An Empirical Investigation into the Nature of Test Smells

Michele Tufano[1], Fabio Palomba[2], Gabriele Bavota[3], Massimiliano Di Penta[4]
Rocco Oliveto[5], Andrea De Lucia[2], Denys Poshyvanyk[1]
[1] The College of William and Mary, USA — [2] University of Salerno, Italy — [3] Università della Svizzera italiana (USI), Switzerland — [4] University of Sannio, Italy — [5] University of Molise, Italy

ABSTRACT

Test smells have been defined as poorly designed tests and, as reported by recent empirical studies, their presence may negatively affect comprehension and maintenance of test suites. Despite this, there are no available automated tools to support identification and repair of test smells. In this paper, we firstly investigate developers' perception of test smells in a study with 19 participants. The results show that developers generally do not recognize (potentially harmful) test smells, highlighting that automated tools for identifying such smells are much needed. However, to build effective tools, deeper insights into the test smells phenomenon are required. To this aim, we conducted a large-scale empirical investigation aimed at analyzing (i) when test smells occur in source code, (ii) what their survivability is, and (iii) whether their presence is associated with the presence of design problems in production code (code smells). The results indicate that test smells are usually introduced 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. Moreover, we found various unexpected relationships between test and code smells. Finally, we show how the results of this study can be used to build effective automated tools for test smell detection and refactoring.

CCS Concepts

•**Software and its engineering** → **Software evolution;**

Keywords

Test Smells, Mining Software Repositories, Software Evolution

1. INTRODUCTION

Testing represents a significant part of the whole software development effort [9]. When evolving a software system, developers evolve test suites as well by repairing them when needed and by updating them to sync with the new version of the system. To ease developers' burden in writing, organizing, and executing test suites, nowadays appropriate frameworks (*e.g.,* JUnit [9])—conceived for unit testing but also used beyond unit testing—are widely adopted.

Concerning other code artifacts (in the following referred to as "production code") researchers have provided definitions of symptoms of poor design choices, known as "code smells" [13] for which refactoring activities are desirable. Subsequently, researchers developed automated tools to detect them (*e.g.,* [6, 40]), and empirically studied the developers' awareness of such smells [29] as well as the relationship between smells and negative phenomena such as higher fault- and change-proneness [18]. At the same time, researchers have also confuted some common wisdom about "software aging", showing that often the presence of code smells is not necessarily due to repeated changes performed on source code artifacts during their evolution, rather in most cases smells are introduced when such artifacts are created [41]. In conclusion, both researchers and practitioners are carefully looking at the code smell identification, and, nowadays, smell detection is often included as part of continuous integration and delivery processes [5, 12, 39].

A quite related phenomenon to code smells can also occur in test suites, which can be affected by *test smells*. Test smells—defined by van Deursen *et al.* [42]—are caused by poor design choices (similarly to code smells) when developing test cases: the way test cases are documented or organized into test suites, the way test cases interact with each other, with the production code and with external resources are all indicators of possible test smells. For instance, *Mystery Guest* occurs when a test case is using an external resource, such as a file or a database (thus, making the test not self-contained), and *Assertion Roulette* when a test case contains multiple assertions without properly documenting all of them [42].

Empirical studies have shown that test smells can hinder the understandability and maintainability of test suites [7], and refactoring operations aimed at removing them have been proposed [42]. Nevertheless, it is still not clear how developers perceive test smells and whether they are aware of them at all. Also, it is not known whether test smells are introduced as such when test suites are created, or whether test suites become "smelly" during software evolution, and whether developers perform any refactoring operations to remove test smells. Such information is of paramount importance for designing smell detection rules and building automated detection tools to be incorporated in the development process, and especially in the continuous integra-

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970340

tion processes [12], where automated tools could identify test smells and, because of that, make the build fail and notify developers about the presence of the test smells. Highlighting test smells in scenarios where it is known that developers do not want and need to maintain them—*e.g.,* because there is no better solution—would make automated smell detection tools usable, avoiding recommendation overload [27] and even build failures.

Paper contribution. This paper reports a thorough empirical investigation into the perceived importance of test smells and of their lifespan across software projects' change histories. First, we conducted a survey with 19 developers assessing whether developers could recognize instances of test smells in software projects. Such a survey obtained a clear negative result, indicating that, unlike what previously found for code smells [29], there is basically no awareness about test smells, highlighting the need for (semi-) automatic support to aid in detecting these design issues. Thus, we conducted a mining study over the change history of 152 software projects to gather the deeper knowledge needed to design effective test smells detectors. In the context of this study we investigate (i) when test smells are introduced; (ii) how long test smells survive (and whether developers try to remove them); and (iii) whether test smells are related to the presence of smells in production code, and, therefore, there can be synergies in their detection. The achieved results indicate that (i) test smells mostly appear as the result of bad design choices made during the creation of the test classes, and not as the result of design quality degradation over time, (ii) test smells stay in the system for a long time, with a probability of 80% that a test smell would not be fixed after 1,000 days from its introduction, and (iii) complex classes (*e.g.,* Blob classes) in the production code are often tested by smelly test classes, thus, highlighting a relationship existing between code and test smells.

Paper structure. Section 2 describes the survey we performed to investigate developers' perception of test smells. Section 3 details the mining study definition and planning, while the results are reported in Section 4. Threats to the studies' validity are discussed in Section 5. After a discussion of the related work (Section 6), Section 7 concludes the paper and outlines directions for future work.

