An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance

Gabriele Bavota[1], Abdallah Qusef[1], Rocco Oliveto[2], Andrea De Lucia[1], David Binkley[3]
[1]*University of Salerno, Fisciano (SA), Italy*
[2]*University of Molise, Pesche (IS), Italy*
[3]*Loyola University Maryland, Baltimore, USA*
gbavota@unisa.it, aqusef@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it, binkley@cs.loyola.edu

***Abstract*—Unit testing represents a key activity in software development and maintenance. Test suites with high internal quality facilitate maintenance activities, such as code comprehension and regression testing. Several guidelines have been proposed to help developers write good test suites. Unfortunately, such rules are not always followed resulting in the presence of *bad test code smells* (or simply *test smells*). Test smells have been defined as poorly designed tests and their presence may negatively affect the maintainability of test suites and production code. Despite the many studies that address code smells in general, until now there has been no empirical evidence regarding test smells (i) distribution in software systems nor (ii) their impact on the maintainability of software systems.**

This paper fills this gap by presenting two empirical studies. The first study is an exploratory analysis of 18 software systems (two industrial and 16 open source) aimed at analyzing the distribution of test smells in source code. The second study, a controlled experiment involving twenty master students, is aimed at analyzing whether the presence of test smells affects the comprehension of source code during software maintenance. The results show that (i) test smells are widely spread throughout the software systems studied and (ii) most of the test smells have a strong negative impact on the comprehensibility of test suites and production code.

***Keywords*-Test smells; Unit testing; Mining software repositories; Controlled experiments**

I. Introduction

Data abstraction, encapsulation, and modularity are key Object-Oriented design principles that assure a set of non-functional quality characteristics. Example characteristics of a software system include maintainability, understandability and ease of evolution [1], [2], [3]. However, even when developers are familiar with OO principles, deadline pressure, too much focus on pure functionality, or just inexperience may lead to violations of these design rules [4].

The presence of bad code smells is symptomatic when developers disobey OO design principles. The term *bad code smells* was coined by Fowler [4] who presented an informal definition of 22 code smells and provided a set of characteristics used as indicators for design flaws with respect to the maintainability of software systems.

Recent studies have proved that the occurrence of bad code smells in a system's source code can significantly reduce its understandability, especially when the source code contains combinations of different bad code smells [5]. In addition, bad code smells increase the likelihood of classes needing to changed to fix a fault [6]. To reduce all of these concerns, specific refactoring operations can be applied to remove bad smells [4].

Bad code smells do not plague only production code, but they are also found in test code such as unit test suites [7]. However, test code has a distinct set of smells (*bad test code smells* or simply *test smells*) that relate to the ways in which test cases are organized, how they are implemented, and how they interact with each other. Similar to bad code smells, test smells are conjectured in the literature to decrease the quality of systems and *ad-hoc* refactoring operations have to be applied to remove them [7].

Despite several studies that consider test smell definitions, identification and refactoring [7], [8], [9], no studies have empirically investigated to what extent test smells are spread in existing software systems nor the impact that they have on program comprehension, a central activity of effective software maintenance and evolution. A good understanding of both the production code and the test code is essential to allow the inspection, maintenance, reuse, and extension of source code.

This paper fills this gap, by presenting two empirical studies. The first study is an exploratory study of 18 software systems (two industrial and 16 open source) aimed at analyzing the distribution of test smells in source code (e.g., how are test smells spread in software systems? Which test smells are the most frequent?). The second study is a controlled experiment involving twenty master students. It is aimed at analyzing whether the presence of test smells affects the comprehension of source code during software maintenance. In the study we asked subjects to perform different program comprehension tasks and we measured the subjects' performance using both correctness and the time spent to perform a task.

Collected data from the first study shows that there is a high diffusion of the test smells in both open source and

978-1-4673-2312-3/12/$31.00

Table I
TEST SMELLS DEFINITION [7]

Name	Description	Possible Effects
Mystery Guest	A test uses external resources (e.g., file containing test data)	Difficulties in test comprehension because of unknown values
Resource Optimism	A test makes assumptions about the state/existence of external resources	Non-deterministic result depending on the state of the resources
Test Run War	A test allocates resources also used by others (e.g., tmp files)	Failures occur when several people run tests simultaneously
General Fixture	A test case fixture is too general and the test methods only access a part of it	Difficulties in test comprehension
Eager Test	A test method checks several methods of the tested object	Difficulties in test comprehension and maintenance
Lazy Test	Several test methods check a method of the tested class using the same fixture	Difficulties maintaining consistency during test maintenance
Assertion Roulette	Several assertions with no explanation within the same test method	If an assertion fails it can be difficult to identify which type it is
Indirect Testing	A test interacts with the object under test indirectly via another object	Difficulties in test maintenance and debugging
For Testers Only	A production class contains methods used only by test methods	Difficulties during production code maintenance and comprehension
Sensitive Equality	The *toString* method is used in assert statements	Failures may occur if the *toString* method is changed
Test Code Duplication	Code clones contained inside the unit tests	Code clones have bad effects on maintainability.

industrial software systems. In addition, the second study provides evidence that test smells have a strong negative impact on program comprehension and maintenance.

The rest of the paper is organized as follows. Section II provides background information on test smells and discusses the related literature. Section III presents the results of the exploratory study, while Section IV presents the results of the controlled experiment. Finally, Section V concludes the paper highlighting directions for future work.

II. BACKGROUND AND RELATED WORK

Code smells in production code and test smells in test code should be avoided by following well defined best practices of good programming. For example, best practices for JUnit tests have been defined by Schneider [10]. However, the quality of unit tests is mainly dependent on the quality of the engineer who wrote the tests [11]. For this and other reasons, such as strict deadlines and developers' inexperience, not all developers follow these guidelines eventually leading to increased maintenance costs especially for unit tests.

Fowler defined a large set of production code bad smells (and refactoring operations to remove them) [4]. However, bad smells affecting test suites are not taken into account in Fowler's work. The importance of refactoring production code and its test suites was highlighted for the first time by Beck [12]. In his book, Beck explains the importance of refactoring and testing activities in Test Driven Development (TDD). When refactoring, the developer must ensure that all unit tests continue to pass, so unit tests may need to be refactored alongside the source code. Therefore, refactoring the code should be followed by refactoring the tests [8].

