Complex Class after adding several new features (spread across 50 commits after its creation), such as the management of page buttons and online visualization of the article. All these changes contributed to increase the slope of the regression line for the RFC metric of a factor of 3.91 and for WMC of a factor of 2.78.

TABLE 6: **RQ$_2$**: Project-Status tags to smell-introducing commits.

Ecosystem	Smell	Working on Release				Project Startup			
		One Day	One Week	One Month	More	One Week	One Month	One Year	More
Android	Blob	7	54	35	4	6	3	35	56
	CDSP	14	20	62	4	7	17	33	43
	CC	0	6	94	0	0	12	65	23
	FD	1	29	59	11	0	4	71	25
	SC	0	0	100	0	0	0	0	100
Apache	Blob	19	37	43	1	3	7	54	36
	CDSP	10	41	46	3	3	8	45	44
	CC	12	30	57	1	2	14	46	38
	FD	5	14	74	7	3	8	43	46
	SC	21	18	58	3	3	7	15	75
Eclipse	Blob	19	37	43	1	3	20	32	45
	CDSP	10	41	46	3	6	12	39	43
	CC	12	30	57	1	2	12	42	44
	FD	5	14	73	8	2	5	35	58
	SC	21	18	58	3	1	5	19	75
Overall	Blob	15	33	50	2	5	14	38	43
	CDSP	10	29	58	3	6	12	39	43
	CC	18	28	53	1	4	13	42	41
	FD	7	22	66	5	3	7	42	48
	SC	16	20	58	6	2	6	17	75

Project status: Table 6 reports the percentage of smell-introducing commits assigned to each tag of the *project-status* category. As expected, most of the smells are introduced the last month before issuing a release. Indeed, the percentage of smells introduced more than one month prior to issuing a release is really low (ranging between 0% and 11%). This consideration holds for all the ecosystems and for all the bad smells analyzed, thus suggesting that the deadline pressure — assuming that release dates are planned — could be one of the main causes for smell introduction. Clearly, such a pressure might also be related to an expected more intense de-

6. Aard is an offline Wikipedia reader.

TABLE 7: $\mathbf{RQ_2}$: Developer-Status tags to smell-introducing commits.

Ecosystem	Smell	Workload			Ownership		Newcomer	
		High	Medium	Low	True	False	True	False
Android	Blob	44	55	1	73	27	4	96
	CDSP	79	10	11	81	19	11	89
	CC	53	47	0	100	0	6	94
	FD	68	29	3	100	0	8	92
	SC	100	0	0	100	0	100	0
Apache	Blob	67	31	2	64	36	7	93
	CDSP	68	26	6	53	47	14	86
	CC	80	20	0	40	60	6	94
	FD	61	36	3	71	29	7	93
	SC	79	21	0	100	0	40	60
Eclipse	Blob	62	32	6	65	35	1	99
	CDSP	62	35	3	44	56	9	91
	CC	66	30	4	47	53	9	91
	FD	65	30	5	58	42	11	89
	SC	43	32	25	79	21	3	97
Overall	Blob	60	36	4	67	33	3	97
	CDSP	68	25	7	56	44	11	89
	CC	69	28	3	45	55	3	97
	FD	63	33	4	67	33	8	92
	SC	55	28	17	79	21	15	85

velopment activity (and a higher workload) developers are forced to bear to meet the deadline. Indeed, while we found no correlation in general between the distribution of commits and the distribution of code smell introduction (Spearman correlation value = -0.19), we observed a higher frequency of commits during the last month before a deadline, which tends to increase in the last week with a peak in the last day. This increasing rate of commits close to the deadline is also moderately correlated to a slightly increasing rate of code smell introduction during last month of activity and close to the deadline (Spearman correlation value = 0.516).

Considering the *project startup* tag, the results are quite unexpected. Indeed, a high number of smell instances are introduced few months after the project startup. This is particularly true for Blob, Class Data Should Be Private, and Complex Class, where more than half of the instances are introduced in the first year of systems' observed life history. Instead, Functional Decomposition, and especially Spaghetti Code, seem to be the types of smells that take more time to manifest themselves with more than 75% of Spaghetti Code instances introduced after the first year. This result contradicts, at least in part, the common wisdom that smells are introduced after several continuous maintenance activities and, thus, are more pertinent to advanced phases of the development process [28], [62].

Developer status: Finally, Table 7 reports the percentage of smell-introducing commits assigned to each tag of the *developer-status* category. From the analysis of the results it is evident that the developers' workload negatively influences the quality of the source code produced. On the overall dataset, at least in 55% of cases the developer who introduced the smell had a high workload. For example, on the `InvokerMavenExecutor` class in Apache Maven a developer introduced a Blob smell while adding the command line parsing to enable users to alternate the settings. When performing such a change, the developer had relatively high workload while working on nine other different classes (in this case, the workload was classified as high).

TABLE 8: Descriptive statistics of the number of days needed a smell remained in the system before being removed.

Ecosystem	Min	1st Qu.	Median	Mean	3rd Qu.	Max.
Android	0	5	40	140.8	196	1261
Apache	0	10	101	331.7	354	3244
Eclipse	0	21	135	435.2	446	5115

Developers who introduce smells are not newcomers, while often they are owners of smell-related files. This could look like an unexpected result, as the owner of the file—one of the most experienced developers of the file—is the one that has the higher likelihood of introducing a smell. However, it is clear that somebody who performs many commits has a higher chance of introducing smells. Also, as discussed by Zeller in his book *Why programs fail*, more experienced developers tend to perform more complex and critical tasks [92]. Thus, it is likely that their commits are more prone to introducing design problems.

Summary for $\mathbf{RQ_2}$. Smells are generally introduced by developers when enhancing existing features or implementing new ones. As expected, smells are generally introduced in the last month before issuing a deadline, while there is a considerable number of instances introduced in the first year from the project startup. Finally, developers who introduce smells are generally the owners of the file and they are more prone to introducing smells when they have higher workloads.

3.3 What is the survivability of code smells?

We start by analyzing the data for smells that have been removed from the system, *i.e.,* those for which there is a closed interval delimited by a *last-smell-introducing commit* and *smell-removing-commit*. Figure 3 shows the box plot of the distribution of the number of days needed to fix a code smell instance for the different ecosystems. The box plots, depicted in log-scale, show that while few code smell instances are fixed after a long period of time (*i.e.,* even over 500 days) most of the instances are fixed in a relatively short time.

Table 8 shows the descriptive statistics of the distribution of the number of days when aggregating all code smell types considered in our study. We can notice considerable differences in the statistics for the three analyzed ecosystems. In particular, the median value of such distributions are 40, 101 and 135 days for Android, Apache and Eclipse projects, respectively. While it is difficult to speculate on the reasons why code smells are fixed quicker in the Android ecosystem than in the Apache and Eclipse ones, it is worth noting that on one hand Android apps generally have a much smaller size with respect to systems in the Apache and Eclipse ecosystems (*i.e.,* the average size, in terms of KLOC, is 415 for Android, while it is 1,417 for Apache and 1,534 for Eclipse), and on the other hand they have a shorter release cycles if compared with the other

considered ecosystems. Because of these differences we decided to perform separate survivability analysis for the three ecosystems. As a consequence, we also selected a different threshold for each ecosystem when excluding code smell instances introduced too close to the end of the observed change history, needed to avoid cases in which the period of time needed for removing the smell is too short for being analyzed (see Section 2.2.3). Analyzing the distribution, we decided to choose the median as threshold, since it is a central value not affected by outliers, as opposed to the mean. Also, the median values of the distributions are small enough to consider discarded smells in the censored interval close to the end of the observed change history (if compared for example to the mean time to remove a smell). Therefore, we used as threshold values 40, 101 and 135 days respectively for Android, Apache and Eclipse projects. Note that the censored intervals that we did not exclude were opportunely managed by the survival model.

Figure 4 shows the number of modifications (*i.e.*, commits modifying the smelly file) performed by the developer between the introduction and the removal of the code smell instance. These results clearly show that most of the code smell instances are removed after a few commits, generally no more than five commits for Android and Apache, and ten for Eclipse. By combining what has been observed in terms of the number of days and the number of commits a smell remains in the system before being removed, we can conclude that if code smells are removed, this usually happens after few commits from their introduction, and in a relatively short time.

Figures 5 and 6 show the survivability curves for each type of code smell and for each ecosystem in terms of number of days and number of commits, respectively. Remember that, while the previous analysis was just limited to closed intervals (*i.e.*, smells that have been removed), here we also consider censored intervals (*i.e.*, smells that have been introduced but not removed until the last day of the analyzed change history). Overall, the plots show that the survivability of code smells is quite high. In particular, after 1,000 days, the survival probability of a code smell instance (*i.e.*, the probability that the code smell has not been removed yet) is around 50% for Android and 80% for Apache and Eclipse. Looking at the number of commits, after 2,000 commits the survival probability is still 30% for Android, 50% for Apache, and 75% for Eclipse.