2. TEST SMELL PERCEPTION

In this section, we report the design and the results of a survey we conducted with the aim of understanding whether developers perceive test smells as design problems. Specifically, we aim at answering the following research question:

- **RQ_0:** *Are test smells perceived by developers as actual design problems?*

Our study focuses on five types of test smells from the catalogue by van Deursen *et al.* [42]:

1. *Assertion Roulette (AR)*: As defined by van Deursen *et al.* this smell *comes from having a number of assertions in a test method that have no explanation* [42]. Thus, if an assertion fails, the identification of the assert that failed can be difficult. Besides removing the unneeded assertions, to remove this smell and make the test more clear an operation of *Add Assertion Explanation* can be applied [42].
2. *Eager Test (ET)*: A test is affected by *Eager Test* when it checks more than one method of the class to be tested [42], making the comprehension of the actual test target difficult. The problem can be solved by applying *Extract Method* refactoring, splitting the test method in order to specialize its responsibilities [13].
3. *General Fixture (GF)*: A test class is affected by this smell when the `setUp` method is too generic and the test methods only access part of it [42]. In other words, the test fixture is not only responsible for setting the common environment for all the test methods. A suitable refactoring to remove the smell is the *Extract Method*, which reorganizes the responsibilities of the `setUp` method [13].
4. *Mystery Guest (MG)*: This smell arises when a test uses external resources (*e.g.,* a file containing test data), and thus it is not self-contained [42]. Tests containing such a smell are difficult to comprehend and maintain, due to the lack of information to understand them. To remove a *Mystery Guest* a *Setup External Resource* operation is needed [42].
5. *Sensitive Equality (SE)*: When an assertion contains an equality check through the use of the `toString` method, the test is affected by a *Sensitive Equality* smell. In this case, the failure of a test case can depend on the details of the string used in the comparison, *e.g.,* commas, quotes, spaces *etc.* [42]. A simple solution for removing this smell is the application of an *Introduce Equality Method* refactoring, in which the use of the `toString` is replaced by a real equality check.

While several other smells exist in the literature [42], we decided to limit our analysis to a subset of such smells in order to focus the questions for survey's participants on a few smell types, allowing to collect more opinions for the same smell. However, we take into account a fairly diverse catalogue of test smells, which are related to different characteristics of test code. As reported in a previous work [8], our selection includes test smells having the greatest diffusion in both industrial and open source projects.

To answer RQ_0, we invited the original developers of five projects from the Apache and Eclipse ecosystems, namely *Apache James Mime4j*, *Apache JSPWiki*, *Apache POI*, *Eclipse Mylyn*, and *Eclipse Platform UI*. These projects represent a subset of those considered in our larger mining study described in Section 3, and they were selected because they contain all types of smells from the considered catalogue. For this study, smells were manually identified by one of the authors and double-checked by another author.

We chose to involve original developers rather than external developers (*i.e.,* developers with no experience with the subject systems) since we wanted to collect the opinions of developers that actually developed the systems under analysis and, therefore, have a good knowledge about the rationale behind the design choices applied during the development of such systems. In total, we invited 298 developers receiving responses from 19 of them: three from *Apache James Mime4j*, one from *Apache JSPWiki*, six from *Apache POI*, one from *Eclipse Mylyn*, and eight from *Eclipse Platform UI*. Note that even though the number of respondents appears to be low (6.3% response rate), our results are close to the suggested minimum response rate for the survey studies, which is defined around 10% [16].

2.1 Survey Questionnaire Design

The general idea behind the study design was to show, to each developer, one test smell instance of each type. This is done to avoid having a long questionnaire that might have discouraged developers to take part in our study. For each test smell instance, the study participants had to look at the source code and answer the following questions:

1. In your opinion, does this class have any design problem? Please, rate your opinion from 1=strongly disagree to 5=strongly agree.
2. If **you agreed or strongly agreed** to the question number 1, please explain what are, in your opinion, the design problems of this class.
3. If **you agreed or strongly agreed** to the question number 1, please explain why the design problem has been introduced.
4. If **you agreed or strongly agreed** to the question number 1, do you think that this class needs to be refactored? Please, rate your opinion from 1=strongly disagree to 5=strongly agree.
5. If **you agreed or strongly agreed** to the question number 4, how would you refactor this class?

The survey was designed to be completed within approximately 30 minutes.

To automatically collect the answers, the survey was hosted using a Web application, *eSurveyPro*[1]. Developers were given 20 days to respond to the survey. Note that the Web application allowed developers to complete the questionnaire in multiple rounds, *e.g.,* to answer the first two questions in one session and finish the rest later. At the end of the response period, we collected developers' answers of the 19 complete questionnaires in a spreadsheet in order to perform data analysis. Note that the developers of the five systems were invited to evaluate only the test smells detected in the system they contribute to.

2.2 Analysis Method

To answer $\mathbf{RQ_0}$ we computed:

1. The distribution of values assigned by developers when evaluating whether the analyzed test classes had a design problem (question #1 of the survey).
2. The percentage of times the smell has been *identified* by the participants. By *identified* we mean cases where participants, besides perceiving the presence of a smell, were also able to identify the exact smell affecting the analyzed test code, by describing it when answering question #2 of the survey. Note that we consider a smell as *identified* only if the design problems described by the participant are clearly traceable onto the definition of the test smell affecting the code component.
3. The distribution of values assigned by developers when evaluating whether the test classes analyzed should be refactored (question #4 of the survey).
4. The percentage of times the refactoring of the test smell has been *identified* by the participants (question #5 of the survey). In this case, by *identified* we mean cases where participants correctly identified how the design problems affecting the test class should be removed.

[1]http://www.esurveyspro.com

Table 1: **Answers for questions #1 and #4.**

Question	Answer				
	1	2	3	4	5
#1	78	1	4	8	4
#4	87	1	1	7	0

Moreover, we collected the answers to question #3 in order to understand the reasons why test smells are introduced.

2.3 Analysis of the Results

Table 1 reports the distribution of values assigned by developers when answering the questions #1 and #4, respectively. We can see that often developers do not perceive test smells as actual problems. Indeed, only in 17 cases (out of the 95 total test smells analyzed by the developers) a design flaw has been identified (*i.e.,* answers to question #1 with value > 1). These answers come from only five developers (out of 19). In these cases, however, participants were often not able to correctly diagnose the test smell affecting the analyzed test code (only in 2% of the cases developers correctly identified a test smell). Moreover, when analyzing the answers to question #4 of the survey, we found that almost always (91% of the cases) participants did not feel that the refactoring activity would be beneficial to improve the design of the considered test classes. For this reason, developers were not able to provide good suggestions for possible refactoring operations (question #5).

The most important feedback we obtained from this study is related to the answers provided when answering question #3. As an example, analyzing an instance of *Eager Test*, a developer from `Apache POI` claimed that *"probably the code was written in a hurry and was never reviewed"*. Also, a developer from `Eclipse Platform UI` claimed that *"the code analyzed was introduced in 2002 and hasn't got much attention"*. This feedback highlights that, even when perceiving design flaws, developers are not able to correctly identify and explain the reasons behind test smell introduction. This constitutes the need for automated tool support in order to alert developers about the presence of test smells in the test code that they produce.