The concept of test smells – denoting a poorly designed test – was introduced by Van Deursen *et al.* [7]. They identified eleven static test code smells and describe how to remove them through specific refactoring operations. The identified test smells (shown in Table I) refer to tests making inappropriate assumptions on the availability of external resources (*Mystery Guest* and *Resource Optimism*), tests that are long and complex (*General Fixture*, *Eager Test*, *Lazy Test*, *Indirect Testing*), tests containing bad programming decisions (*Assertion Roulette* and *Sensitive Equality*), and tests exposing signs of redundancy (*Test Code Duplication*). The final test smell, *For Testers Only*, is unusual in that, unlike the other ten, it does not appear in the test suite but rather in the production code.

Meszaros [9] described the concept of test smells in a broader context by explaining, in detail, the reasons test smells appear as well as their side effects. Although both Van Deursen *et al.* [7] and Meszaros [9] describe the potential negative effects of each one of the test smells summarized in Table I, no empirical investigation has considered their presence or impact. We fill in this gap by empirically analyzing which of these smells (i) appear in software systems and (ii) which have a negative impact on software maintenance.

Despite the lack of evidence regarding the negative impacts of test smells on software maintenance, there has been work on the automatic identification of test smells. Van Rompaey *et al.* [13] propose a heuristic metric-based approach to identify the *General Fixture* and *Eager Test* bad smells. Reichhart *et al.* [14] propose *TestLint*, a rule-based tool to detect static and dynamic test smells in Smalltalk SUnit code. Breugelmans and Van Rompaey [15] introduce a reverse engineering tool called *TestQ* able to detect test smells through static source code analysis. These authors also identify the need for empirical study to further characterize test smells, their interaction, and their impact on maintainability.

III. TEST SMELLS IN SOFTWARE PROJECTS

This section reports the results of the study we conducted to analyze the distribution of the test smells defined by Van Deursen *et al.* [7] in real software applications. In the study nine of the eleven test smells are considered. Because the test smells *Mystery Guest*, *Resource Optimism* and *Test Run War* are similar and are caused by the same problem (i.e., usage of an external resource), we merge them under the name *Mystery Guest*.

A. Planning

The four *goals* of the study are (i) determining how test smells are spread in software systems; (ii) identifying the most frequent test smells; (iii) investigating the similarities

Table II
OBJECT SYSTEMS USED IN OUR STUDY

System	KLOC	# Classes	#JUnit Classes	#JUnit KLOC	Link
AgilePlanner 2.5	24	299	32	4	ase.cpsc.ucalgary.ca
Apache Ant 1.8.1	108	851	75	8	ant.apache.org
ArgoUML 0.30.1	124	1,430	75	7	argouml.tigris.org
Barcode 2.1.0	14	167	35	3	barcode.sourceforge.net
Colossus 0.13.0	58	304	9	5	colossus.sourceforge.net
DependencyFinder 1.2.1.b3	29	498	120	19	depfind.sourceforge.net
eXVantage 20090507173755	28	348	17	4	research.avayalabs.com
FindBugs 2.0.0	92	1,023	27	2	findbugs.sourceforge.net
Hsqldb 2.2.8	131	443	12	3	hsqldb.org
Jabref 2.7.2	62	544	50	5	jabref.sourceforge.net
JMulTi 4.24	44	192	3	1	www.jmulti.de
JwebUnit 3.0	8	36	30	3	jwebunit.sourceforge.net
Morph 1.1.1	19	262	20	2	morph.sourceforge.net
Optal 1.4	28	356	11	1	opal.sourceforge.net
QuickFixj 1.5.2	19	204	49	6	www.quickfixj.org
Regain 1.7.11	22	223	4	1	regain.sourceforge.net
TripleA 1.3.2.2	97	640	54	9	triplea.sourceforge.net
xBaseJ 20090902	8	36	14	2	xbasej.sourceforge.net
All Systems	915	7,856	637	85	-

and differences in the distribution of test smells in industrial and open source systems; and (iv) investigating the correlation between system characteristics (i.e., production code LOC, number of Classes, JUnit Classes LOC, number of JUnit Classes) and the test smells present.

We analyzed the distribution of the test smells in the 18 software systems reported in Table II. For each system the table reports its name, Kilo Lines Of Code (KLOC) in the production code, number of classes, number of JUnit tests under study, KLOC for the JUnit tests, and a reference link. Two systems, AgilePlanner and eXVantage, are industrial, while the remaining 16 are open source systems. All the object systems are written in Java and have a JUnit test suite.

Having 637 JUnit classes to analyze makes manual detection of the nine test smells prohibitively expensive. For this reason, we developed a simple tool to detect the nine analyzed test smells. The tool outputs a list of candidate JUnit classes (production code classes for *For Testers Only*) potentially exhibiting a test smell. Then, we manually validated the classes suggested by the tool. The validation was performed by three Ph.D. students who individually analyzed and classified as *true positive* or *false positive* all the candidate test smells. Finally, the students performed an open discussion with researchers to resolve any conflicts and reach a consensus on the detected test smells.

To ensure high recall, our detection tool uses very simple rules that overestimate the presence of test smells in the code. This is done at the expense of precision. Even though this choice resulted in a longer list of candidates and thus more expensive manual validation, it was necessary because of our goal to try to not miss any test-smell instances. Table III reports the rules applied by our tool to detect each of the nine analyzed test smells.

Note that we choose to not use existing detection tools because their detection rules are too restrictive and may miss test smell instances. As an example, to detect the *General Fixture* test smell, the *TestQ* detection tool [15] uses a heuristic metrics-based approach, while we simply retrieve as candidates those JUnit classes that have at least one method not using the entire test fixture defined in the *setUp()* method. Moreover, to detect the three test smells, *Eager Test*, *Lazy Test*, and *Indirect Testing*, requires knowing the tested classes of the analyzed JUnit tests. While this information is ignored by *TestQ* during the detection of these three test smells, we exploit test-to-code traceability information previously derived by the same three Ph.D. students.