These results may appear in contrast with respect to what has been previously observed while analyzing closed intervals. However, this is due to the very high percentage of unfixed code smells present in the subject systems and ignored in the closed intervals analysis. Table 9 provides an overview of the percentage of fixed and unfixed code smell instances found in the observable change history[7]. As we can see, the vast majority of code smells (81.4%, on average) are not removed, and this result is consistent across the three ecosystem (83% in Android, 87% in Apache, and 74% in Eclipse). The most refactored smell is the Blob with, on average, 27% of refactored instances. This might be due to the fact that such a smell is more visible than others due to the large size of the classes affected by it.

TABLE 9: Percentage of code smells removed and not in the observed change history.

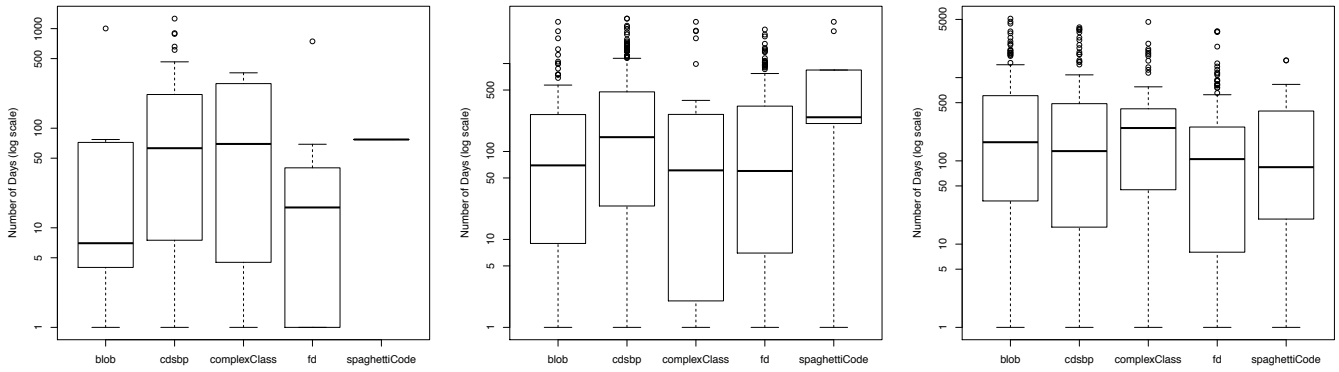
Smell	Android		Apache		Eclipse	
	Removed	Not Removed	Removed	Not Removed	Removed	Not Removed
Blob	36	64	15	85	31	69
CDSBP	14	86	12	88	17	83
CC	15	85	14	86	30	70
FD	9	91	9	91	10	90
SC	11	89	13	87	43	57

Further insights about the survivability of the smells across the three ecosystems are provided in the survival models (*i.e.*, Figures 5 and 6). The survival of Complex Class (blue line) and Spaghetti Code (brown line) is much higher in systems belonging to the Apache ecosystem with respect to systems belonging to the Android and Eclipse ecosystems. Indeed, these two smell types are the ones exhibiting the highest survivability in Apache and the lowest survivability in Android and Eclipse. Similarly, we can notice that the survival curves for CDSBP (green) and FD (yellow) exhibit quite different shapes between Eclipse (higher survivability) and the other two ecosystems (lower survivability). Despite these differences, the outcome that can be drawn from the observation of the survival models is one and valid across all the ecosystems and for all smell types: *the survivability of code smells is very high, with over* 50% *of smell instances still "alive" after 1,000 days and 1,000 commits from their introduction*.

Finally, we analyzed differences in the survivability of code smell instances affecting "born-smelly-artifacts" (*i.e.*, code files containing the smell instance since their creation) and "not-born-smelly-artifacts" (*i.e.*, code files in which the code smell has been introduced as a consequence of maintenance and evolution activities). Here there could be two possible scenarios: on the one hand developers might be less prone to refactor and fix born-smelly-artifacts than not-born-smelly-artifacts, since the code smell is somehow part of the original design of the code component. On the other hand, it could also be the case that the initial design is smelly because it is simpler to realize and release, while code smell removal is planned as a future activity. Both these conjectures have not been confirmed by the performed data analysis. As an example, we report the results achieved for the CDSBP and the Complex Class smell (the complete results are available in our online appendix [81]).

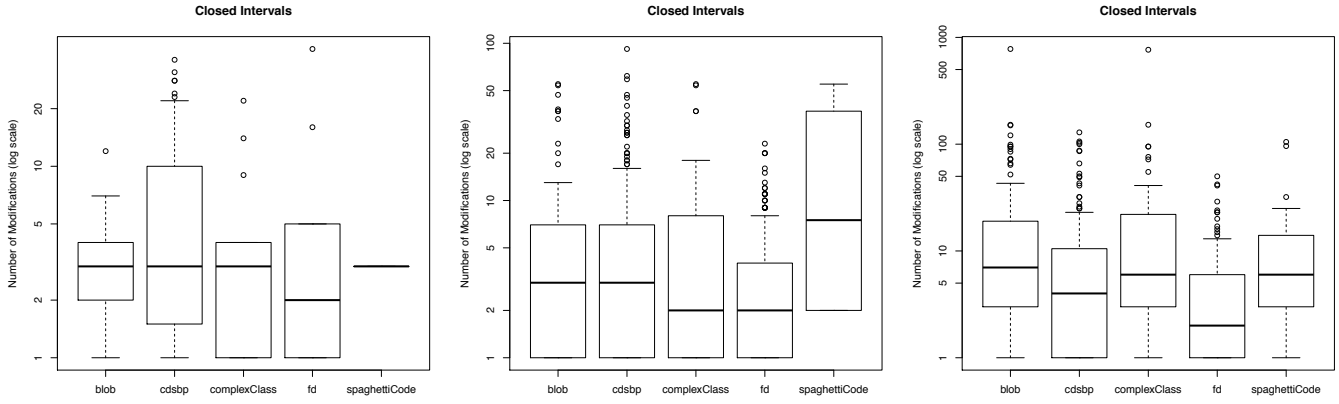
Figure 7 shows the survivability of born-smelly and not-born-smelly artifacts for the CDSBP instances. In

7. As also done for the survival model, for the sake of consistency the data reported in Table 9 exclude code smell instances introduced too close to the end of the analyzed change history



(a) Android (b) Apache (c) Eclipse

Fig. 3: Distribution of number of days a smell remained in the system before being removed.



(a) Android (b) Apache (c) Eclipse

Fig. 4: Distribution of number of commits between a smell introduction and its removal.

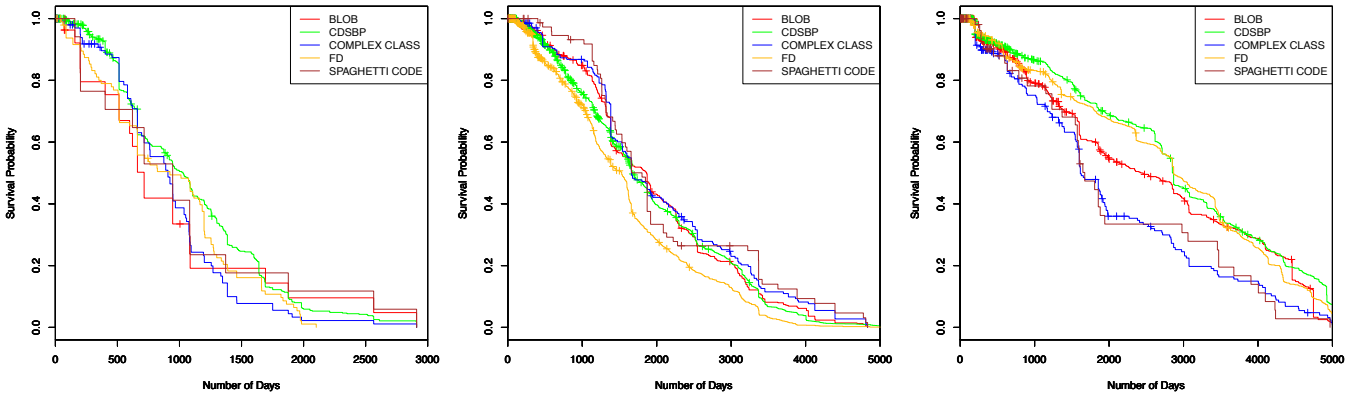
this case, on two of the three analyzed ecosystems the survivability of born-smelly artifacts is actually higher, thus confirming in part the first scenario drawn above. However, when looking at the results for Complex Class instances (Figure 8), such a trend is not present in Android and Apache and it is exactly the opposite in Eclipse (*i.e.,* not-born-smelly-artifacts survive longer than the born-smelly ones). Such trends have also been observed for the other analyzed smells and, in some cases, contradictory trends were observed for the same smell in the three ecosystems (see [81]). Thus, it is not really possible to draw any conclusions on this point.