3. TEST SMELL LIFECYCLE: DESIGN

The *goal* of the study is to analyze the change history of software projects, with the *purpose* of investigating when test smells are introduced by developers, what is their survivability, and whether they are associated with code smells in production code. The *context* of the study consists of 152 open source projects belonging to two ecosystems (Apache and Eclipse) for which we investigated the presence and evolution of test smells.

3.1 Research Questions and Context

The study aims at answering the following RQs:

- **$\mathbf{RQ_1}$:** *When are test smells introduced?* This research question aims at assessing whether test smells are introduced as a consequence of maintenance activities performed on test classes or whether they are introduced as soon as the corresponding test class is committed to the repository for the first time. Results of

Table 2: **Ecosystems under analysis.**

Ecosystem	#Proj.	#Classes	KLOC	#Commits	Mean Story Length	Min-Max Story Length
Apache	164	4-5,052	1-1,031	207,997	6	1-15
Eclipse	26	142-16,700	26-2,610	264,119	10	1-13
Overall	**190**	-	-	**472,116**	**6**	**1-15**

Table 3: **Test cases found in the analyzed projects.**

Ecosystem	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Apache	1	9	35	78	96	553
Eclipse	1	15	32	102	106	809

RQ_1 will help understand the kind of automatic detection tools developers need to identify test smells, and the way such tools should be integrated in the development process. Indeed, if test smells are introduced as the result of continuous maintenance and evolution activities, then detectors can be executed periodically, or as a part of a continuous build process, as it happens with approaches proposed in the literature to catch code smells (*e.g.,* [25, 30]). Instead, if test smells are introduced when the test code is written in the first place, then *just-in-time refactoring* tools could be built; such tools should continuously monitor the (test) code written in the IDE, alerting the developer when it is deviating from good design practices, thus avoiding the introduction of the (test) smells in the first place.

- **RQ_2:** *What is the longevity of test smells?* This research question aims at analyzing the lifetime of test smells, with the goal of understanding to what extent they remain in a software project from their introduction until their (possible) removal. Long living test smells are likely to indicate design issues difficult to catch for software developers, thus indicating the need for (semi-)automated detection tools.

- **RQ_3:** *Are test smells associated with particular code smells affecting the production code?* In this research question, we intend to assess whether test smells are usually associated with design problems occurring in production code. While test and code smells have a different nature, uncovering relationships between the two can highlight possible synergies in their detection and refactoring recommendation.

The *context* of the study consists of 190 software projects belonging to two different ecosystems, *i.e.,* those managed by the Apache Software Foundation and by the Eclipse Foundation. Table 2 reports for each ecosystem (i) the number of projects analyzed, (ii) size ranges in terms of the number of classes and KLOC, (iii) the overall number of commits 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. The Apache ecosystem consists of 164 Java systems randomly selected among those available[2], while the Eclipse ecosystem consists of 26 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 guided by the will to consider projects having (i) different scope, (ii) different sizes, and (iii) different architectures, *e.g.,* we have Apache libraries as well as plug-in based architectures in Eclipse projects. From the original dataset, we discarded projects having no test case in their entire change history. This was the case for 36 Apache projects and two Eclipse projects, thus leading to 152 total projects analyzed. Table 3 reports the distribution of number of test cases identified in the 152 analyzed projects. The complete list of projects and the number of test cases identified is available in our (anonymized) online appendix [1].

The study focuses on the five types of test smells also considered in the survey: *Assertion Roulette*, *Eager Test*, *General Fixture*, *Mystery Guest*, and *Sensitive Equality*. Also, since in the context of RQ_3 we assess the possible relationship between test and code smells, in our study we consider the following types of code smells:

1. *Blob Class*: a large class with different responsibilities that monopolizes most of the system's processing [10];
2. *Class Data Should be Private*: a class that exposes its attributes, thus, violating the information hiding principle [13];
3. *Complex Class*: a class having a high cyclomatic complexity [10];
4. *Functional Decomposition*: a class where inheritance and polymorphism are poorly used, declaring many private fields and implementing few methods [10];
5. *Spaghetti Code*: a class without structure that declares long methods without parameters [10].

This analysis is limited to a subset of the smells that exist in the literature [10, 13] due to the computational constraints. However, we preserve 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).

3.2 Data Extraction

To answer our research questions, we firstly cloned the `Git` repositories of the subject software systems. Then, we mined the evolution history of each repository using a tool that we developed, named `HistoryMiner`: it checks out each commit of a repository in chronological order and identifies the code files added and modified in each specific commit. Note that only `java` files are considered as code files, while the remaining files are discarded (*e.g.,* building files, images).

3.2.1 Artifact Lifetime Log

For each commit `HistoryMiner` classifies code files in two sets: *test classes* and *production classes*. Specifically, a file is classified as a *test class* if the corresponding Java Class extends the class `junit.framework.TestCase`. Those code files, which are not classified as *test classes*, are considered as *production classes*. Finally, for each test class identified, our tool analyzes the structural dependencies of the test code to identify the list of associated production classes (*i.e.,* the classes exercised by the test class). All production classes having structural dependencies (*e.g.,* method invocations) with a test class C_t are considered as exercised by C_t.

The output of `HistoryMiner` is an artifact lifetime log for each code file in a project repository. The tool generates two

[2]https://projects.apache.org/indexes/quick.html
[3]https://github.com/eclipse

Table 4: **Rules used to detect test smells** [7].

Name	Abbr.	Description
Assertion Roulette	AR	JUnit classes containing at least one method having more than one assertion statement, and having at least one assertion statement without explanation.
General Fixture	GF	JUnit classes having at least one method not using the entire test fixture defined in the `setUp()` method
Eager Test	EG	JUnit classes having at least one method that uses more than one method of the tested class.
Mystery Guest	MG	JUnit classes that use an external resource (e.g., a file or database).
Sensitive Equality	SE	JUnit classes having at least one assert statement invoking a toString method.

different lifetime logs for *Test Cases* and *Production Classes*. For each test f_{tc} in the repository, the tool outputs a log_{tc} which contains a row for each commit which modified f_{tc}. The i-th row contains the following fields:

- $f_{tc}.name$: the fully qualified name of f_{tc};
- $c_i.id$: the commit hash of the i-th commit which modified f_{tc};
- $c_i.time$: the timestamp when the i-th commit has been performed in the repository;
- TS_{tc}: the list of the test smells affecting f_{tc} at $c_i.time$ (if any);
- PC_{tc}: the list of fully qualified names of production classes having structural dependencies with f_{tc}. We assume these classes to be the ones exercised by f_{tc}.