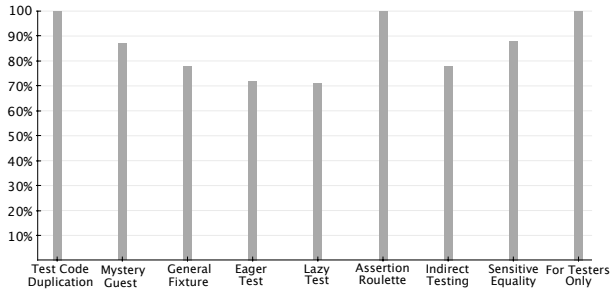

Figure 1. The precision of our tool for detecting test smells.

B. Analysis of the Results

Before presenting the test smells' distribution, it is important to report the precision achieved by the tool used to detect candidate test-smell instances. Figure 1 reports the precision of the tool in detecting each of the nine test smells. We are assuming that recall is 100%, since our detection rules overestimate the presence of test smells in the code. Even using the simple detection rules shown in Table III, the tool achieved very high precision, with the lowest point being the detection of the *Lazy Test* bad smell (71%). Note that the rule applied to detect this smell was very simple (i.e., *"all the JUnit classes having at least two methods using the same method of the tested class"*).

As for the results related to our research goals, Table IV shows the distribution of the test smells in the analyzed object systems[1]. It is worth noting that the results for the *For Testers Only* test smell are not shown in the table since the instances of this smell appear in the production code and not in the test suite. In particular, *For Testers Only* represents a method (or an entire class) in the production code that is used only by some test methods. We found instances of *For Testers Only* in only two of the analyzed systems, AgilePlanner and Apache Ant where three classes in AgilePlanner and twelve in Apache Ant were *For Testers Only*.

As for the other eight test smells, Table IV highlights their significant presence in the analyzed systems. In particular, the two test smells *Eager Test* and *Assertion Roulette* are present in all 18 systems. These test smells are present in the 32% and 62% of the total JUnit classes, respectively. Thus, understanding if they represent an actual problem for

[1]The raw data of our study is available online [16].

Table III
RULES APPLIED TO DETECT TEST SMELLS IN JUNIT CLASSES

Test Smell	Candidate Classes
Mystery Guest	JUnit classes using an external resource (e.g., a file or database)
General Fixture	JUnit classes having at least one method not using the entire test fixture defined in the *setUp()* method
Eager Test	JUnit classes having at least one method that uses more than one method of the tested class
Lazy Test	JUnit classes having at least two methods using the same method of the tested class
Assertion Roulette	JUnit classes containing at least one method having more than one assert statement and at least one assert statement without explanation
Indirect Testing	JUnit classes invoking, besides methods of the tested class, methods of other classes in the production code
For Testers Only	Classes in the production code having structural relationships (e.g., method invocations, inheritance) with only JUnit classes
Sensitive Equality	JUnit classes having at least one assert statement invoking a *toString* method
Test Code Duplication	JUnit classes identified as containing clones by the CCFinder clone detection tool (http://www.ccfinder.net)

Table IV
THE DISTRIBUTION OF TEST SMELLS IN SOFTWARE SYSTEMS

System	#JUnit Tests	JUnit Tests with test smells	Test Code Duplication	Mystery Guest	General Fixture	Eager Test	Lazy Test	Assertion Roulette	Indirect Testing	Sensitive Equality
Agileplanner	32	29 (91%)	1 (3%)	5 (16%)	6 (19%)	11 (34%)	5 (16%)	26 (81%)	3 (9%)	1 (3%)
Apache Ant	75	65 (87%)	20 (27%)	22 (29%)	12 (16%)	38 (51%)	0 (0%)	42 (56%)	10 (13%)	3 (4%)
ArgoUML	75	73 (97%)	12 (16%)	0 (0%)	17 (23%)	18 (24%)	3 (4%)	56 (75%)	14 (19%)	6 (8%)
Barcode	35	29 (83%)	4 (11%)	1 (3%)	0 (0%)	11 (31%)	1 (3%)	25 (71%)	2 (6%)	2 (6%)
Colossus	9	8 (89%)	2 (22%)	0 (0%)	2 (22%)	5 (56%)	1 (11%)	6 (78%)	4 (44%)	0 (0%)
DependencyFinder	120	98 (82%)	40 (33%)	5 (4%)	41 (34%)	30 (25%)	0 (0%)	46 (38%)	10 (8%)	9 (7%)
eXVantage	17	17 (100%)	6 (35%)	1 (6%)	2 (12%)	7 (41%)	1 (6%)	17 (100%)	12 (71%)	1 (6%)
FindBugs	27	24 (89%)	4 (15%)	1 (4%)	6 (22%)	3 (11%)	2 (7%)	19 (70%)	2 (7%)	4 (15%)
Hsqldb	12	11 (92%)	8 (67%)	2 (17%)	2 (17%)	1 (8%)	0 (0%)	7 (58%)	0 (0%)	0 (0%)
Jabref	50	28 (56%)	7 (14%)	6 (12%)	5 (10%)	8 (16%)	2 (4%)	23 (46%)	14 (28%)	1 (2%)
Jmulti	3	3 (100%)	1 (33%)	1 (33%)	0 (0%)	1 (33%)	0 (0%)	3 (100%)	1 (33%)	0 (0%)
JwebUnit	30	18 (60%)	6 (20%)	2 (7%)	3 (10%)	2 (7%)	0 (0%)	15 (50%)	0 (0%)	0 (0%)
Morph	20	8 (40%)	1 (5%)	0 (0%)	2 (10%)	1 (5%)	0 (0%)	5 (25%)	0 (0%)	0 (0%)
Optal	11	10 (91%)	5 (45%)	0 (0%)	0 (0%)	8 (73%)	2 (18%)	9 (82%)	7 (64%)	0 (0%)
QuickFixj	49	42 (86%)	13 (27%)	3 (6%)	6 (12%)	22 (45%)	0 (0%)	34 (69%)	8 (16%)	8 (16%)
Regain	4	2 (50%)	0 (0%)	0 (0%)	0 (0%)	1 (25%)	0 (0%)	1 (25%)	0 (0%)	0 (0%)
Triplea	54	49 (91%)	18 (33%)	0 (0%)	16 (30%)	27 (50%)	6 (11%)	47 (87%)	17 (31%)	3 (6%)
Xbasej	14	13 (93%)	0 (0%)	1 (7%)	1 (7%)	9 (64%)	0 (0%)	13 (93%)	3 (21%)	0 (0%)
All Systems	637	525 (82%)	148 (23%)	50 (8%)	121 (19%)	203 (32%)	23 (4%)	394 (62%)	107 (17%)	38 (6%)

software maintenance is very important. The high diffusion of the *Assertion Roulette* test smell was also previously noted by Qusef *et al.* [11].

