Empir Software Eng (2015) 20:1052–1094
DOI 10.1007/s10664-014-9313-0

Are test smells really harmful? An empirical study

Gabriele Bavota · Abdallah Qusef · Rocco Oliveto · Andrea De Lucia · Dave Binkley

Published online: 31 May 2014

Abstract Bad code smells have been defined as indicators of potential problems in source code. Techniques to identify and mitigate bad code smells have been proposed and studied. Recently bad test code smells (test smells for short) have been put forward as a kind of bad code smell specific to tests such a unit tests. What has been missing is empirical investigation into the prevalence and impact of bad test code smells. Two studies aimed at providing this missing empirical data are presented. The first study finds that there is a high diffusion of test smells in both open source and industrial software systems with 86 % of JUnit tests exhibiting at least one test smell and six tests having six distinct test smells. The second study provides evidence that test smells have a strong negative impact on program comprehension and maintenance. Highlights from this second study include the finding that comprehension is 30 % better in the absence of test smells.

Keywords Test smells · Unit testing · Mining software repositories · Controlled experiments

Communicated by: Thomas Zimmermann

G. Bavota
University of Sannio, Benevento (BN), Italy
e-mail: gbavota@unisannio.it

A. Qusef
Princess Sumaya University for Technology, Amman, Jordan
e-mail: a.qusef@psut.edu.jo

R. Oliveto (✉)
University of Molise, Pesche (IS), Italy
e-mail: rocco.oliveto@unimol.it

A. De Lucia
University of Salerno, Fisciano (SA), Italy
e-mail: adelucia@unisa.it

D. Binkley
Loyola University Maryland, Baltimore, USA
e-mail: binkley@cs.loyola.edu

1 Introduction

A *bad code smell* is an indication of a potential problem in the source code (Fowler 1999). While a bad code smell may not definitively identify an error, it does suggest a potential trouble spot, that is, a place where there is an increased risk of a future failure. The presence of bad code smells is symptomatic of developers failing to follow good design principles. This notion was introduced by Fowler (1999) who presented an informal definition of 22 bad code smells. He also described a set of characteristics that can be used to uncover the presence of bad code smells in a program. Once identified, one approach to removing bad code smells is to apply refactorings from a catalogue of operations (Fowler 1999).

More recently, Van Deursen et al. (2001) argued that bad code smells were not limited to production source code, but they could also plague test code such as unit test suites. This led to a distinct set of *bad test code smells* or simply *test smells*. Similar to bad code smells, test smells are conjectured to decrease the quality of systems and *ad-hoc* refactoring operations can be applied to remove them (Van Deursen et al. 2001).

Recent empirical work has shown how bad code smells accompany code that has an increased likelihood of needing to be changed to fix a fault (Khomh et al. 2012). These problems are exacerbated when the source code contains combinations of different bad code smells (Abbes et al. 2011a). Despite several studies that consider test smell definitions, identification, and refactoring (van Deursen and Moonen 2002; Meszaros 2007; Van Deursen et al. 2001), no empirical work has fully investigated the extent to which test smells exist in software systems nor their impact on programmer comprehension.

The two empirical investigations presented in this paper fill this gap. They provide empirical evidence of the prevalence of test smells and their negative impact on programmer comprehension. The two empirical studies investigate the following three research questions:

- **RQ**$_1$: What is the diffusion of test smells in software systems?
- **RQ**$_2$: Is the diffusion of test smells dependent on systems characteristics?
- **RQ**$_3$: What is the impact of test smells on program comprehension during maintenance activities?

The first empirical study is an exploratory study of 27 software systems (two industrial and 25 open source) aimed at investigating **RQ**$_1$ and **RQ**$_2$ by considering questions such as *"how are test smells spread in software systems?"* and *"which test smells are the most frequent?"*. The second study is a controlled experiment involving four groups of participants covering four different experience levels. It is aimed at investigating **RQ**$_3$. In this study we asked participants to perform different program comprehension tasks on test suites with and without test smells and we measured the participants' performance using both correctness and the time spent to perform a task. Both studies build upon a pilot study we conducted (Bavota et al. 2012), which suggested the need for more in depth studies to further investigate the three research questions described above. In particular, this paper extends our previous work (Bavota et al. 2012) as follows:

- we adopt a wider set of object systems (now 27 up from 18) to address the first two research questions and further generalize our findings. Also, we investigate the influence of new systems'l characteristics on the diffusion of test smells.
- we performed three replications of the previous user study to address the last research question, **RQ**$_3$. The original investigation involved 20 Master students. The replications involve 13 Bachelors students (2^{nd} year), 16 Fresher students (1^{st} year), and 12

industrial developers. These replications not only allow us to generalize our findings, but also permit the analysis of the role of participant experience on the impact of test smells during code comprehension tasks.