Similarly, for each production class file f_{pc} in the project repository, the tool outputs a log_{pc}, which contains a row for each commit which modified f_{pc}. The i-th row contains the following fields:

- $f_{pc}.name$: the fully qualified name of f_{tc};
- $c_i.id$: the commit hash of the i-th commit, which modified f_{pc};
- $c_i.time$: the timestamp when the i-th commit has been performed in the repository;
- CS_{pc}: the list of code smells affecting f_{pc} at $c_i.time$ (if any).

With the collected information we are able to identify for a given test file f_{tc} the list of code smells affecting the production classes it tests (PC_{tc}) at any moment in time (this is needed in order to answer RQ$_3$). In particular, given a commit c_i modifying f_{tc}, we retrieve the list of code smells affecting each class $C \in PC_{tc}$ at time $c_i.time$ by:

- Retrieving, among all commits affecting (*i.e.,* adding/-modifying) C, the one (c_j) having the greatest timestamp lower than $c_i.time$ (*i.e.,* the commit affecting C being the "closest" in terms of time to c_i among those preceding c_i).
- Consider the smells affecting C at time $c_j.time$ as the ones affecting it also at time $c_i.time$, when the commit modifying f_{tc} was performed.

3.2.2 Identification of Smells

In the context of our analysis we had to identify test and production code smells at each commit. Given the high number of commits analyzed (*i.e.,* 472,116), a manual detection is practically infeasible. For this reason, during the analysis of each commit `HistoryMiner` used detection rules which identify both test and code smells. Concerning the test smell detection, `HistoryMiner` relied on the approach proposed by Bavota *et al.* [7] to detect the five analyzed test smells. Such an approach applies a heuristic metric-based technique that overestimates the presence of test design flaws with the goal of identifying all the test smell instances (*i.e.,* it targets 100% recall). Table 4 reports the set of rules used by the tool in order to detect instances of test smells. For example, it marks JUnit classes as affected by *General Fixture* those having at least one test method not using the entire test fixture defined in the `setUp()` method. This approach has been shown to have a precision higher than 70% for all detected smells [7]. However, it is not able to recommend refactoring operations to remove the identified smells, thus only providing a very limited support to software developers interested in removing test smells from their system.

As for the code smells detection, we run on each commit an implementation of the *DECOR* smell detector based on the original rules defined by Moha *et al.* [25]. *DECOR* identifies smells using detection rules rooted in 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 [25]; and (ii) it applies simple detection rules that make it very efficient.

3.3 Data Analysis

In this section we describe the data analysis performed to answer each of the three formulated research questions.

3.3.1 RQ$_1$: When Are Test Smells Introduced?

We analyzed each lifetime log for a test file f_{tc} to identify for each test smell TS_k affecting it the following two commits:

- $c_{creation}$: The first commit creating f_{tc} (*i.e.,* adding it in the versioning system);
- c_{smell_k}: The first commit where TS_k is detected in f_{tc} (*i.e.,* the commit in which TS_k has been introduced).

Then, we analyze the distance between $c_{creation}$ and c_{smell_k} in terms of number of commits, considering only the commits which involved f_{tc}. We show the distribution of such distances for each type of test smell as well as for all test smell instances aggregated.

3.3.2 RQ$_2$: What Is the Longevity of Test Smells?

To address RQ$_2$, we need to determine when a smell has been introduced and when a smell disappears from the system. To this aim, given a test smell TS_k affecting a test file f_{tc}, we analyze log_{tc} and formally define two types of commits:

- Smell-introducing commit: the first chronological commit c_i where TS_k has been detected.
- Smell-removing commit: the first chronological commit c_j following the smell-introducing commit c_i in which the test smell TS_k no longer affects f_{tc}.

Given a test smell TS_k, the time interval between the smell-introducing commit and the smell-removing commit

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

is defined as *smelly interval*, and determines the longevity of TS_k. Given a smelly interval for a test smell affecting the file f_{tc} 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:

- #commits: the total number of commits between c_i and c_j;
- #tcChanges: the number of commits between c_i and c_j that modified the test case f_{tc} (a subset of the previously defined set #commits);
- #days: the number of days between the introduction of the smell ($c_i.time$) and its fix ($c_j.time$).

Note that 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 smell-introducing commit but not the smell-removing commit, due to the fact that the file is still affected by the test smell in the last commit. In other words, we can discriminate two different types of smelly intervals in our dataset:

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

The collected data was used to answer the following sub-questions.

RQ$_{2.1}$ How long does it take to fix a test smell? We analyze only the closed smelly intervals, which correspond to the fixed test smell instances. For each type of test smell we show the distribution of #commits, #tcChanges and #days related to the closed smelly intervals.

RQ$_{2.2}$ What is the percentage of test smell instances fixed? We report, for each type of test smell, the percentage of fixed instances (*i.e.,* closed smelly intervals). In doing this, we pay particular attention to the censored intervals related to test smells introduced towards the end of the observable change history of each project repository. This is because for those instances developers might not have had enough time to fix them. Indeed, we compute the percentage of fixed and not fixed instances by progressively removing instances introduced x days before the date of the last commit of the project repository we mined. To define x, we use the results of the previous sub-research question related to the number of days usually needed to fix a test smell. Therefore, we compute the percentages of fixed and not fixed test smell instances using: (i) all the smelly intervals; (ii) excluding censored intervals using x=1^{st} quartile of the #days to fix a test smell; (iii) x=median value; and (iv) x=3^{rd} quartile. It is worth noting that, while in general any test smell instance defining a censored interval could potentially be fixed in the future, this is progressively less likely to happen as the time goes by.

RQ$_{2.3}$ What is the survivability of test smells? We answer this subquestion by relying on *survival analysis* [36], 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.*) [36]. 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 test 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 test smell instance, till its fix (if observed in the available change history). We refer to such a time period as "the lifetime" of a test 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 # of days and the # of commits in the smelly intervals as time variables. We analyzed the survivability of test smells by ecosystem. That is, for each ecosystem, we generated a survival model for each type of test 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 [17] for censored data.

It is important to highlight that, while the survival analysis is designed to deal with censored data, we performed a cleaning of our dataset aimed at reducing possible biases caused by censored intervals before running the analysis. In particular, test 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, as done for the previous sub-question, 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 might not have had "enough time" to fix them). In the paper we report the survival analysis considering all the closed and censored intervals (no instances removed).