Other diffused test smells are *Test Code Duplication* (23%), *General Fixture* (19%), and *Indirect Testing* (17%). On the other hand, the three test smells *Mystery Guest* (8%), *Sensitive Equality* (6%), and *Lazy Test* (4%), have a low diffusion in the analyzed 18 systems. Note that the latter is the test smell affecting the lowest number of systems (nine).

It is also worth noting that among the 637 analyzed JUnit classes, only 112 (18%) are not affected by any test smell. This means that 525 (82%) of the analyzed test suites are affected by at least one test smell. Among these, 219 (34%) are affected by only one test smell, 156 (25%) by two, 83 (13%) by three, 46 (7%) by four, 16 (3%) by five, and 6 (1%) by six. Table V reports the detailed data for each system. An example of a test suite affected by six test smells is the JUnit class *SynchronousPersisterTest* contained in the AgilePlanner project. This class is affected by the *Mystery Guest*, *Test Code Duplication*, *General Fixture*, *Eager Test*, *Lazy Test*, and *Assertion Roulette* test smells.

We also analyzed the co-occurrences of the test smells inside the JUnit classes. In particular, we investigated how often the presence of a test smell in a JUnit class implies the presence of another test smell. Thus, for each test smell T_i we measured the percentage of times that its presence in a JUnit class co-occurs with each other test smell T_j ($i \neq j$).

Table V
TEST SMELLS PRESENCE IN JUNIT CLASSES

System	Number of Test Smells Present						
	0	1	2	3	4	5	6
AgilePlanner	3	16	5	3	3	1	1
Apache Ant	10	23	24	5	8	1	4
ArgoUML	2	37	23	10	2	1	0
Barcode	6	16	10	2	1	0	0
Colossus	1	2	2	2	1	1	0
DependencyFinder	22	46	32	12	7	1	0
eXVantage	0	2	6	5	3	1	0
FindBugs	3	14	4	5	1	0	0
Hsqldb	1	5	3	3	0	0	0
Jabref	22	10	6	7	3	1	1
JMulTi	0	2	0	0	0	1	0
JwebUnit	12	9	8	1	0	0	0
Morph	12	7	1	0	0	0	0
Optal	1	0	2	5	3	0	0
QuickFixj	8	13	14	8	5	2	0
Regain	2	2	0	0	0	0	0
TripleA	6	11	10	15	7	6	0
xBaseJ	1	4	6	1	2	0	0
All Systems	112	219	156	83	46	16	6

Specifically, for each pair of test smells T_i, T_j, we measured the percentage of co-occurrences of T_i and T_j as:

$$\textit{co-occurrences}_{T_{i,j}} = \frac{|T_i \wedge T_j|}{|T_i|}$$

where $|T_i \wedge T_j|$ is the number of co-occurrences of T_i and T_j and $|T_i|$ is the number of occurrences of T_i. Note that

Table VI
TEST SMELLS CO-OCCURRENCES IN THE ANALYZED JUNIT CLASSES

	Test Code Dupl.	Mystery Guest	General Fixture	Eager Test	Lazy Test	Assertion Roulette	Indirect Testing	Sensitive Equality
Test Code Dupl.		14%	28%	42%	6%	**64%**	22%	11%
Mystery Guest	42%		30%	**58%**	6%	**66%**	26%	4%
General Fixture	34%	12%		31%	4%	**67%**	26%	7%
Eager Test	31%	14%	18%		9%	**73%**	30%	8%
Lazy Test	39%	13%	22%	**78%**		**83%**	**61%**	22%
Assertion Roulette	24%	8%	21%	38%	5%		22%	6%
Indirect Testing	31%	12%	29%	**57%**	13%	**82%**		7%
Sensitive Equality	47%	6%	25%	44%	14%	**61%**	19%	

co-occurrences$_{T_{i,j}}$ differs from *co-occurrences*$_{T_{j,i}}$ since the formula's denominator changes from $|T_i|$ to $|T_j|$.

Table VI shows the results. The first result that leaps to the eyes is that all the test smells frequently co-occur with *Assertion Roulette*. However, this is easily explained by the high diffusion of this test smell, which is present in 62% of the JUnit classes. Perhaps, more interesting is that when a *Lazy Test* test smell is present in a JUnit class, then 78% of the time it is accompanied by an *Eager Test*. On the contrary, an *Eager Test* is accompanied by *Lazy Test* only 9% of the time. We manually analyzed these cases, observing that a *Lazy Test* often occurs when there is a method in the tested class that is hard to test because several different test scenarios are needed to exhaustively test the class. Moreover, this kind of method often implements the key responsibility in the tested class, which makes its execution essential to support the test of other methods in the tested class. This results in the introduction of an *Eager Test*. On the other hand, the presence of an *Eager Test* implies the presence of a *Lazy Test* only 9% of the time. We observed that for classes relatively simple to test, developers often write test methods that test several (simple) methods of the tested class. This results in the introduction of an *Eager Test* without a *Lazy Test*. Thus, the 9% of co-occurrences is likely due to the causes described above, where the introduction of both *Lazy Test* and *Eager Test* is forced by the peculiarity of the tested method (i.e., particularly hard to test and essential to support the test of other methods).

Another interesting analysis is a comparison between the two industrial systems and the sixteen open source systems involved in our study. In particular, we analyzed the diffusion of test smells in the two types of systems. We excluded *For Testers Only* from the analysis since we know that it is present only in two systems, the industrial system AgilePlanner and the open source system Apache Ant.