Summary for RQ$_3$. Most of the studied code smell instances (∼80%) are not removed during the observed system's evolution. When this happens, the removal is generally performed after few commits from the introduction (∼10) and in a limited time period (∼100 days). Overall, we can observe a very high survivability of code smells, with over 50% of smell instances still "alive" after 1,000 days and 1,000 commits from their introduction.

3.4 How do developers remove code smells?

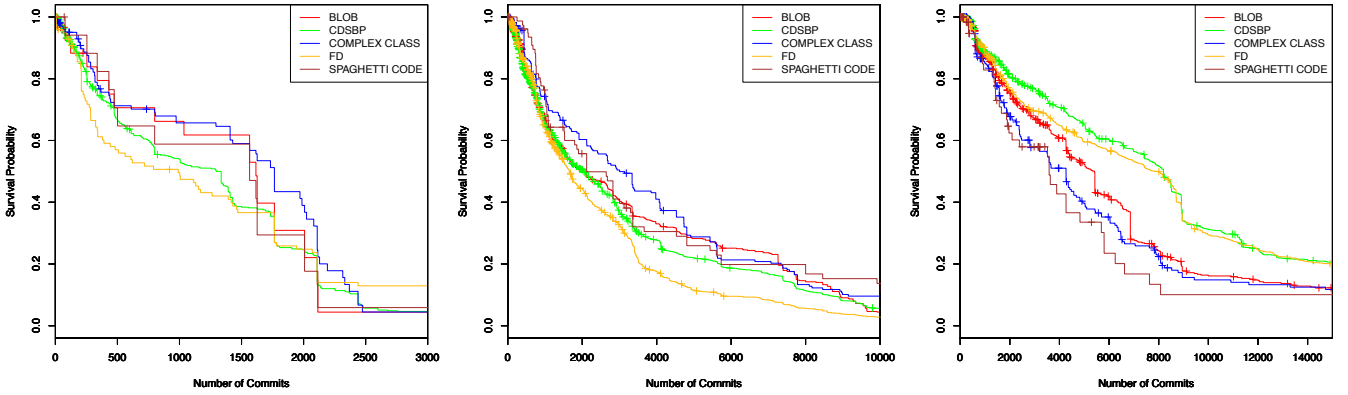
Table 10 shows the results of the open coding procedure, aimed at identifying how developers fix code smells (or, more generally, how code smells are removed from the system). We defined the following categories:

- **Code Removal**. The code affected by the smell is deleted or commented. As a consequence, the code smell instance is no longer present in the system. Also, it is not replaced by other code in the smell-removing-commit.
- **Code Replacement**. The code affected by the smell is substantially rewritten. As a consequence, the code smell instance is no longer present in the system. Note that the code rewriting does not include any specific refactoring operation.
- **Code Insertion**. A code smell instance disappears after new code is added in the smelly artifact. While at a first glance it might seem unlikely that the insertion of new code can remove a code smell, the addition of a new method in a class could, for



(a) Android (b) Apache (c) Eclipse

Fig. 5: Survival probability of code smells in terms of the number of days.



(a) Android (b) Apache (c) Eclipse

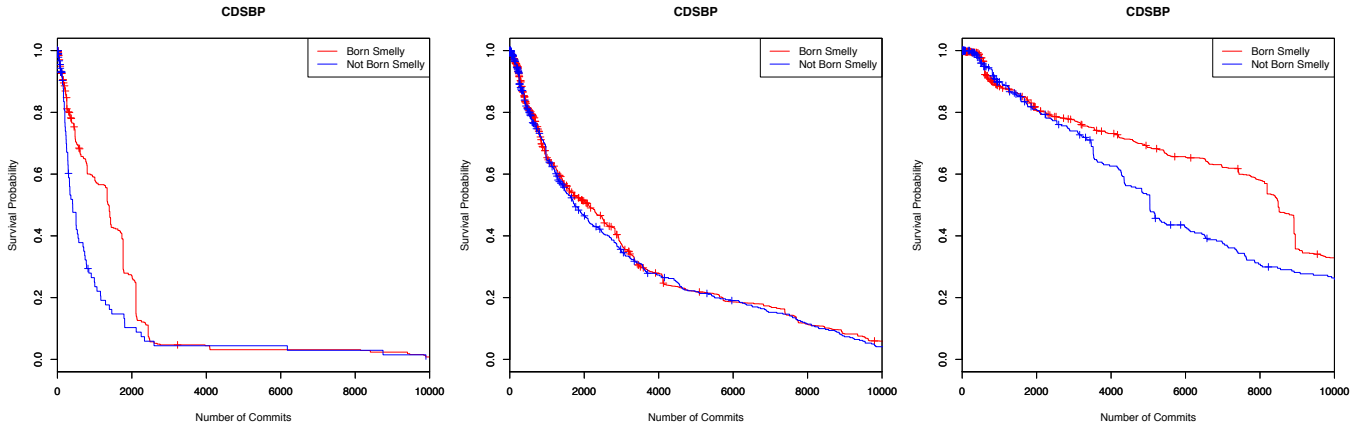
Fig. 6: Survival probability of code smells in terms of the number of commits.

example, increase its cohesion, thus removing a Blob class instance.

- **Refactoring**. The code smell is explicitly removed by applying one or multiple refactoring operations.
- **Major Restructuring**. A code smell instance is removed after a significant restructuring of the system's architecture that totally changes several code artifacts, making it difficult to track the actual operation that removed the smell. Note that this category might implicitly include the ones listed above (*e.g.*, during the major restructuring some code has been replaced, some new code has been written, and some refactoring operations have been performed). However, it differs from the others since in this case we are not able to identify the exact code change leading to the smell removal. We only know that it is a consequence of a major system's restructuring.
- **Unclear**. The GitHub URL used to see the commit diff (*i.e.*, to inspect the changes implemented by the smell-removing-commit) was no longer available at the time of the manual inspection.

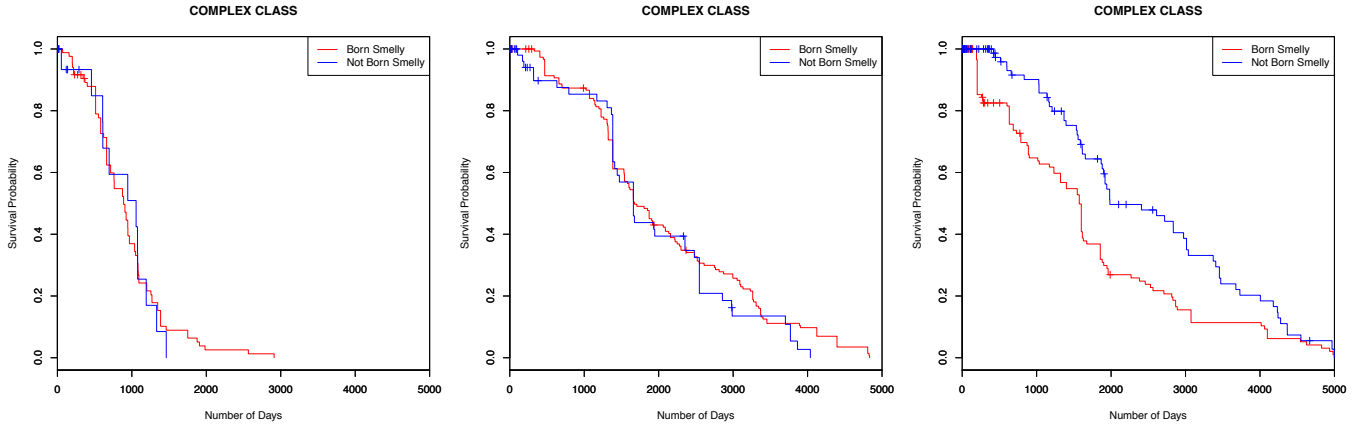
For each of the defined categories, Table 10 shows (i) the absolute number of smell-removing-commits classified in that category; (ii) their percentage over the total of 979 instances and (iii) their percentage computed excluding the *Unclear* instances.

The first surprising result to highlight is that *only 9% (71) of smell instances are removed as a result of a refactoring operation*. Of these, 27 are Encapsulate Field refactorings performed to remove a CDSBP instance. Also, five additional CDSBP instances are removed by performing Extract Class refactoring. Thus, in these five cases the smell is not even actually fixed, but just moved from one class to another. Four Extract Class refactorings have been instead performed to remove four Blob instances. The Substitute Algorithm refactoring has been applied to remove Complex Classes (ten times) and Spaghetti code (four times). Other types of refactorings we observed (*e.g.*, move method, move field) were only represented by one or two instances. Note that this result (*i.e.*, few



(a) Android (b) Apache (c) Eclipse

Fig. 7: Survival probability of CDSBP instances affecting born and not born smelly artifacts.