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

Table 5: **RQ_1: Number of commits between the creation of the test case and the introduction of a test smell.**

Ecosystem	Smell	Min.	1st Qu.	Median	Mean	3rd Qu.	Max	% Smelly$_{1^{st}}$
Apache	AR	0	0	0	0.13	0	58	96
	ET	0	0	0	0.12	0	245	97
	GF	0	0	0	0.27	0	23	93
	MG	0	0	0	0.49	0	245	93
	SE	0	0	0	0.59	0	62	90
Eclipse	AR	0	0	0	0.37	0	99	96
	ET	0	0	0	0.13	0	26	98
	GF	0	0	0	0.41	0	99	94
	MG	0	0	0	1.13	0	56	91
	SE	0	0	0	1.79	0	102	88

The results of the survival analysis with different thresholds to clean the dataset are available on our online appendix [1].

3.3.3 RQ_3: Are Test Smells Associated with Particular Code Smells Affecting the Production Code?

For each test file f_{tc}, we collect the set of test smells (TS_{tc}) affecting it at the time of its creation as well as the set of code smells ($CS_{PC_{tc}}$) affecting the production classes exercised by f_{tc} (PC_{tc}) at the same time. The goal is to identify patterns of co-occurrence among items belonging to TS_{tc} and items of $CS_{PC_{tc}}$. To do so, we generate a database of transactions, which contains a transaction for each test file f_{tc}. Each transaction contains the test and code smells in TS_{tc} and $CS_{PC_{tc}}$ as items. Note that if TS_{tc} is empty, we insert the item *cleanTC* to represent a test file created without any test smell. Similarly, if $CS_{PC_{tc}}$ is empty, we insert the item *cleanPC*.

We analyze such database of transactions using Association Rule Mining [3] to identify patterns of co-occurrence of test and code smells. In particular, we use the statistical software *R* and the package `arules`.

4. TEST SMELL LIFECYCLE: RESULTS

This section reports the results of our empirical study.

4.1 RQ_1: Introduction of Test Smells

Table 5 reports, for each ecosystem, the statistics of the distribution of the number of commits needed by each test smell type to manifest itself. Also, the last column of Table 5 reports the percentage of test smell instances that have been introduced when the affected test class has been committed for the first time. The results shown in Table 5 clearly highlight one finding: *test smells mostly appear as the result of bad design choices made during the creation of the test classes, and not as the result of design quality degradation during maintenance and evolution activities.* Indeed, for all the considered test smell types the 3rd quartile of the distributions equals zero (*i.e.,* the commit which introduces the smell is the same which introduces the test class). Also, for all test smells and in both ecosystems, at least 88% of the smell instances are introduced when the artifact is created and committed for the first time in the repository (see last column in Table 5). This result is in line with previous findings on code smells in production code [41], and it contradicts the common wisdom for which test smell instances are the result of test code evolution [42]. For this reason, future automatic identification tools should take into account the fact that the lifespan of test smells starts, in most of the cases, with the creation of a test case. IDEs and automatic refactoring tools should pay particular attention to when test classes are firstly created and committed to the repository, since their quality can be compromised by the presence of a design flaw. Promptly suggesting an alternative design (or refactorings) for a newly created test class could significantly increase the chances of having a clean test class.

Table 6: **Percentage of fixed instances in the observed change history using different thresholds to remove censored intervals.**

Smell	History	Apache	Eclipse	Overall
AR	All	0.97	2.03	1.15
	Excluding 1st Q.	0.98	2.03	1.16
	Excluding Median	1.00	2.03	1.18
	Excluding 3rd Q.	1.06	2.05	1.25
ET	All	2.31	1.71	2.19
	Excluding 1st Q.	2.39	1.71	2.24
	Excluding Median	2.39	1.71	2.24
	Excluding 3rd Q.	2.48	1.72	2.32
GF	All	3.29	2.99	3.22
	Excluding 1st Q.	3.32	2.99	3.24
	Excluding Median	3.41	2.99	3.31
	Excluding 3rd Q.	3.70	3.13	3.55
MG	All	2.23	5.39	2.71
	Excluding 1st Q.	2.25	5.39	2.73
	Excluding Median	2.27	5.39	2.74
	Excluding 3rd Q.	2.35	5.39	2.82
SE	All	3.50	7.38	4.21
	Excluding 1st Q.	3.51	7.41	4.22
	Excluding Median	3.53	7.53	4.31
	Excluding 3rd Q.	3.82	7.56	4.52
All	All	2.09	2.57	2.18
	Excluding 1st Q.	2.11	2.57	2.20
	Excluding Median	2.15	2.57	2.24
	Excluding 3rd Q.	2.27	2.62	2.34

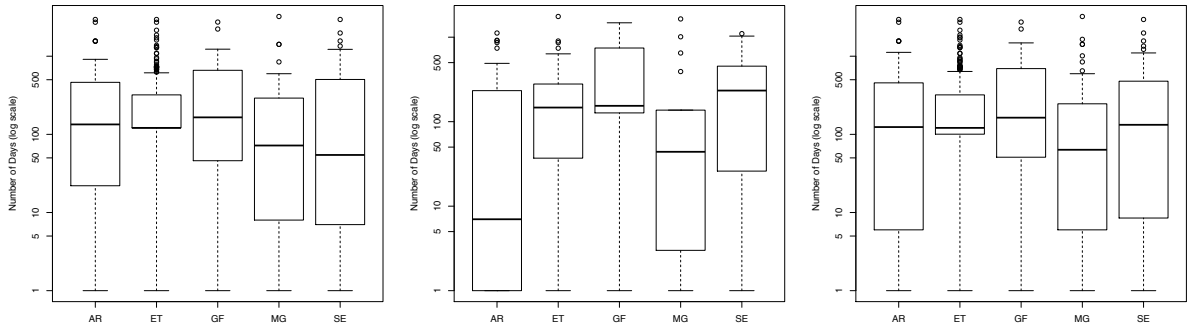
4.2 RQ_2: Longevity of Test Smells

When evaluating the survivability of test smells, we take into account several aspects related to this main research question, as detailed in the following subsections.