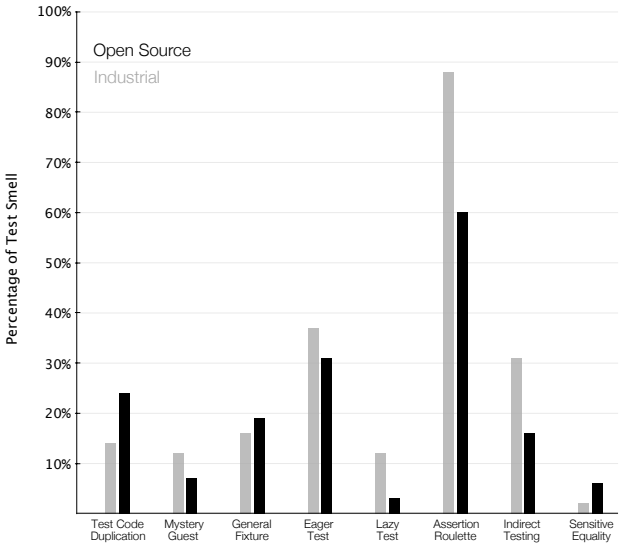

Figure 2. Test smells distribution on industrial and open source systems

Figure 2 depicts the distribution of test-smell instances in open source (black bars) and industrial (gray bars) systems. As is visually evident, the trend is very similar for the two categories of systems. In both industrial and open source systems *Assertion Roulette* is the most frequent test smell with 43 out of 46 (88%) JUnit classes affected in the industrial systems and 351 out of 588 (60%) in the open source systems, followed by *Eager Test*. On the other hand, test smells like *Sensitive Equality* and *Lazy Test* have few instances in both industrial and open source systems. While the trend in the distribution of bad smells is similar, it is interesting to note that for most types of bad smells, the percentage of bad-smell instances is higher in the industrial systems. This is potentially explained by the greater time pressure often found in an industrial context, which would make industrial programmers more prone to bad programming practices. Note that, due to the difficulty in finding industrial repositories, we have analyzed only two industrial systems. This clearly limits the external validity of the results.

Finally, to analyze possible correlations between the systems' characteristics (i.e., production code LOC, number of Classes, number of JUnit Classes, and JUnit Classes LOC) and the test smells' presence, we computed, for each object system, the Pearson product-Moment Correlation Coefficient (PMCC) [17] between the values of each system's characteristic and the percentage of occurrences of each test smell in this system. PMCC is a measure of correlation between two variables X and Y defined in $[-1, 1]$, where 1 represents a perfect positive linear relationship, -1 represents a perfect negative linear relationship, and values in between indicate the degree of linear dependence between X and Y. Cohen *et al.* [17] provided a set of guidelines for the interpretation

Table VII
CORRELATIONS BETWEEN SYSTEMS CHARACTERISTICS AND TEST SMELL PRESENCE (PMCC)

Test Smell	LOC	#Classes	#JUnit Classes	JUnit Classes LOC
Mystery Guest	0.22	-0.04	-0.02	-0.07
General Fixture	0.49	0.44	0.45	0.46
Eager Test	-0.15	-0.15	-0.08	0.02
Lazy Test	0.03	0.12	-0.17	-0.11
Assertion Roulette	0.06	0.01	-0.26	-0.19
Indirect Testing	-0.06	-0.02	-0.22	-0.08
Sensitive Equality	0.18	0.25	0.25	0.24
Test Code Duplication	0.15	0.10	0.04	0.19

of the correlation coefficient. It is assumed that there is no correlation when $0 \leq \rho < 0.1$, small correlation when $0.1 \leq \rho < 0.3$, medium correlation when $0.3 \leq \rho < 0.5$, and strong correlation when $0.5 \leq \rho \leq 1$. Similar intervals also apply for negative correlations.

Table VII reports the PMCC for the analyzed correlations. As we can see there are no strong correlations between the system characteristics and the presence of test smells in their test suites. However, there are some interesting medium correlations as for example those between the *General Fixture* bad smell and the four analyzed systems characteristics. The positive correlations achieved tell us that the bigger the system (in terms of all LOC, number of Classes, number of JUnit Classes, and JUnit Classes LOC) the higher the likelihood that its JUnit classes are affected by the *General Fixture* test smell. This is in someway an expected result, since this test smell generally implies a large test environment declared in the affected test suites. These large test environments are mostly declared when several objects are needed to exhaustively test a class. It is reasonable to think that larger systems are more complex and thus more often require complex test environments in their test suites. As for the other test smells, no interesting correlations were observed with the four investigated systems' characteristics.

Summarizing, the diffusion of the test smells in the 18 analyzed software systems is generally high. Their prevalence highlights the need for empirical evaluation targeted at analyzing test smells influence on the maintainability of the test suites. In our second empirical study (Section IV) we provide such evidence.

C. Threats to Validity

There are three main threats that could affect the validity of our results. First, the tool used to detect candidate test-smell instances could fail to retrieve some of the test-smell instances in the software repositories. To mitigate this concern, we defined the rules used in the detection process (see Table III) to overestimate the presence of test smells in the code, confiding in the subsequent manual validation to eliminate false positives. In fact, given the test smell definitions and the exploited detection rules, our tool will certainly overestimate the test-smell instances.

The second threat is related to the manual validation of the candidate test-smell instances performed by the three Ph.D. students. To avoid biasing the experiment, these students were not aware of the experimental goal. To further mitigate this threat, the students individually validated the test-smell instances and then the list of true positives was finalized in a review meeting attended by the students and academic researchers.

Finally, while the number of analyzed open source systems (16) is sufficient to infer generalizations of the results, more industrial systems are needed beyond the two analyzed in this paper to corroborate our results.