The rest of the paper is organized as follows. Section 2 provides background information on test smells and discusses the related literature. Section 3 presents the results of the study into the diffusion of test smells, while Section 4 presents the results of the comprehension experiment. Finally, Section 5 concludes the paper highlighting directions for future work.

2 Background and Related Work

In this section we provide background information about code and test smells and discuss the related literature. Background information on code and test smells is needed to fully understand why they are different and what are the possible problems they can cause. Concerning the related literature, we focus on work dealing with (i) the study and automatic identification of code and test smells, and (ii) empirical studies focused on the test code.

2.1 Code and Test Smells

Code smells have been defined by Fowler (1999) as symptoms of poor design and implementation choices. In some cases, such symptoms may originate by activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making sub-optimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns. In his refactoring book (Fowler 1999), Fowler defined 22 bad smells together with refactoring operations aimed at removing them. For instance, a *Feature Envy* bad smell arises when a method *seems to be more interested in another class (know as envied class) than in the one it is member of* (1999). In other words, the methods suffering of this smell exhibits a higher number of dependencies towards the envied class than towards the class containing it. The solution in this case is a *move method* refactoring, moving the method affected by feature envy from its class to the envied class.

Van Deursen et al. (Van Deursen et al. 2001) recently highlighted how poorly designed test code is also quite common in practice. Thus, they described symptoms that could indicate potential design problems in test code, i.e., *test smells*. These smells are different from those defined by Fowler for the production code, and are based on how the test cases are organized, implemented, and interact with each other. Because of this distinction the corresponding refactoring operations for test code and production code differ. Table 1 shows the test smells defined by Van Deursen et al. (2001), reporting for each of them a brief description, the possible side effects it might cause, and the refactoring needed to remove it.

The *Mystery Guest* smell happens when a test makes use of external resources, such as a file containing test data. This smells may create problems for program comprehension due to the test using unknown values stored in the external resource. The *Mystery Guest* can be removed by performing an *Inline Resource* refactoring, incorporating the data contained in the external resource inside the test method using them. However, it is worth noting that loading data from external resources is in some cases unavoidable, and can even be considered as a good practice, for example when adopting a Data-Driven Testing (DDT) approach. In such a case, it is not rare to have test methods needing to read hundreds of lines of data from an external making the application of *Inline Resource* inconvenient (due to the necessity of copying these hundreds lines inside the test code). This reflects exactly

Table 1 Refactoring test smells (Van Deursen et al. 2001)

Name	Description	Possible Effects	Refactoring Technique
Mystery Guest	A test uses external resources (e.g., file containing test data)	Difficulties in test comprehension because of unknown values	Inline Resource Setup External Resource
Resource Optimism	A test makes assumptions about the state/existence of external resources	Non-deterministic result depending on the state of the resources	Setup External Resource
Test Run War	A test allocates resources also used by others (e.g., tmp files)	Failures occur when several people run tests simultaneously	Make Resource Unique
General Fixture	A test case fixture is too general and the test methods only access a part of it	Difficulties in test comprehension Extract Method	Inline Method and Extract Class
Eager Test	A test method checks several methods of the tested object	Difficulties in test comprehension and maintenance	Extract Method
Lazy Test	Several test methods check a method of the tested class using the same fixture	Difficulties maintaining consistency during test maintenance	Inline Method
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	Add Assertion Explanation
Indirect Testing	A test interacts with the object under test indirectly via another object	Difficulties in test maintenance and debugging	Extract Method Move Method
For Testers Only	A production class contains methods used only by test methods	Difficulties during production code maintenance and comprehension	Extract Subclass
Sensitive Equality	The *toString* method is used in assert statements	Failures may occur if the *toString* method is changed	Introduce Equality Method
Test Code Duplication	Code clones contained inside the unit tests	Code clones have bad effects on maintainability	Extract Method

what a test smell is about: it is a symptom in the code that may (or may not) indicate a design problem. This is why, analyzing the impact of test smells on code comprehension and maintenance is very important.