4.2.1 How Long Does It Take to Fix a Test Smell?

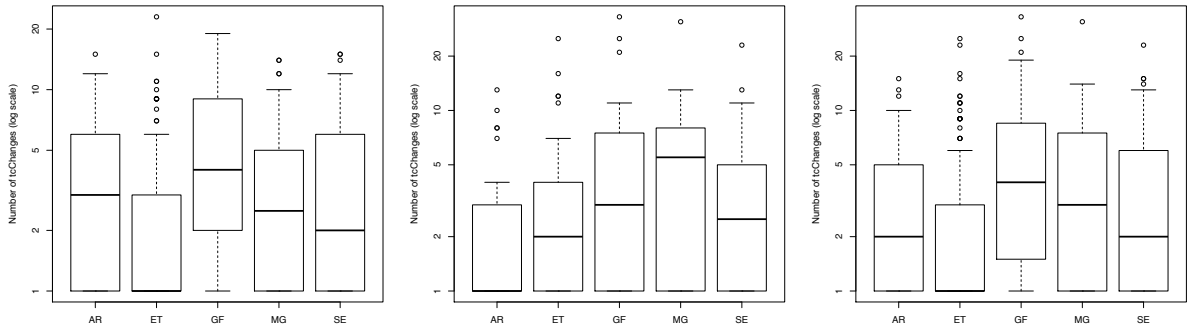
Fig. 1 shows the distribution of the number of days between the introduction of a test smell and its removal. Remember that this data is only available for test smell instances for which we observed both their introduction and their removal over the analyzed change history. The boxplots are organized by the ecosystem with the rightmost one showing the overall results obtained by combining both Apache and Eclipse projects. Overall, we can observe that test smell instances are removed, on average, after 100 days from their introduction[6]. We do not find any significant difference among the different test smell types. We also considered the distribution of the number of modifications a developer performs before fixing a test smell (shown in Fig. 2). As we can see, no more than five modifications involving the affected test class are performed before the test smell disappears. This finding shows on one hand that test classes are not often modified during the history of a software project (on average, five changes in 100 days), and on the other hand that a limited number of modifications is generally required to remove a test smell.

[6]On median, 169 commits and, on average, 671 commits are performed in this time period. We do not report these boxplots due to space constraints. Complete data are available in our online appendix [1].



(a) Apache (b) Eclipse (c) Overall

Figure 1: **Distribution of number of days a test smell remained in the system before being removed.**



(a) Apache (b) Eclipse (c) Overall

Figure 2: **Distribution of number of modifications a smell remained in the system before being removed.**

4.2.2 *What Is the Percentage of Fixed Smell Instances?*

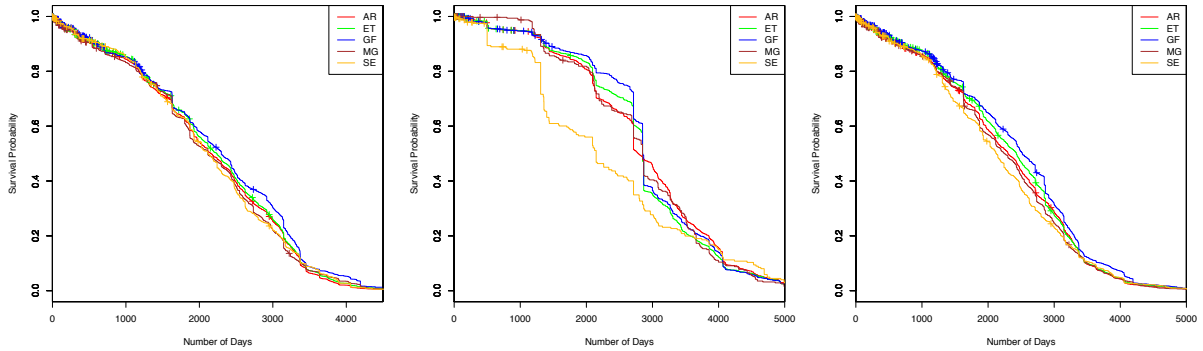
Table 6 shows the percentage of test smell instances fixed in Apache, Eclipse, and in the complete dataset. In a conservative fashion, we show the percentage of fixed instances considering all the change history and progressively removing censored instances (*i.e.,* not fixed yet) introduced too close to the end of the mined change history. It is interesting to notice that the percentage of fixed instances is definitely small. Indeed, only 2.09% and 2.59% of all test smell instances respectively in Apache and Eclipse are fixed (2.18% overall). During this analysis, we noticed that in the Eclipse ecosystem specific test smells appear to have higher fixing percentage (7.38% and 5.39% for Sensitive Equality and Mystery Guest, respectively) with respect to the Apache ecosystem. Even when conservatively discarding instances too close to the end of the observable change history, the percentage of fixed instances remains very small (2.34% overall). This result highlights the poor attention devoted by developers to the removal of test smells that, however, have been shown to hinder code comprehensibility and maintainability [7]. Automated tools, promptly recommending developers on how to refactor test smells, could help in drastically increasing the percentage of fixed instances.

4.2.3 *What Is the Survivability of Test Smells?*

Fig. 3 shows the survival curves, in terms of number of days, for the different test smell types grouped by ecosystems. Overall, test smells have a very high survivability. Indeed, after 1,000 days the probability that a test smell survives (*i.e.,* has not been fixed yet) is 80%. After 2,000 days the survival probability is still around 60%. Fig. 4 reports instead the survivability of test smells when considering the number of commits. Again, the survivability is very high, with 50% of probability that a test class is still affected by a test smell after 2,000 commits from its introduction. Having test smells lasting that long in the systems: (i) further stresses the need for automatic detection tools, and (ii) poses questions on the high maintenance costs the affected test classes could have. We plan to empirically investigate the latter point in our future work.