(a) Android (b) Apache (c) Eclipse

Fig. 8: Survival probability of Complex Class instances affecting born and not born smelly artifacts.

code smells are removed via refactoring operations) is in line with what was observed by Bazrfashan and Koschke [11] when studying how code clones had been removed by developers: They found that most of the clones were removed accidentally as a side effect of other changes rather than as the result of targeted code transformations.

One interesting example of code smell removed using an appropriate refactoring operation relates to the class `org.openejb.alt.config.ConfigurationFactory` of the `Apache Tomee` project. The main responsibility of this class is to manage the data and configuration information for assembling an application server. Until the commit `0877b14`, the class also contained a set of methods to create new jars and descriptors for such jars (through the `EjbJar` and `EjbJarInfo` classes). In the commit mentioned above, the class affected by the Blob code smell has been refactored using Extract Class refactoring. In particular, the developer extracted two new classes from the original class, namely `OpenejbJar` and `EjbJarInfoBuilder` containing the extra functionalities previously contained in `ConfigurationFactory`.

The majority of code smell instances (40%) are simply removed due to the deletion of the affected code components. In particular: Blob, Complex Class, and Spaghetti Code instances are mostly fixed by removing/commenting large code fragments (*e.g.,* no longer needed in the system). In case of Class Data Should Be Private, the code smell frequently disappears after the deletion of public fields. As an example of code smell removed via the deletion of code fragments, the class `org.apache.subversion.javahl.ISVNClient` of the `Apache Subversion` project was a Complex Class until the snapshot `673b5ee`. Then, the developers completely deleted several methods, as explained in the commit message: *"JavaHL: Remove a completely superfluous API"*. This resulted in the consequent removal of the Complex Class smell.

In 33% of the cases, smell instances are fixed by rewriting the source code in the smelly artifact. This

TABLE 10: How developers remove code smells.

Category	# Commits	% Percentage	% Excluding Unclear
Code Removal	329	34	40
Code Replacement	267	27	33
Unclear	158	16	-
Code Insertion	121	12	15
Refactoring	71	7	9
Major Restructuring	33	3	4

frequently occurs in Complex Class and Spaghetti Code instances, in which the rewriting of method bodies can substantially simplify the code and/or make it more inline with object-oriented principles. Code Insertion represents 15% of the fixes. This happens particularly in Functional Decomposition instances, where the smelly artifacts acquire more responsibilities and are better shaped in an object-oriented flavor. Interestingly, also three Blob instances were removed by writing new code increasing their cohesion. An example of Functional Decomposition removed by adding code is represented by the `ExecutorFragment` class, belonging to the `org.eclipse.ocl.library.executor` package of the `Eclipse OCL` project. The original goal of this class was to provide the description of the properties for the execution of the plug-in that allows users to parse and evaluate Object Constraint Language (OCL) constraints. In the commit `b9c93f8` the developers added to the class methods to access and modify such properties, as well as the `init` method, which provides APIs allowing external users to define their own properties.

Finally, in 4% of the cases the smell instance was removed as a consequence of a major restructuring of the whole system.

Summary for RQ$_4$. The main, surprising result of this research question is the very low percentage (9%) of smell instances that are removed as a direct consequence of refactoring operations. Most of the code smell instances (40%) are removed as a simple consequence of the deletion of the smelly artifact. Interestingly, the addition of new code can also contribute to removing code smells (15% of cases).

4 Threats to Validity

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 *DECOR* rules to detect smells. Notice that our re-implementation uses the exact rules defined by Moha *et al.* [51], and has been already used in our previous work [59]. Nevertheless, we are aware that our results can be affected by (i) the thresholds used for detecting code smell instances, and (ii) the presence of false positives and false negatives.

A considerable increment/decrement of the thresholds used in the detection rules might determine changes in the set of detected code smells (and thus, in our results). In our study we used the thresholds suggested in the paper by Moha *et al.* [51]. As for the presence of false positives and false negatives, Moha *et al.* reported for *DECOR* a precision above 60% and a recall of 100% on Xerces 2.7.0. As for the precision, other than relying on Moha *et al.*'s assessment, we have manually validated a subset of the 4,627 detected smell instances. This manual validation has been performed by two authors independently, and cases of disagreement were discussed. In total, 1,107 smells were validated, including 241 Blob instances, 317 Class Data Should Be Private, 166 Complex Class, 65 Spaghetti Code, and 318 Functional Decomposition. Such a (stratified) sample is deemed to be statistically significant for a 95% confidence level and $\pm 10\%$ confidence interval [70]. The results of the manual validation indicated a mean precision of 73%, and specifically 79% for Blob, 62% for Class Data Should Be Private, 74% for Complex Class, 82% for Spaghetti Code, and 70% for Functional Decomposition. In addition, we replicated all the analysis performed to answer our research questions by just considering the smell-introducing commits (2,555) involving smell instances that have been manually validated as true positives. The results achieved in this analysis (available in our replication package [81]) are perfectly consistent with those obtained in our paper on the complete dataset, thus confirming all our findings. Finally, we are aware that our study can also suffer from the presence of false negatives. However, (i) the sample of investigated smell instances is pretty large (4,627 instances), and (ii) the *DECOR*'s claimed recall is very high.

Another threat related to the use of *DECOR* is the possible presence of "conceptual" false positive instances [27], *i.e.,* instances detected by the tool as true positives but irrelevant for developers. However, most of the code smells studied in this paper (*i.e., Blob*, *Complex Class* and *Spaghetti Code*) have been shown to be perceived as harmful by developers [61]. This limits the possible impact of this threat.

The overlap between the quality metrics used when building the linear regression models (**RQ**$_1$) and the metrics used by DECOR for detecting code smells may bias the findings related to when code smells are introduced. In our empirical investigation we are not interested in predicting the presence of code smells over time, but we want to observe whether the trends of quality metrics are different for classes that will become smelly with respect to those that will not become smelly. For this reason, the use of indicators that are used by the detector to identify smells should not influence our observations. However, in most of the cases we avoided the overlap between the metrics used by DECOR and the ones used in the context of **RQ**$_1$. Table 11 reports, for each smell, (i) the set of metrics used by the detector, (ii) the set of metrics evaluated in the context of **RQ**$_1$, and (iii) the overlap between them. We can note that the overlap between the two sets of metrics is often minimal or even empty (*e.g.,* in the case of *Spaghetti Code*). Also, it is worth noting that the detector uses specific thresholds for detecting smells, while in our case we simply look for the changes of metrics' value over time.

TABLE 11: Metrics used by the detector compared to the metrics evaluated in $\mathbf{RQ}_1$.

Code Smell	Metrics used by DECOR	Metrics used in $\mathbf{RQ}_1$	Overlap
Blob	#Methods*, #Attributes* LCOM*, MethodName, ClassName	LOC, LCOM*, WMC, RFC, CBO #Methods*, #Attributes*	3 metrics out of 5 used by DECOR. Note that in this case DECOR also uses textual aspects of the source code that we do not take into account in the context of RQ1.
CDSBP	# Public Attributes	LOC, LCOM, WMC, RFC, CBO #Methods, #Attributes	–
Complex Class	WMC	LOC, LCOM, WMC* RFC, CBO #Methods, #Attributes	1 metric in overlap between the two sets. Note that in the paper we did not only observe the growth of the WMC metric, but we found that other several metrics tend to increase over time for the classes that will become smelly (e.g., LCOM and NOA).
Functional Decomposition	# Private Attributes, #Attributes* Class name	LOC, LCOM, WMC, RFC, CBO #Methods, #Attributes*	1 metric in overlap. Also in this case, we found decreasing trends for all the metrics used in RQ1, and not only for the one used by DECOR.
Spaghetti Code	Method LOC, #Parameters DIT	LOC, LCOM, WMC, RFC, CBO #Methods, #Attributes	–

As explained in Section 2, the heuristic for excluding projects with incomplete history from the *Project startup* analysis may have failed to discard some projects. Also, we excluded the first commit from a project's history involving Java files from the analysis of smell-introducing commits, because such commits are likely to be imports from old versioning systems, and, therefore, we only focused our attention (in terms of the first commit) on the addition of new files during the observed history period. Concerning the tags used to characterize smell-introducing changes, the commit classification was performed by two different authors and results were compared and discussed in cases of inconsistencies. Also, a second check was performed for those commits linked to issues (only 471 out of 9,164 commits), to avoid problems due to incorrect issue classification [3], [33].