Test smells very similar to *Mystery Guest* are *Resource Optimism* and *Test Run War*. The former makes assumptions about the state/existence of external resources thus possibly exhibiting a non-deterministic behavior depending on when/where it is run. The latter arises when a test allocates resources that are also used by other tests (e.g., temporary files). In both cases, a good solution is to apply a *Setup External Resource* refactoring, making sure

that a test using external resources explicitly creates or allocates them before testing and releases them when done (Van Deursen et al. 2001). Also, to remove a *Test Run War* smell a *Make Resource Unique* refactoring can be applied, making sure to use unique identifiers for all external resources allocated by the different tests (Van Deursen et al. 2001). As in the case of the *Mystery Guest*, it is worth noting that sometimes assumptions (as in the case of *Resource Optimism*) are needed during testing, and thus these smells can not always be considered as a poor programming practice.

A *General Fixture* smell happens when a test case fixture (setup) is too general and the test methods only access a part of it. The most common cause of this problem is the design of a single test fixture to support many tests having different fixture requirements. This smell could be an obstacle for program comprehension and can be removed using standard refactoring operations (i.e., *Extract Method*, *Inline Method*, and *Extract Class*). For example, if two different groups of test methods require different fixtures, the setup method can be split into two different methods (i.e., using *Extract Method*) or the test class hosting the test methods can be split into two separated classes (i.e., *Extract Class*).

An *Eager Test* is a test method that attempts to test several behaviors of the tested object. A failure in such a test method makes it harder for the developer to understand what went wrong during testing. The removal of this smell can be accomplished by splitting the method affected by *Eager Test* into smaller methods (i.e., *Extract Method*), each one testing a specific behavior of the tested object. Note that, as previously observed, the presence of this test smell (as of any other one) does not always imply poor programming choices. Indeed, sometimes testing different behaviors of the tested object together is simply a requirement.

A *Lazy Test* smell is somewhat the opposite of an *Eager Test*. Here we have several test methods exercising the same method of the tested object using the same test fixture. Also, these methods should be generally executed together. This smell can be removed with the application of an *Inline Method* refactoring to group all the involved methods inside a single method.

The *Assertion Roulette* smell describes a test method where many different assertions are made about the state of a behavior without providing explicit messages in case any of them fails. Thus, it is difficult to attribute a failure of the test method to a specific assertion. This smell can be removed by the *Add Assertion Explanation* refactoring (i.e., adding a specific failure message to each assertion).

An *Indirect Testing* smell arises when a test method is interacting with the object under test indirectly via another object thereby making the test more difficult to understand. As explained by Van Deursen et al. (2001) not all people consider *Indirect Testing* as a smell. Indeed, some of them consider it a way to guard tests against changes in the tested object, maybe more prone to change than the object used by the test as broker.

For Testers Only represents a production class containing methods that are only used by test methods. These methods are often not needed or just used to set up the test fixture. This additional code has two main drawbacks: (i) it represents additional code to be maintained, and (ii) it can confuse developers performing program comprehension on the class affected by the smells, since the goal of the smelly methods could not be immediately clear. The solution here is generally represented by an *Extract Subclass* refactoring isolating the *For Testers Only* code.

The *Sensitive Equality* smells occurs when using the *toString* method in assert statements. Whenever the *toString* implementation changes, tests will start failing. The solution is to replace *toString* equality by real equality checks using *Introduce Equality Method* (Van Deursen et al. 2001).

Finally, code clones in unit tests can be as problematic just as in the production code and are captured by the *Test Code Duplication* smell. In such a case the solutions are those generally adopted for clones present in the production code (e.g., *Extract Method* refactoring).

The smells presented above have been further described by Meszaros (2007) that explained in detail (i) the reasons why test smells appear and (ii) their potential side effects. However, no empirical evidence of such side effects is presented in his paper.