Table VIII
JUNIT CLASSES AND TEST METHODS INVOLVED IN OUR STUDY

Test Smell	JUnit Class	Test Methods
Mystery Guest	ConversionTest (AgilePlanner)	testStoryCardExpectingWierdness
	tmptest (eXVantage)	testInteg
General Fixture	ServerBlackBoxTest (AgilePlanner)	testServerSetup
	ProjectTest (eXVantage)	test1Create
Eager Test	CardModelTest (AgilePlanner)	testUpdatedIterationModelAndStoryCardModel
	NewCFGTest (eXVantage)	test1
Lazy Test	ModelTests (AgilePlanner)	testCreatedStoryCardIteration, testCreatedIteration
	CFGActionTest (eXVantage)	test1, test2, test3
Assertion Roulette	ConversionTest (AgilePlanner)	testIterationExpectingWeirdness
	SessionTraceBitFormatterTest (eXVantage)	testEncodeOversizeId
Indirect Testing	PersisterFactoryTest (AgilePlanner)	testSetPersister
	AboutCFGTest (eXVantage)	test2
Sensitive Equality	CardModelTest (AgilePlanner)	testHashCode
	ASTTest (eXVantage)	test30
Test Code Duplication	SynchronousPersisterTest (AgilePlanner)	Whole class
	CFGActionTest (eXVantage)	Whole class

IV. INFLUENCE OF TEST SMELLS ON MAINTENANCE

This section reports the design and results of the empirical study we conducted to analyze the effects of the eight test smells (*Mystery Guest*, *General Fixture*, *Eager Test*, *Lazy Test*, *Assertion Roulette*, *Indirect Testing*, *Sensitive Equality*, and *Test Code Duplication*) on software maintenance. The test smell *For Testers Only* was not considered since (i) it appears only in two of the systems and (ii) in contrast to the other eight test smells, it affects the production code rather than the test suite.

A. Design

In this study the following research question is investigated:

> *What is the impact of test smells on program comprehension during maintenance activities?*

To answer this research question we performed a controlled experiment involving 20 master students attending the Software Engineering course at the University of Salerno (Italy). We performed the experiment on two systems, AgilePlanner and eXVantage. We chose these systems since (i) both have at least one instance of each test smell (see Table IV) and (ii) they are both industrial systems. The latter reason reduces the possibility of the development environment being a confounding factor.

We randomly selected for each of the eight test smells a JUnit class from each object system having the smell. Table

Table IX
EXPERIMENTAL DESIGN

Group	Test Smells	
	NO	YES
A	AgilePlanner (Lab1)	eXVantage (Lab2)
B	AgilePlanner (Lab2)	eXVantage (Lab1)
C	eXVantage (Lab1)	AgilePlanner (Lab2)
D	eXVantage (Lab2)	AgilePlanner (Lab1)

VIII reports the selected JUnit classes with the methods affected by the test smells. To obtain a version of each selected JUnit class without test smells, we manually refactored them following the guidelines provided by Van Deursen *et al.* [7].

The experiment was organized in two laboratory sessions. Each subject worked on JUnit classes of a system with test smells in one laboratory session and on JUnit classes of the other system without test smells in the other laboratory session. The organization of each group of subjects in each lab session (*Lab1* and *Lab2*) followed the design shown in Table IX. The rows represent the four experimental groups and the columns show the presence or absence of test smells in the analyzed JUnit classes.

The outcome observed in the experiment was the ability of the subjects to correctly understand maintenance activities. This was evaluated by asking subjects to answer a questionnaire (similar to that used by Ricca et al. [18]) consisting of 16 questions (eight for each system). The questions cover all the test smells involved in our evaluation (each question covers one of the eight test smells). Note that the questions were exactly the same (and involved exactly the same JUnit classes) between the questionnaire for the JUnit classes with and without test smells. The only difference was the presence of the test smells in the analyzed test code. The questionnaire was uploaded on a server in the form of a web-application able to (i) automatically balance the subjects among the four experimental groups, (ii) show the questions to the subjects in a random order to reduce the impact of learning effects and subject fatigue, and (iii) measure the time spent by each subject in answering each question.

Figure 3 shows two sample questions from the AgilePlanner questionnaire. The first question was used to evaluate the influence of the *Lazy Test* smell, while the second was used to evaluate the influence of the *Test Code Duplication* smell. The complete questionnaire is available online [16].

B. Variable Selection and Data Analysis

We performed a single factor within-subjects design, where the independent variable (main factor) is the presence or absence of test smells in the analyzed test suites. This variable, denoted **TestSmells**, takes the value *true* or *false*.

The dependent variables are **correctness**, which denotes the ability of a subject to correctly understand the maintenance activities, and **time**, which measures the time spent by the subject in answering each question. To measure the **correctness** we used a combination of the two well

Lazy Test

The method ***getIterations*** implemented inside the class ***ProjectModel*** is tested by the Test Suite ***ModelTests***. If a change is performed to ***getIterations***, which test methods inside ***ModelTests*** should be executed to perform regression testing?

Test Code Duplication

The Test Suite ***SynchronousPersisterTest*** tests the class ***PersisterToXML***. The constructor of ***PersisterToXML*** has been changed, and now takes one more parameter as input. Which lines of code from the Test Suite are impacted by this change?

Figure 3. AgilePlanner: sample questions

known Information Retrieval metrics, recall and precision [19]. These two are defined as follows:

$$recall_s = \frac{\sum_i |answer_{s,i} \cap correct_i|}{\sum_i |correct_i|}\%$$

$$precision_s = \frac{\sum_i |answer_{s,i} \cap correct_i|}{\sum_i |answer_{s,i}|}\%$$

where $answer_{s,i}$ is the set of answers given by subject s to question i and $correct_i$ is the set of correct answers expected for the question i. Note that the aggregate measures defined above differ from mean average precision and mean average recall because they take into account the cases where a subject does not provide an answer to a given question [20]. Finally, recall and precision measure two different (but related) concepts, and thus we use their harmonic mean (i.e., F-measure [19]) to obtain a balance between them when measuring **correctness**.

As for the **time**, we measured (in seconds) the time spent by the subjects in answering each question. In this way, it is possible to determine if the time needed to answer the questions related to test suites with test smells was higher than that needed when test smells were not present.