The analysis of developer-related tags was performed using the *Git author* information instead of relying on *committers* (not all authors have commit privileges in open source projects, hence observing committers would give an imprecise and partial view of the reality). However, there is no guarantee that the reported authorship is always accurate and complete. We are aware that the *Workload* tag measures the developers' activity within a single project, while in principle one could be busy on other projects or different other activities. One possibility to mitigate such a threat could have been to measure the workload of a developer within the entire ecosystem. However, in our opinion, this would have introduced some bias, *i.e.*, assigning a high workload to developers working on several projects of the same ecosystem and a low workload to those that, while not working on other projects of the same ecosystem, could have been busy on projects outside the ecosystem. It is also important to point out that, in terms of the relationship between *Workload* tag and smell introduction, we obtained consistent results across three ecosystems, which at least mitigates the presence of a possible threat. Also, estimating the *Workload* by just counting commits is an approximation. However, we do not use the commit size because there might be a small commit requiring a substantial effort as well.

The proxies that we used for the survivability of code smells (*i.e.*, the number of days and the number of commits from their introduction to their removal) should provide two different views on the survivability phenomenon. However, the level of activity of a project (*e.g.*, the number of commits per week) may substantially change during its lifetime, thus, influencing the two measured variables.

When studying the survival and the time to fix code smell instances, we relied on DECOR to assess when a code smell instance has been fixed. Since we rely on a metric-based approach, code smell instances whose metrics' values alternate between slightly below and slightly above the detection threshold used by DECOR appear as a series of different code smell instances having a short lifetime, thus introducing imprecisions in our data. To assess the extent of such imprecisions, we computed the distribution of a number of fixes for each code file and each type of smell in our dataset. We found that only between 0.7% and 2.7% (depending on the software ecosystem) of the files has been fixed more than once for the same type of code smell during the considered change history. Thus, such a phenomenon should only marginally impact our data.

Concerning $\mathbf{RQ}_4$, we relied on an open coding procedure performed on a statistically significant sample of *smell-removing commits* in order to understand how code smells are removed from software systems. This procedure involved three of the authors and included open discussion aimed at double checking the classifications individually performed. Still, we cannot exclude imprecision and some degree of subjectiveness (mitigated by the discussion) in the assignment of the *smell-removing commits* to the different fixing/removal categories.

As for the threats that could have influenced the results (*internal validity*), we performed the study by comparing classes affected (and not) by a specific type of smell. However, there can also be cases of classes affected by different types of smells at the same time. Our investigation revealed that such classes represent a minority (3% for Android, 5% for Apache, and 9% for Eclipse), and, therefore, the coexistence of different types of smells in the same class is not particularly interesting to investigate, given also the complexity it would have added to the study design and to its presentation. Another threat could be represented by the fact that a commit identified as a smell-removing-commit (*i.e.*, a commit which fixes a code smell) could potentially introduce another type of smell in the same class. To assess the extent to which this could represent a threat to our study, we analyzed in how many cases this happened in our entire dataset. We found that in only

four cases a fix of a code smell led to the introduction of a different code smell type in the same software artifact.

In $\mathbf{RQ}_2$ we studied tags related to different aspects of a software project's lifetime—characterizing commits, developers, and the project's status itself—we are aware that there could be many other factors that could have influenced the introduction of smells. In any case, it is worth noting that it is beyond the scope of this work to make any claims related to causation of the relationship between the introduction of smells and product or process factors characterizing a software project.

The survival analysis in the context of $\mathbf{RQ}_3$ has been performed by excluding smell instances for which the developers had not "enough time" to fix them, and in particular censored intervals having the *last-smell-introducing commit* too close to the last commit analyzed in the project's history. Table 12 shows the absolute number of censored intervals discarded using different thresholds. In our analysis, we used the median of the smelly interval (in terms of the number of days) for closed intervals as a threshold. As we can observe in Table 12, this threshold allows the removal of a relatively small number of code smells from the analysis. Indeed, we discarded 3 instances (0.4% of the total number of censored intervals) in Android, 203 instances (3.5%) in Apache and 51 instances (1.9%) in Eclipse. This is also confirmed by the analysis of the distribution of the number of days composing the censored intervals, shown in Table 13, which highlights how the number of days composing censored intervals is quite large. It is worth noting that if we had selected the first quartile as threshold, we would have removed too few code smells from the analysis (*i.e.,* 1 instance in Android, 43 in Apache, and 7 in Eclipse). On the other hand, a more conservative approach would have been to exclude censored data where the time interval between the *last-smell-introducing commit* and the last analyzed commit is greater than the third quartile of the smell removing time distribution. In this case, we would have removed a higher number of instances with respect to the median (*i.e.,* 26 instances in Android, 602 in Apache, and 51 in Eclipse). Moreover, as we show in our online appendix [81], this choice would have not impacted our findings (*i.e.,* the achieved results are consistent with what we observed by using the median). Finally, we also analyzed the proportion of closed and censored intervals considering (i) the original change history (no instance removed), (ii) the first quartile as threshold, (iii) the median value as threshold, and (iv) the third quartile as threshold. As shown in our online appendix [81], we found that the proportion of closed and censored intervals after excluding censored intervals using the median value, remains almost identical to the initial proportion (*i.e.,* original change history). Indeed, in most of the cases the differences is less than 1%, while in only few cases it reaches 2%.

Still in the context of $\mathbf{RQ}_3$, we considered a code smell as removed from the system in a commit c_i when DECOR detects it in c_{i-1} but does not detect it in c_i. This might lead to some imprecisions why computing the lifetime of the smells. Indeed, suppose that a file f was affected by the Blob smell until commit c_i (*i.e.,* DECOR still identify f as a Blob class in commit c_i). Then, suppose that f is completely rewritten in c_{i+1} and that DECOR still identifies f as a Blob class. While it is clear that the Blob instance detected in commit c_i is different with respect to the one detected in commit c_{i+1} (since f has been completely rewritten), we are not able to discriminate the two instances since we simply observe that DECOR was detecting a Blob in f at commit c_i and it is still detecting a Blob in f at commit c_{i+1}. This means that (i) we will consider for the Blob instance detected at commit c_i a lifetime longer than it should be, and (ii) we will not be able to study a new Blob instance. Also, when computing the survivability of the code smells we considered the smell introduced only after the *last-smell-introducing-commit* (*i.e.,* we ignored the other commits contributing to the introduction of the smell). Basically, our $\mathbf{RQ}_3$ results are conservative in the sense that they consider the minimum survival time of each studied code smell instance.

The main threats related to the relationship between the treatment and the outcome (*conclusion validity*) are represented by the analysis method exploited in our study. In $\mathbf{RQ}_1$, we used non-parametric tests (Mann-Whitney) and effect size measures (Cliff's Delta), as well as regression analysis. Results of $\mathbf{RQ}_2$ and $\mathbf{RQ}_4$ are, instead, reported in terms of descriptive statistics and analyzed from purely observational point of view. As for $\mathbf{RQ}_3$, we used the Kaplan-Meier estimator [34], which estimates the underlying survival model without making any initial assumption upon the underlying distribution.

Finally, regarding the generalization of our findings (*external validity*) this is, to the best of our knowledge, the largest study—in terms of the number of projects (200)—concerning the analysis of code smells and of their evolution. However, we are aware that we limited our attention to only five types of smells. As explained in Section 2, this choice is justified by the need for limiting the computational time since we wanted to analyze a

TABLE 12: Number of censored intervals discarded using different thresholds. Percentages are reported between brackets.

# Censored Intervals	Android	Apache	Eclipse
Total	708	5780	2709
Discarded using 1st Q.	1 (0.1)	43 (0.7)	7 (0.3)
Discarded using Median	3 (0.4)	203 (3.5)	51 (1.9)
Discarded using 3rd Q.	26 (3.7)	602 (10.0)	274 (10.0)

TABLE 13: Descriptive statistics of the number of days of censored intervals.

Ecosystem	Min	1st Qu.	Median	Mean	3rd Qu.	Max.
Android	3	513	945	1,026	1,386	2,911
Apache	0	909	1,570	1,706	2,434	5,697
Eclipse	0	1,321	2,799	2,629	4,005	5,151

large number of projects. Also, we tried to diversify the types of smells by including smells representing violations of OO principles and "size-related" smells. Last, but not least, we made sure to include smells—such as Complex Class, Blob, and Spaghetti Code—that previous studies indicated to be perceived by developers as severe problems [61]. Our choice of the subject systems is not random, but guided by specific requirements of our underlying infrastructure. Specifically, the selected systems are written in Java, since the code smell detector used in our experiments is able to work with software systems written in this programming language. Clearly, results cannot be generalized to other programming languages. Nevertheless, further studies aiming at replicating our work on other smells, with projects developed for other ecosystems and in other programming languages, are desirable.