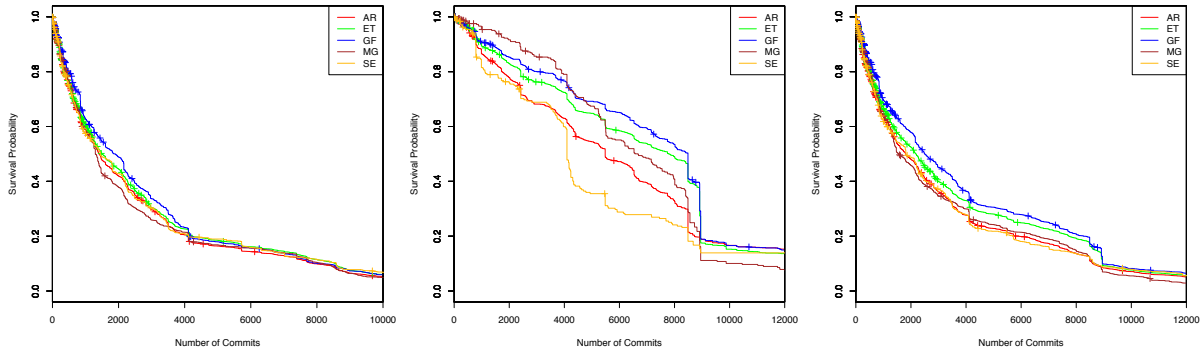
4.3 RQ_3: Test and Code Smells

Table 7 reports the results achieved when applying association rules mining to identify patterns of co-occurrence between test and code smells. We observed several interesting associations relating design flaws occurring in test and production code. A clear case is the first rule shown in Table 7, associating clean test cases to clean production class. This indicates that test classes do not affected by test smells (*i.e.,* clean test classes) usually tests production classes do



(a) Apache (b) Eclipse (c) Overall

Figure 3: **Survival probability of test smells (with respect to the number of days).**



(a) Apache (b) Eclipse (c) Overall

Figure 4: **Survival probability of test smells (with respect to the number of commits).**

Table 7: **RQ_3: Relationships between test and code smells.**

id	lhs	rhs	support	confidence	lift
1	cleanTC	cleanPC	0.05	0.96	1.24
2	AR, SC	BC	0.05	0.99	4.74
3	CDSBP	ET	0.06	0.93	1.17
4	SC	ET	0.09	0.97	1.22
5	BC	ET	0.20	0.95	1.18
6	CDSBP, SC	ET	0.02	0.98	1.22
7	AR, CDSBP	ET	0.07	0.94	1.17
8	CDSBP, GF	BC	0.02	0.77	3.66
9	CDSBP, MG	AR	0.01	0.72	1.14
10	AR, CDSBP	ET	0.04	0.94	1.17
11	AR, SC	CC	0.04	0.85	7.93
12	GF, SE	AR	0.02	0.76	1.22
13	AR, CC, ET, GF, SC	BC	0.02	1.00	4.75
14	GF, SE	ET	0.02	0.87	1.08
15	MG, SE	AR	0.01	0.72	1.15
16	BC, SE	AR	0.02	0.79	1.25
17	BC, SE	ET	0.02	0.97	1.21

not affected by code smells (*i.e.,* clean production classes).

Eager Test (ET) in test classes is often associated with code smells in production code (see rules #3, #4, and #5). Specifically, in our dataset we found several cases in which three code smells, namely *Class Data Should Be Private* (CDSBP), *Spaghetti Code* (SC), and *Blob Class* (BC) co-occur with the *Eager Test* smell. This result is quite reasonable if we consider the definitions of these smells. Indeed, a CDSBP instance appears when a production class violates the information hiding principle by exposing its attributes [13], and often co-occur with code smells related to long or complex code, such as SC and BC [45]. On the other hand, an *Eager Test* appears when a test method checks more than a single method of the class to be tested [42]. Thus, it is reasonable to think that when testing large and complex classes, developers tend to create more complex test methods, exercising multiple methods of the tested class. Similar observations can explain the results achieved for the *Assertion Roulette* (AR) test smell (rules #9 and #12). Indeed, this smell appears when a test case has several assertions with no explanation [42]. When testing complex code, such as the one represented by a *Blob Class*, more assertions which test the behavior of the production class might be needed.

The results of this analysis highlight some interesting findings that should be taken into account for building effective tools for detecting code and test smells. Indeed, on one side the presence of an *Eager Test* could indicate complex and long production code to test. On the other side, complex code to test is likely to trigger design issues in the related test code.

5. THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation and are mainly related to the

measurements we performed. In particular, such threats are related to the way we detect test smells and, for $\mathbf{RQ}_3$, code smells. As explained in Section 3.2, we rely on the implementation of previously proposed approaches to identify test smells [7] and code smells [25], both exhibiting a reasonable precision in smell detection. Clearly, we cannot ignore the fact that some smells have not been detected by the tools and, while for the survey study we have manually validated the considered smells, we are aware that for the second study the reported results could be affected by tools' imprecision. Another threat is related to the observation of the smells' lifetime. We consider a period of time since the first observable commit, which, however, might not correspond to when a file has been created (and this threat can affect the results of $\mathbf{RQ}_1$) until the last observed snapshot (the latter can introduce a threat in $\mathbf{RQ}_2$ because smells could be removed in the future, however survival analysis properly deals with censored data).

Threats to *internal validity* concern factors internal to our study that we cannot control and that could have influenced the results. Above all, the results observed in $\mathbf{RQ}_2$ about smell survival might not only be due to the lack of awareness (observed in the survey study), but also to the lack of necessity to perform (risky) improvements to working test suites.

Threats to *external validity* concern the generalization of our findings. On one hand, the results of the survey are clearly confined to the specificity of our 19 respondents. Although the results are quite consistent, it is possible that other developers might exhibit different levels of awareness about test smells. While large, the mining study surely needs to be extended to other projects beyond the open source ones that we considered (which belong to two ecosystems). Also, it is desirable to extend the study to other test smells beyond the five considered; however, the considered test smells are the most diffused ones [7].

6. RELATED WORK

As well as production code, test code should be designed following good programming practices [37]. During the last decade, the research community spent a lot of effort to define the methods and tools for detecting design flaws in production code [19, 20, 23, 25, 26, 28, 31, 30, 33, 40], as well as empirical studies aimed at assessing their impact on maintainability [2, 4, 11, 18, 22, 29, 34, 35, 38, 45, 44, 21, 32]. However, design problems affecting test code have been only partially explored. The importance to have well designed test code has been originally highlighted by Beck [9], while van Deursen *et al.* [42] defined a catalogue of 11 test smells, *i.e.,* a set of a poor design solutions to write tests, together with refactoring operations aimed at removing them. This catalogue takes into account different types of bad design choices made by developers during the implementation of test fixtures (*e.g.,* `setUp()` method too generic where test methods only access a part of it), or of single test cases (*e.g.,* test methods checking several objects of the class to be tested). Besides the test smells defined by van Deursen *et al.* [42], Meszaros defined other smells affecting test code [24]. Starting from these catalogues, Greiler *et al.* [14, 15] showed that test smells related to fixture set-up frequently occur in industrial projects and, therefore, presented a static analysis tool, namely *TestHound*, to identify fixture related test smells. van Rompaey *et al.* [43] proposed a heuristic structural metric-based approach to identify *General Fixture* and *Eager Test* instances. However, the results of an empirical study demonstrated that structural metrics have lower accuracy while detecting these test smells. Bavota *et al.* [8] conducted an empirical investigation in order to study (i) the diffusion of test smells in 18 software projects, and (ii) their effects on software maintenance. The results of the study demonstrated that 82% of JUnit classes in their dataset are affected by at least one test smell, and that the presence of design flaws has a strong negative impact on the maintainability of the affected classes.