Because the data did not follow a normal distribution, the non-parametric Wilcoxon test [21] was used to analyze the differences exhibited by subjects working with and without test smells for both **correctness** and **time**. Moreover, because each subject performed a task on two different systems (AgilePlanner or eXVantage) analyzing test suites with or without test smells (i.e., **TestSmells** was *true* for one system and *false* for the other), a paired test was used. Differences are considered statistically significant at $\alpha = 0.05$ level. We also estimated the magnitude of the effect of the main treatment on the dependent variables using the

Table X
DESCRIPTIVE STATISTICS OF **CORRECTNESS** AND **TIME** BY TEST SMELL

Test Smell	Smell not present (**TestSmells** = *false*)						Smell present (**TestSmells** = *true*)					
	correctness			**time**			**correctness**			**time**		
	Mean	Median	St. Dev.	Mean	Median	St. Dev.	Mean	Median	St. Dev.	Mean	Median	St. Dev.
Mystery Guest	0.83	1.00	0.37	290	276	120	0.22	0.00	0.39	347	292	182
General Fixture	0.84	0.88	0.23	270	168	257	0.63	0.62	0.22	274	245	175
Eager Test	0.88	1.00	0.16	176	147	189	0.52	0.46	0.29	354	294	245
Lazy Test	0.97	1.00	0.11	235	181	213	0.86	1.00	0.32	250	239	135
Assertion Roulette	0.90	1.00	0.31	272	229	215	0.00	0.00	0.00	335	276	230
Indirect Testing	0.87	1.00	0.31	219	143	166	0.66	0.48	0.37	242	256	121
Sensitive Equality	0.95	1.00	0.22	176	203	112	0.58	1.00	0.49	197	159	119
Test Code Duplication	0.90	1.00	0.24	227	182	145	0.72	0.80	0.33	258	238	107

Table XI
WILCOXON TEST FOR **CORRECTNESS** AND **TIME** BY TEST SMELL

Test Smell	**correctness**					**time**				
	NoTestSmell$FM - TestSmell$FM			p-value	effect size	NoTestSmell$Time - TestSmell$Time			p-value	effect size
	Mean	Median	St. Dev.			Mean	Median	St. Dev.		
Mystery Guest	0.62	1.00	0.60	$<$ **0.01**	1.03	-57	-22	336	0.27	-0.17
General Fixture	0.21	0.21	0.26	$<$ **0.01**	0.79	-5	-22	327	0.43	-0.01
Eager Test	0.37	0.34	0.37	$<$ **0.01**	0.98	-178	-207	326	$<$ **0.01**	-0.55
Lazy Test	0.12	0.00	0.35	0.11	0.33	-16	-18	264	0.23	-0.06
Assertion Roulette	0.90	1.00	0.31	$<$ **0.01**	2.92	-63	-82	383	0.28	-0.17
Indirect Testing	0.21	0.33	0.54	0.05	0.39	-24	-12	204	0.32	-0.12
Sensitive Equality	0.37	0.00	0.48	$<$ **0.01**	0.76	-20	-69	161	0.31	-0.13
Test Code Duplication	0.18	0.14	0.41	**0.01**	0.44	-31	-81	175	0.29	-0.17

Cohen d effect size [17]. The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$ and large for $d \geq 0.8$ [17].

Finally, to better assess the effect of the test smells on the subjects' performance, it is necessary to consider other factors (called co-factors) that may impact the results. In the context of our study, we identify the following co-factors:

- **System**: since our experiment used two different systems, there is the risk that they may have confounding effect with the main factor. For this reason we considered the analyzed system as a co-factor.
- **Lab**: as explained before, the experiment was organized in two laboratory sessions. Although the experimental design limits learning and fatigue effects, it is still important to analyze whether subjects perform differently across subsequent lab sessions.

To analyze the effects of the co-factors on subject performance and their interaction with the main factor we used the two-way Analysis of Variance (ANOVA) [21].

C. Analysis of the Results

Table X shows descriptive statistics for the dependent variables, **correctness** and **time** separated by test smell presence. For all the analyzed JUnit tests the subjects achieved a higher correctness on the version without the test smells. Moreover, for six out of eight test smells the difference in terms of correctness is statistically significant (see Table XI). Also the analysis of the effect size confirms that the impact of this six test smells on the correctness is strong. In particular, for four test smells, *Mystery Guest*, *General Fixture*, *Eager Test*, and *Assertion Roulette*, the effect size is large (≥ 0.8) while for the remaining two (i.e., *Sensitive Equality* and *Test Code Duplication*) is medium (≥ 0.5).

Finally, the *Lazy Test* and *Indirect Testing* test smells have a negative impact on the correctness achieved by the subjects (see Table X) although it is not statistically significant (p-value ≥ 0.05). In particular, the *Lazy Test* smell does not seem to have a strong impact on program comprehension and maintenance.

Assertion Roulette deserves specific consideration. From the results reported in Table X it is clear that in the presence of this test smell subjects were not able to perform the required maintenance activity (F-Measure always equals zero). In particular, we required subjects to identify which line of code in a test suite generated a particular error trace. It is worth noting that this test smell *"comes from having a number of assertions in a test method that have no explanation"* [7] and thus if one of the assertions fails it is difficult to identify which one it is since no explanation is present in the reported error trace. This is the cause of the zero F-Measure achieved by the subjects in presence of this test smell against 90% without it. This is of particular importance, because *Assertion Roulette* is by far the most frequent test smell in the 18 projects analyzed in Section III, occurring in 62% of the JUnit tests.

Another interesting case is the *Mystery Guest* test smell. In this case the presence of this test smell in the test suite lowered the average correctness by over 60 percentage points (from 83% to 22%). A test suite affected by this smell *"uses external resources, such as a file containing test data"* [7]. In our questionnaire we asked the subjects what changes should be applied in the test suite to modify the test data. In the test suite with the *Mystery Guest* the test data were

read from an XML file, while in the version without test smell an Inline Resource Refactoring [7] had been applied, putting the test data inside a String defined in the test suite. As highlighted by Tables X and XI the effect of this simple refactoring was dramatic.

Concerning **time**, Table X shows that the time spent by the subjects was generally higher in presence of test smells. The strongest difference is seen in presence of the *Eager Test* smell, occurring when *"a test method checks several methods of the object to be tested"* [7]. In this case we asked the subjects to identify the methods tested by a test method representing an *Eager Test*. Clearly, since this smell was removed using the Extract Method Refactoring [4], which separates the test code into several test methods that each only test one method, the time needed to answer the question in absence of the test smell was considerably lower. Note that this test smell is also the only one for which we had a statistically significant difference between the time spent in the analysis of JUnit tests with and without test smell (see Table XI).

Summarizing, the results show that test smells have a strong negative impact on the maintainability of the affected test suites in terms of both accuracy and time. This is true for all the analyzed test smells except for the *Lazy Test* for which we did not observe meaningful differences in the subject performance.