5 Related Work

This section reports the literature related to (i) empirical studies conducted to analyze the evolution and (ii) the impact of code smells on maintainability; (iii) methods and tools able to detect them in the source code. Finally we also reported the empirical studies conducted in the field of refactoring.

5.1 Evolution of Smells

A first study that takes into account the way the code smells evolve during the evolution of a system has been conducted by Chatzigeorgiou and Manakos [18]. The reported results show 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. On the same line are the results reported by Peters and Zaidman [63], who show 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 reason for this behavior is given by Arcoverde *et al.* [4], 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 reasons behind this result [4]. The analyses conducted in the context of $\mathbf{RQ}_3$ confirm previous findings on code smell longevity, showing that code smells tend to remain in a system for a long time. Moreover, the results of $\mathbf{RQ}_4$ confirm that refactoring is not the primary way in which code smells are removed.

The evolution of code smells is also studied by Olbrich *et al.* [57], who analyzed the evolution of two types of code smells, namely *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.* [84] 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 is to define a method able to discriminate between *God Class* instances that have been introduced by design and *God Class* instances that were introduced unintentionally. Our study complements the work by Vaucher *et al.* [84], because we look into the circumstances behind the introduction of smells, other than analyzing when they are introduced.

In a closely related field, Bavota *et al.* [7] analyzed the distribution of unit test smells in 18 software systems providing evidence that they are widely spread, but also that most of the them have a strong negative impact on code comprehensibility. On the same line, Tufano *et al.* [80] reported a large-scale empirical study, which showed that test smells are usually introduced by developers when the corresponding test code is committed in the repository for the first time and they tend to remain in a system for a long time. The study conducted in this paper is complementary to the one by Tufano *et al.*, since it is focused on the analysis of the design flaws arising in the production code.

Some related research has been conducted to analyze one very specific type of code smell, *i.e.,* code clones. Göde [31] investigated to what extent code clones are removed through deliberate operations, finding significant divergences between the code clones detected by existing tools and the ones removed by developers. Bazrafshan and Koschke [11] extended the work by Göde, analyzing whether developers remove code clones using deliberate or accidental modifications, finding that the former category is the most frequent. To this aim, the authors classified the changes which removed clones in *Replacement*, *Movement*, and *Deletion*, thus leading to a categorization similar to the one presented in our RQ_4. However, such a categorization is focused on code clones, since it considers specific types of changes aimed at modeling code clones evolution (*e.g.,* whether the duplicated code is placed into a common superclass), while we defined a more generic taxonomy of changes applied by developers for removing a variety of code smells.

Kim *et al.* [41] studied the lifetime of code clones, finding that many clones are fixed shortly, while long-lived code clones are not easy to refactor because they evolve independently. Unlike this work, our analyses revealed that other code smells have generally a long life, and that, when fixed, their removal is usually performed after few commits.

Thummalapenta *et al.* [77] introduced the notion of "late propagation" related to changes that have been propagated across cloned code instances at different times. An important difference between research conducted in the area of clone evolution and code smell evolution is that, differently from other code smells, clone evolution can be seen of the co-evolution of multiple, similar (*i.e.,* cloned) code elements, and such evolution

can either be consistent or inconsistent (*e.g.,* due to missing change propagation) [77]. Such a behavior does not affect the code smells studied in this paper.

Finally, related to the variables investigated in this study, and specifically related to the authorship of smell-related changes, is the notion of code ownership. Rahman and Devanbu [64] studied the impact of ownership and developers' experience on software quality. The authors focused on software bugs analyzing whether "troubled" code fragments (*i.e.,* code involved in a fix) are the result of contributions from multiple developers. Moreover, they studied if and what type of developers' experience matter in this context. The results show that code implicated in bugs is more strongly associated with contributions coming from a single developer. In addition, specialized experience on the target file is shown to be more important than developer's general experience.

5.2 Impact of Smells on Maintenance Properties

Several empirical studies have investigated the impact of code smells on maintenance activities. Abbes *et al.* [1] studied the impact of two types of code smells, namely *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 has also been studied by Yamashita *et al.* [89], who confirmed that developers experience more difficulties in working on classes affected by more than one code smell. The same authors also analyzed the impact of code smells on maintainability characteristics [90]. 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.* [73] 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 has been measured through an Eclipse plug-in, while a regression analysis has 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.

Lozano *et al.* [49] proposed the use of change history information to better understand the relationship between code smells and design principle violations, in order to assess the severity of design flaws. The authors found that the types of maintenance activities performed over the evolution of the system should be taken into account to focus refactoring efforts. In our study, we point out how particular types of maintenance activities (*i.e.,* enhancement of existing features or implementation of new ones) are generally more associated with code smell introduction. Deligiannis *et al.* [25] 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.

Khomh *et al.* [39] demonstrated that the presence of code smells increases the code change proneness. Also, they showed that the code components affected by code smells are more fault-prone with respect to components not affected by any smell [39]. Gatrell and Counsell [30] conducted an empirical study aimed at quantifying the effect of refactoring on change- and fault-proneness of classes. In particular, the authors 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, during the same time period, have not been refactored. The results revealed that classes subject to refactoring have a lower change- and fault-proneness, both considering the time period in which the same classes were not refactored and classes in which no refactoring operations were applied. Li *et al.* [46] 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.* [57] 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 with respect to the other classes. D'Ambros *et al.* [24] also studied the correlation between the *Feature Envy* and *Shotgun Surgery* smells and the defects in a system, reporting no consistent correlation between them. Recently, Palomba *et al.* [61] 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.

5.3 Detection of Smells

Several techniques have been proposed in the literature to detect code smell instances affecting code components, and all of these take their cue from the suggestions provided by four well-known books: [29], [16], [86], [66]. The first one, by Webster [86] defines common pitfalls in Object Oriented Development, going from the project management down to the implementation. Riel [66] describes more than 60 guidelines to rate the

integrity of a software design. The third one, by Fowler [29], describes 22 code smells describing for each of them the refactoring actions to take. Finally, Brown *et al.* [16] define 40 code antipatterns of different nature (i.e., architectural, managerial, and in source code), together with heuristics to detect them.

From these starting points, in the last decade several approaches have been proposed to detect design flaws in source code. Travassos *et al.* [78] define the "reading techniques", a mechanism suggesting manual inspection rules to identify defects in source code. van Emden and Moonen [83] presented jCOSMO, a code smell browser that visualizes the detected smells in the source code. In particular, they focus their attention on two Java programming smells, known as *instanceof* and *typecast*. The first occurs when there are too many *instanceof* operators in the same block of code that make the source code difficult to read and understand. The *typecast* smell appears instead when an object is explicitly converted from one class type into another, possibly performing illegal casting which results in a runtime error. Simon *et al.* [72] provided a metric-based visualization tool able to discover design defects representing refactoring opportunities. For example, a *Blob* is detected if different sets of cohesive attributes and methods are present inside a class. In other words, a *Blob* is identified when there is the possibility to apply *Extract Class* refactoring. Marinescu [50] proposed a metric-based mechanism to capture deviations from good design principles and heuristics, called "detection strategies". Such strategies are based on the identification of *symptoms* characterizing a particular smell and *metrics* for measuring such symptoms. Then, thresholds on these metrics are defined in order to define the rules. Lanza and Marinescu [44] showed how to exploit quality metrics to identify "disharmony patterns" in code by defining a set of thresholds based on the measurement of the exploited metrics in real software systems. Their detection strategies are formulated in four steps. In the first step, the *symptoms* characterizing a smell are defined. In the second step, a proper *set of metrics* measuring these symptoms is identified. Having this information, the next step is to define thresholds to classify the class as affected (or not) by the defined symptoms. Finally, AND/OR operators are used to correlate the symptoms, leading to the final rules for detecting the smells.

Munro [53] presented a metric-based detection technique able to identify instances of two smells, i.e., *Lazy Class* and *Temporary Field*, in the source code. A set of thresholds is applied to some structural metrics able to capture those smells. In the case of *Lazy Class*, the metrics used for the identification are Number of Methods (NOM), LOC, Weighted Methods per Class (WMC), and Coupling Between Objects (CBO). Moha *et al.* [51] introduced DECOR, a technique for specifying and detecting code and design smells. DECOR uses a Domain-Specific Language (DSL) for specifying smells using high-level abstractions. Four design smells are identified by DECOR, namely *Blob*, *Swiss Army Knife*, *Functional Decomposition*, and *Spaghetti Code*. As explained in Section 2, in our study we rely on DECOR for the identification of code smells over the change history of the systems in our dataset because of its good performances both in terms of accuracy and execution time.