7. CONCLUSION

This paper presented (i) a survey with 19 developers aimed at investigating their perception of test smells as design issues, and (ii) a large-scale empirical study conducted over the commit history of 152 open source projects and aimed at understanding *when* test smells are introduced, what their *longevity* is, and whether they have *relationships with code smells* affecting the tested production code classes. The achieved results provide several valuable findings for the research community:

Lesson 1. *Test smells are not perceived by developers as actual design problems.* Our survey with 19 original developers of five systems showed that developers are not able to identify the presence of test smells in their code. However, recent studies empirically highlighted the negative effect of test smells on code comprehensibility and maintainability [7]. This highlights the importance of investing effort in the development of tools to identify and refactor test smells.

Lesson 2. *In most cases test artifacts are affected by bad smells since their creation.* This result contradicts the common wisdom that test smells are generally due to a negative effect of software evolution and it is inline with what observed for code smells [41]. Also, this finding highlights that the introduction of most smells can simply be avoided by performing quality checks at commit time, or even while the code is written in the IDE by recommending the developer how to "stay away" from bad design practices (*i.e., just-in-time refactoring*). Tools supporting these quality checks could avoid or at least limit the introduction of test smells.

Lesson 3. *Test smells have a very high survivability.* This result further stresses the fact that developers are not catching such problems in the design of their test code. This might be due to (i) the limited time dedicated to refactoring activities, (ii) the unavailability of test smells refactoring tools, or, as shown in our survey, (iii) the fact that developers are not perceiving test smells as bad design practices. The reason behind this result must be further investigated in the future work.

Lesson 4. *There exist relationships between smells in test code and the ones in the tested production code.* Given the different nature of test and code smells, we found this result to be quite surprising. Still, knowing the existence of these relationships could definitively help in better managing both types of smells, by using the presence of test smells as an alarm bell for the possible presence of code smells in the tested classes and *vice versa*.

These lessons learned represent the main input for our future research agenda on the topic, mainly focused on designing and developing a new generation of code quality-checkers, such as those described in Lesson 2.

8. REFERENCES

[1] Online appendix. https://sites.google.com/site/testsmells/, 2016.

[2] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension. In *European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 181–190. IEEE, 2011.

[3] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, June 1993.

[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*, pages 33–36. ACM, 2011.

[5] T. Bakota, P. Hegedüs, I. Siket, G. Ladányi, and R. Ferenc. Qualitygate sourceaudit: A tool for assessing the technical quality of software. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pages 440–445, 2014.

[6] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Automating extract class refactoring: An improved method and its evaluation. *Empirical Softw. Engg.*, 19(6):1617–1664, Dec. 2014.

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

[8] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

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

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

[11] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of bad smells in object-oriented code. In *Int'l Conf. Quality of Information and Communications Technology (QUATIC)*, pages 106–115. IEEE, 2010.

[12] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

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

[14] M. Greiler, A. van Deursen, and M.-A. Storey. Automated detection of test fixture strategies and smells. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 322–331, March 2013.

[15] M. Greiler, A. Zaidman, A. van Deursen, and M.-A. Storey. Strategies for avoiding text fixture smells during software evolution. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, pages 387–396. IEEE, 2013.

[16] R. M. Groves. *Survey Methodology, 2nd edition*. Wiley, 2009.

[17] E. L. Kaplan and P. Meier. Nonparametric estimation from incomplete observations. *Journal of the American Statistical Association*, 53(282):457–481, 1958.

[18] 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*, 17(3):243–275, 2012.

[19] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *Proc. Int'l Conf. on Quality Software (QSIC)*, pages 305–314. IEEE, 2009.

[20] 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.

[21] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc. Domain matters: Bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 232–243, New York, NY, USA, 2014. ACM.

[22] A. Lozano, M. Wermelinger, and B. Nuseibeh. Assessing the impact of bad smells using historical information. In *Proc. of the Int'l workshop on Principles of Software Evolution (IWPSE)*, pages 31–34. ACM, 2007.

[23] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 350–359, 2004.

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

[25] 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 Trans. on Software Engineering*, 36(1):20–36, 2010.

[26] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *Proc. Int'l Software Metrics Symposium (METRICS)*, page 15. IEEE, 2005.

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

[28] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc. Numerical signatures of antipatterns: An approach based on B-splines. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 248–251. IEEE, 2010.

[29] 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*, pages 101–110, 2014.

[30] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto,

D. Poshyvanyk, and A. De Lucia. Mining version histories for detecting code smells. *IEEE Trans. on Software Engineering*, 41(5):462–489, May 2015.

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

[32] F. Palomba, D. D. Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. D. Lucia. Landfill: An open dataset of code smells with public evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 482–485, May 2015.

[33] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman. A textual-based technique for smell detection. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, page to appear. IEEE, 2016.

[34] R. Peters and A. Zaidman. Evaluating the lifespan of code smells using software repository mining. In *Proc. of the European Conf. on Software Maintenance and ReEngineering (CSMR)*, pages 411–416. IEEE, 2012.

[35] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu. Using history information to improve design flaws detection. In *European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 223–232. IEEE, 2004.

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

[37] A. Schneider. Junit best practices. Java World, 2000.

[38] 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.*, 39(8):1144–1156, 2013.

[39] G. Szoke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy. FaultBuster: An automatic code smell refactoring toolset. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*, pages 253–258, 2015.

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

[41] 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 *Int'l Conf. on Softw. Engineering (ICSE)*, pages 403–414. IEEE, 2015.

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

[43] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, Dec 2007.

[44] A. F. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 242–251. IEEE, 2013.

[45] A. F. Yamashita and L. Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 682–691, May 2013.