D. Threats to Validity

In the following we discuss threats that could affect the validity of our findings.

Even though the chosen design aims to mitigate learning and fatigue effects, there is still the risk that, during labs, subjects might have learned how to improve their performance. We tried to limit this effect by means of a preliminary training phase performed through a two-hours seminar about the JUnit framework. In addition, since the subjects worked on two different systems, there is the risk that one system might be easier than the other. For this reason, as explained in Section IV-A, we analyzed the effect of these two co-factors, **Lab** and **System**, and their interaction with the main factor through the ANOVA test. The analysis did not reveal any significant influence of either co-factor nor any significant interaction between the main factor and the two co-factors.

Another possible threat is represented by the questions chosen to test the effects of the test smells on software maintenance. For each test smell we tried to include in our questionnaire a question focused as much as possible on maintenance activities involving the test smell. However, a set of different questions might lead to different results.

During the statistical analysis of the results we paid attention to the assumptions made by statistical tests. Whenever the conditions necessary to use a parametric test did not hold, an appropriate non-parametric test, most often the Wilcoxon test for paired analyses was used. We verified these conditions using the non-parametric Wilk-Shapiro test [21]. We also used the parametric ANOVA test to analyze the effect of the co-factors even though the distribution was not normal. This is reasonable because the ANOVA test is a very robust test [22]. In addition, even when the data was not normally distributed we can relax the normality assumption under the law of large numbers, which states that with a population higher than 100 (our population is 320 = 20 subjects $\times$ 16 questions) it is safe to relax the normality assumption [23].

The controlled experiment involved Master students attending the Software Engineering course. The students had good knowledge of Object Oriented programming and testing, and a week before the experiment, they attended a two hour seminar about the JUnit framework. As highlighted by Arisholm and Sjoberg [24] the difference between students and professionals is not always easy to identify. Nevertheless, there are several differences between industrial and academic contexts. For these reasons, we plan to replicate the experiment with industrial subjects to corroborate our findings.

During the controlled experiment students analyzed the source code by using the web-application we developed. On one hand, this avoided to confound the results with how familiar subjects were with a given IDE. On the other hand, using some IDE's features, subjects might be able to achieve better performance during some of the required maintenance activities.

To avoid social threats due to evaluation apprehension, students were not evaluated on their performance. During the experiment, we monitored the subjects to verify whether they were motivated and paid attention in performing the assigned task. We observed that students performed the required task with dedication and there was no abandonment. Moreover, students were not aware of the goal of our experiment nor of the dependent variables.

As for the objects, we performed the experiment on two industrial systems (AgilePlanner and eXVantage) because both have at least one instance of each test smell (see Table IV) and, belonging to the same category of systems, we reduce the possibility of the development style acting as a confounding factor.

V. Conclusion and Future Work

Test code smells have been presented in the literature as a possible threat to the maintainability of production code and test suites. However, until now no empirical evidence has demonstrated that test code smells occur quite frequently in software systems and that they negatively impact the maintainability of software systems.

This paper filled this gap by performing two empirical studies. In the first study, we analyzed the distribution of test smells in 18 software systems (two industrial and 16 open

source). The results demonstrated that from a total of 637 JUnit classes analyzed, only 112 (18%) were not affected by any test smell, while the remaining 525 (82%) was affected by at least one test smell with a peak of six test smells founded in six (1%) of the JUnit classes. Thus, our first case study highlighted the high diffusion of the test smells in software systems.

In our second study, we asked 20 Master students to perform maintenance activities on test suites of two software systems with and without test smells. The results showed that the presence of test smells has a strong negative impact on maintainability.

As a first direction for future work, we plan to corroborate our results by replicating the second study with different subjects and systems. Moreover, we are working on methods and tools able to (i) detect candidate test smell instances and (ii) automatically refactor them.

ACKNOWLEDGMENTS

We are very grateful to the students who were involved in the controlled experiment as subjects.

REFERENCES

[1] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, 1988.

[2] N. Wilde, P. Mathews, and H. Ross, *Maintaining Object-Oriented Software*. Addison-Wesley, 1993.

[3] L. S. Rising and F. W. Calliss, "An experiment investigating the effect of information hiding on maintainability," in *Proceedings of the 12th Annual International Phoenix Conference on Computers and Communication*, 1993, pp. 510–516.

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] 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 *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*. Oldenburg, Germany: IEEE CS Press, 2011, pp. 181–190.

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

[7] A. Van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," Amsterdam, Tech. Rep., 2001.

[8] A. van Deursen and L. Moonen, "The video store revisited – thoughts on refactoring and testing," in *Proceedings of International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP)*, Alghero, Italy, 2002, pp. 71–76.

[9] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[10] A. Schneider, "JUnit best practices," *Java World*, vol. 12, 2000, http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html.

[11] A. Qusef, G. Bavota, R. Oliveto, A. D. Lucia, and D. Binkley, "Scotch: Test-to-code traceability using slicing and conceptual coupling," in *Proceedings of the 27th IEEE International Conference on Software Maintenance*, Williamsburg, VA, USA, 2011, pp. 63–72.

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

[13] 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*, vol. 33, no. 12, pp. 800–817, 2007.

[14] S. Reichhart, T. Gîrba, and S. Ducasse, "Rule-based assessment of test quality." *Journal of Object Technology*, vol. 6, no. 9, pp. 231–251, 2007.

[15] M. Breugelmans and B. Van Rompaey, "Testq: Exploring structural and maintenance characteristics of unit test suites," in *Proceedings of the 1st International Workshop on Advanced Software Development Tools and Techniques (WASDeTT)*, 2008.

[16] G. Bavota, A. Qusef, R. Oliveto, A. DeLucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," available at http://distat.unimol.it/reports/TestSmells/, Tech. Rep., 2011.

[17] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Lawrence Earlbaum Associates, 1988.

[18] F. Ricca, M. D. Penta, M. Torchiano, P. Tonella, and M. Ceccato, "How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments," *IEEE Transactions on Software Engineering*, vol. 36, pp. 96–118, 2010.

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

[20] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

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

[22] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000.

[23] R. M. Sirkin, *Statistics for the social sciences*. Sage Publications, 2005.

[24] E. Arisholm and D. Sjoberg, "Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 521–534, 2004.