Tsantalis and Chatzigeorgiou [79] presented JDeodorant, a tool able to detect instances of *Feature Envy* smells with the aim of suggesting move method refactoring opportunities. For each method of the system, JDeodorant forms a set of candidate target classes where a method should be moved. This set is obtained by examining the entities (i.e., attributes and methods) that a method accesses from the other classes. In its current version JDeodorant[8] is also able to detect other three code smells (i.e., *State Checking*, *Long Method*, and *God Classes*), as well as opportunities for refactoring code clones. Ligu *et al.* [47] introduced the identification of *Refused Bequest* code smell using a combination of static source code analysis and dynamic unit test execution. Their approach aims at discovering classes that really *wants to support the interface of the superclass* [29]. In order to understand what are the methods really invoked on subclass instances, they intentionally override these methods introducing an error in the new implementation (e.g., division by zero). If there are classes in the system invoking the method, then a failure will occur. Otherwise, the method is never invoked and an instance of *Refused Bequest* is found.

Code smell detection can be also formulated as an optimization problem, as pointed out by Kessentini *et al.* [36] as they presented a technique to detect design defects by following the assumption that what significantly diverges from good design practices is likely to represent a design problem. The advantage of their approach is that it does not look for specific code smells (as most approaches) but for design problems in general. Also, in the reported evaluation, the approach was able to achieve a 95% precision in identifying design defects [36]. Kessentini *et al.* [37] also presented a cooperative parallel search-based approach for identifying code smells instances with an accuracy higher than 85%. Boussaa *et al.* [13] proposed the use of competitive coevolutionary search to code-smell detection problem. In their approach two populations evolve simultaneously: the first generates detection rules with the aim of detecting the highest possible proportion of code smells, whereas the second population generates smells that are currently not detected by the rules of the other population. Sahin *et al.* [68] proposed an approach able to generate code smell detection rules using a bi-level optimization problem, in which the first level of optimization task creates a set of detection rules that maximizes the coverage of code smell examples and artificial code smells generated by the second level. The lower level is instead responsible to maximize the number of code smells artificially generated. The empirical evaluation shows that this approach

8. http://www.jdeodorant.com/

achieves an average of more than 85% in terms of precision and recall.

The approaches described above classify classes strictly as being clean or anti-patterns, while an accurate analysis for the borderline classes is missing [40]. In order to bridge this gap, Khomh *et al.* [40] proposed an approach based on Bayesian belief networks providing a likelihood that a code component is affected by a smell, instead of a boolean value as done by the previous techniques. This is also one of the main characteristics of the approach based on the quality metrics and B-splines proposed by Oliveto *et al.* [58] for identifying instances of *Blobs* in source code.

Besides structural information, historical data can be exploited for detecting code smells. Ratiu *et al.* [65] proposed to use the historical information of the suspected flawed structure to increase the accuracy of the automatic problem detection. Palomba *et al.* [60] provided evidence that historical data can be successfully exploited to identify not only smells that are intrinsically characterized by their evolution across the program history – such as Divergent Change, Parallel Inheritance, and Shotgun Surgery – but also smells such as Blob and Feature Envy [60].

5.4 Empirical Studies on Refactoring

Wang *et al.* [85] conducted a survey with ten industrial developers in order to understand which are the major factors that motivate their refactoring activities. The authors report twelve different factors pushing developers to adopt refactoring practices and classified them in *intrinsic motivators* and *external motivators*. In particular, Intrinsic motivators are those for which developers do not obtain external rewards (for example, an intrinsic motivator is the *Responsibility with Code Authorship*, namely developers want to ensure high quality for their code). Regarding the external motivators, an example is the *Recognitions from Others*, i.e., high technical ability can help the software developers gain recognitions.

Murphy-Hill *et al.* [55] analyzed eight different datasets trying to understand how developers perform refactorings. Examples of the exploited datasets are usage data from 41 developers using the Eclipse environment, data from the Eclipse Usage Collector aggregating activities of 13,000 developers for almost one year, and information extracted from versioning systems. Some of the several interesting findings they found were (i) almost 41% of development activities contain at least one refactoring session, (ii) programmers rarely (almost 10% of the time) configure refactoring tools, (iii) commit messages do not help in predicting refactoring, since rarely developers explicitly report their refactoring activities in them, (iv) developers often perform *floss refactoring*, namely they interleave refactoring with other programming activities, and (v) most of the refactoring operations (close to 90%) are manually performed by developers without the help of any tool.

Kim *et al.* [42] presented a survey performed with 328 Microsoft engineers (of which 83% developers) to investigate (i) when and how they refactor code, (ii) if automated refactoring tools are used by them and (iii) developers' perception towards the benefits, risks, and challenges of refactoring [42]. The main findings of the study reported that:

- While developers recognize refactoring as a way to improve the quality of a software system, in almost 50% of the cases they do not define refactoring as a behavior-preserving operation;
- The most important symptom that pushes developers to perform refactoring is low readability of source code;
- 51% of developers manually perform refactoring;
- The main benefits that the developers observed from the refactoring were improved readability (43%) and improved maintainability (30%);
- The main risk that developers fear when performing refactoring operations is bug introduction (77%).

Kim *et al.* [42] also reported the results of a quantitative analysis performed on the Windows 7 change history showing that code components refactored over time experienced a higher reduction in the number of inter-module dependencies and post-release defects than other modules. Similar results have been obtained by Kataoka *et al.* [35], which analyzed the history of an industrial software system comparing the classes subject to the application of refactorings with the classes never refactored, finding a decreasing of coupling metrics.

Finally, a number of works have studied the relationship between refactoring and software quality. Bavota *et al.* [9] conducted a study aimed at investigating to what extent refactoring activities induce faults. They show that refactorings involving hierarchies (e.g., *pull down method*) induce faults very frequently. Conversely, other kinds of refactorings are likely to be harmless in practice. The study on why code smells are introduced ($\mathbf{RQ}_2$) reveal an additional side-effect of refactoring, *i.e.*, sometimes developers introduce code smells during refactoring operations.

Bavota *et al.* also conducted a study aimed at understanding the relationships between code quality and refactoring [10]. In particular, they studied the evolution of 63 releases of three open source systems in order to investigate the characteristics of code components increasing/decreasing their chances of being object of refactoring operations. Results indicate that often refactoring is not performed on classes having a low metric profile, while almost 40% of the times refactorings have been performed on classes affected by smells. However, just 7% of them actually removed the smell. The latter finding is perfectly in line with the results achieved in the context of $\mathbf{RQ}_4$, where we found that only 9% of code smell instances are removed as direct consequence of refactoring operations.

Stroggylos and Spinellis [74] studied the impact of refactoring operations on the values of eight object-

oriented quality metrics. Their results show the possible negative effects that refactoring can have on some quality metrics (e.g., increased value of the LCOM metric). On the same line, Stroullia and Kapoor [75], analyzed the evolution of one system observing a decrease of LOC and NOM (Number of Method) metrics on the classes in which a refactoring has been applied. Szoke *et al.* [76] performed a study on five software systems to investigate the relationship between refactoring and code quality. They show that small refactoring operations performed in isolation rarely impact software quality. On the other side, a high number of refactoring operations performed in block helps in substantially improving code quality. Alshayeb [2] investigated the impact of refactoring operations on five quality attributes, namely adaptability, maintainability, understandability, reusability, and testability. Their findings highlight that benefits brought by refactoring operations on some code classes are often counterbalanced by a decrease of quality in some other classes. Our study partially confirms the findings reported by Alshayeb [2], since we show how in some cases refactoring can introduce design flaws. Moser *et al.* [52] conducted a case study in an industrial environment aimed at investigating the impact of refactoring on the productivity of an agile team and on the quality of the code they produce. The achieved results show that refactoring not only increases software quality but also helps to increase developers' productivity.

6 Conclusion and Lessons Learned

This paper presented a large-scale empirical study conducted over the commit history of 200 open source projects and aimed at understanding *when* and *why* bad code smells are introduced, *what* is their survaivability, and under which circumstances they are removed. These results provide several valuable findings for the research community:

Lesson 1. *Most of the times code artifacts are affected by bad smells since their creation*. This result contradicts the common wisdom that bad smells are generally introduced due to several modifications made on a code artifact. Also, this finding highlights that the introduction of most smells can simply be avoided by performing quality checks at commit time. In other words, instead of running smell detectors time-to-time on the entire system, these tools could be used during commit activities (in particular circumstances, such as before issuing a release) to avoid or, at least, limit the introduction of bad code smells.

Lesson 2. *Code artifacts becoming smelly as consequence of maintenance and evolution activities are characterized by peculiar metrics' trends, different from those of clean artifacts*. This is in agreement with previous findings on the historical evolution of code smells [49], [59], [65]. Also, such results encourage the development of recommenders able to alert software developers when changes applied to code artifacts result in worrisome metric trends, generally characterizing artifacts that will be affected by a smell.

Lesson 3. *While implementing new features and enhancing existing ones, the main activities during which developers tend to introduce smells, we found almost 400 cases in which refactoring operations introduced smells*. This result is quite surprising, given that one of the goals behind refactoring is the removal of bad smells [28]. This finding highlights the need for techniques and tools aimed at assessing the impact of refactoring operations on source code before their actual application (*e.g.,* see the recent work by Chaparro *et al.* [17]).

Lesson 4. *Newcomers are not necessary responsible for introducing bad smalls, while developers with high workloads and release pressure are more prone to introducing smell instances*. This result highlights that code inspection practices should be strengthened when developers are working under these stressful conditions.

Lesson 5. *Code Smells have a high survivability and are rarely removed as a direct consequence of refactoring activities*. We found that 80% of the analyzed code smell instances survive in the system and only a very low percentage of them (9%) is removed through the application of specific refactorings. While we cannot conjecture on the reasons behind such a finding (*e.g.,* the absence of proper refactoring tools, the developers' perception of code smells, *etc.*), our results highlight the need for further studies aimed at understanding why code smells are not refactored by developers. Only in this way it will be possible to understand where the research community should invest its efforts (*e.g.,* in the creation of a new generation of refactoring tools).

These lessons learned represent the main input for our future research agenda on the topic, mainly focusing on designing and developing a new generation of code quality-checkers, such as those described in Lesson 2, as well as investigating the reasons behind developers' lack of motivation to perform refactoring activities, and which factors (*e.g.,* intensity of the code smell) promote/discourage developers to fix a smell instance (Lesson 5). Also, we intend to perform a deeper investigation of factors that can potentially explain the introduction of code smells, other than the ones already analyzed in this paper.

Acknowledgements

We would like to sincerely thank the anonymous reviewers for their careful reading of our manuscript and extremely useful comments that were very helpful in significantly improving the paper. Michele Tufano and Denys Poshyvanyk from W&M were partially supported via NSF CCF-1253837 and CCF-1218129 grants. Fabio Palomba is partially funded by the University of Molise.

References

[1] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, Blob and

Spaghetti Code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. IEEE Computer Society, 2011, pp. 181–190.

[2] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology*, vol. 51, no. 9, pp. 1319 – 1326, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095058490900038X

[3] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "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, October 27-30, 2008, Richmond Hill, Ontario, Canada*. IBM, 2008, p. 23.

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

[5] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits." in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. ACM, 2010, pp. 97–106.

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

[7] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *ICSM*, 2012, pp. 56–65.

[8] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of Apache," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, 2013, pp. 280–289.

[9] G. Bavota, B. D. Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*. Riva del Garda, Italy: IEEE Computer Society, 2012, pp. 104–113.

[10] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1 – 14, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215001053

[11] S. Bazrafshan and R. Koschke, "An empirical study of clone removals," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 50–59.

[12] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. T. Devanbu, "Don't touch my code!: examining the effects of ownership on software quality," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13rd European Software Engineering Conference, Szeged, Hungary, September 5-9, 2011*. ACM, 2011, pp. 4–14.

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

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

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

[16] ——, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998.

[17] O. Chaparro, G. Bavota, A. Marcus, and M. Di Penta, "On the impact of refactoring operations on code quality metrics," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 456–460.

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

[19] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering (TSE)*, vol. 20, no. 6, pp. 476–493, June 1994.

[20] M. Claes, T. Mens, R. Di Cosmo, and J. Vouillon, "A historical analysis of debian package incompatibilities," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 212–223. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820545

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

[22] J. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociology*, vol. 13, no. 1, pp. 3–21, 1990.

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

[24] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Quality Software (QSIC), 2010 10th International Conference on*, July 2010, pp. 23–31.

[25] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, pp. 129 – 143, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121203002401

[26] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *19th International Conference on Software Maintenance (ICSM 2003), 22-26 September 2003, Amsterdam, The Netherlands*, 2003, pp. 23–.

[27] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 609–613.

[28] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[29] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[30] M. Gatrell and S. Counsell, "The effect of refactoring on change and fault-proneness in commercial c# software," *Science of Computer Programming*, vol. 102, no. 0, pp. 44 – 56, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642314005711

[31] N. Göde, "Clone removal: Fact or fiction?" in *Proceedings of the 4th International Workshop on Software Clones*, ser. IWSC '10. New York, NY, USA: ACM, 2010, pp. 33–40. [Online]. Available: http://doi.acm.org/10.1145/1808901.1808906

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

[33] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE / ACM, 2013, pp. 392–401.

[34] E. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American Statistical Association*, vol. 53, no. 282, pp. 457–481, 1958.

[35] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *Software Maintenance, 2002. Proceedings. International Conference on*, 2002, pp. 576–585.

[36] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. ACM, 2010, pp. 113–122.

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

[38] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the 16th Working Conference on Reverse Engineering*. Lille, France: IEEE CS Press, 2009, pp. 75–84.

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

[40] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the 9th International Conference on Quality Software*. Hong Kong, China: IEEE CS Press, 2009, pp. 305–314.

[41] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 187–196, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1095430.1081737

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

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

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

[45] M. M. Lehman and L. A. Belady, *Software Evolution - Processes of Software Change*. Academic Press London, 1985.

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

[47] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013.

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

[49] A. Lozano, M. Wermelinger, and B. Nuseibeh, "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*, ser. IWPSE '07. New York, NY, USA: ACM, 2007, pp. 31–34.

[50] R. Marinescu, "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, 2004, pp. 350–359.

[51] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, pp. 20–36, 2010.

[52] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "Balancing agility and formalism in software engineering," B. Meyer, J. R. Nawrocki, and B. Walter, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, pp. 252–266. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85279-7_20

[53] M. J. Munro, "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, September 2005.

[54] G. C. Murphy, "Houston: We are in overload," in *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*, 2007, p. 1.

[55] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[56] I. Neamtiu, G. Xie, and J. Chen, "Towards a better understanding of software evolution: an empirical study on open-source software," *Journal of Software: Evolution and Process*, vol. 25, no. 3, pp. 193–218, 2013.

[57] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09, 2009, pp. 390–400.

[58] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *Proceedings of the 14^{th} Conference on Software Maintenance and Reengineering*, R. Capilla, R. Ferenc, and J. C. Dueas, Eds. IEEE Computer Society Press, March 2010.

[59] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 268–278.

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

[61] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? A study on developers' perception of bad code smells," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 101–110.

[62] D. L. Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994*. IEEE Computer Society / ACM Press, 1994, pp. 279–287.

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

[64] F. Rahman and P. T. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 491–500.

[65] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "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, 2004, pp. 223–232.

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

[67] J. Rupert G. Miller, *Survival Analysis, 2nd Edition*. John Wiley and Sons, 2011.

[68] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 6:1–6:44, Oct. 2014. [Online]. Available: http://doi.acm.org/10.1145/2675067

[69] G. Scanniello, "Source code survival with the kaplan meier," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 524–527. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2011.6080823

[70] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures.*, second edition ed. Chapman & Hall/CRC, 2000.

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

[72] F. Simon, F. Steinbr, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of 5th European Conference on Software Maintenance and Reengineering*. Lisbon, Portugal: IEEE CS Press, 2001, pp. 30–38.

[73] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Software Eng.*, vol. 39, no. 8, pp. 1144–1156, 2013.

[74] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Proceedings of the 5th International Workshop on Software Quality*, ser. WoSQ '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 10–. [Online]. Available: http://dx.doi.org/10.1109/WOSQ.2007.11

[75] E. Stroulia and R. Kapoor, "Metrics of refactoring-based development: An experience report," in *OOIS 2001*, X. Wang, R. Johnston, and S. Patel, Eds. Springer London, 2001, pp. 113–122. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-0719-4_13

[76] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?" in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 95–104.

[77] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10664-009-9108-x

[78] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *Proceedings of the* 14^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1999, pp. 47–56.

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

[80] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 4–15. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970340

[81] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. (2014) When and why your code starts to smell bad (and whether the smells go away) - replication package. [Online]. Available: http://www.cs.wm.edu/semeru/data/code-smells/

[82] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 403–414. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818805

[83] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE CS Press, Oct. 2002.

[84] S. Vaucher, F. Khomh, N. Moha, and Y. G. Gueheneuc, "Tracking design smells: Lessons from a study of god classes," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering (WCRE'09)*, 2009, pp. 145–158.

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

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

[87] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: recovering links between bugs and changes," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, 2011, pp. 15–25.

[88] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," *2013 IEEE International Conference on Software Maintenance*, vol. 0, pp. 51–60, 2009.

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

[90] A. F. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 306–315.

[91] ——, "Do developers care about code smells? an exploratory survey," in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*. IEEE, 2013, pp. 242–251.

[92] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.