Neukirchen and Bisanzre define a different catalog of test smells targeted at the Testing and Test Control Notation (TTCN-3) test specifications (Neukirchen and Bisanz 2007). The set of test smells considered in this paper is different, as it is the one defined by Van Deursen et al. (2001).

While test smells are suspected of having a negative impact on maintainability, our preliminary study (Bavota et al. 2012) represents to date the only empirical evidence that (i) test smells are widely spread in software systems and (ii) they have a negative impact on software comprehension and maintenance. In this paper, we aim to more deeply investigate these aspects in order to confirm/refute our previous findings.

2.2 Automatic Detection of Code and Test Smells

Several approaches have been defined in the literature to identify code and test smells. Simon et al. (2001) provide a metric-based visualization tool able to discover design defects representing refactoring opportunities.

Marinescu (2004) proposes a mechanism called "detection strategies" for formulating metric-based rules that capture deviations from good design principles. The detection strategies are formulated in different steps. First, the *symptoms* characterizing a particular smell are defined. Second, a proper *set of metrics* measuring these symptoms is identified. Having this information, the next step is to define thresholds to classify the class as affected (or not) by the defined symptoms. Finally, AND/OR operators are used to correlate the symptoms, leading to the final rule for detecting the smells. Note that this approach could be used to detect both code and test smells.

Khomh et al. (2009) propose an approach based on Bayesian belief networks to specify and detect smells in programs. The main novelty of their approach is represented by the fact that it provides a likelihood that a code component is affected by a smell, instead of a boolean value as done with previous techniques.

Tsantalis et al. (2009) presents JDeodorant, a tool for detecting *Feature Envy* smells with the aim of suggesting move-method refactoring opportunities. In particular, for each method of the system, their approach forms a set of candidate target classes where a method should be moved. This set is obtained by examining the entities (i.e., attributes and methods) that a method accesses from the other classes. The current version of JDeodorant[1] also support the refactoring of other three smells (i.e., *State Checking*, *Long Method*, and *God Class*).

Moha et al. (2010) introduce DECOR, a method for specifying and detecting code and design smells. DECOR uses a Domain-Specific Language (DSL) for specifying smells using high-level abstractions. Four design smells are identified by DECOR, namely *Blob*, *Swiss Army Knife*, *Functional Decomposition*, and *Spaghetti Code*. Also this approach could be easily extended to also support the identification of test smells.

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

Ratiu et al. (2004) describe an approach for detecting smells based on evolutionary information of problematic code components (as detected by code analysis) over their life-time. The aim is to measure persistence of the problem and related maintenance effort spent on the suspected components. In a similar vain, the approach by Palomba et al. (2013) is built on top of historical information extraction and aims at identifying five different code smells.

Note that all techniques described above explicitly focus on code smells, despite some of them can be easily adapted to detect test smells (see e.g., Marinescu (2004) and Moha et al. (2010)). Few authors have targeted test smells. Van Rompaey et al. (2007) propose a heuristic metric-based approach to identify the *General Fixture* and *Eager Test* bad smells.

Reichhart et al. (2007) propose *TestLint*, a rule-based tool to detect static and dynamic test smells in Smalltalk SUnit code. Automated detection of smell instances has been built into the metrics and refactoring tool called *TRex* (Baker et al. 2006).

Greiler et al. (2013) present a test analysis technique, implemented in the *TestHound* tool, which provides reports on test smells together with refactoring recommendations aimed at removing them from the code. Their technique focuses on different types of smells that can arises in the test fixture.

Finally, Breugelmans and Van Rompaey (2008) introduce a reverse engineering tool called *TestQ* able to detect test smells using static analysis. Note that the aim of their work is mainly the presentation of the tool, without insight on test smells distribution or impact on maintenance. Indeed, the authors apply their tool on one system just to show the features it provides. Breugelmans and Van Rompaey highlights in their work the need for empirical studies to further characterize test smells, their interactions, and their impact on maintainability (Breugelmans and Van Rompaey 2008).

2.3 Empirical Studies on Code and Test Smells

Chatzigeorgiou and Manakos (2010) observe how the number of bad smells in software systems increases over time. In particular, their study focus attention on three bad smells, *Long Method*, *Feature Envy*, and *State Checking* on different versions of two open source systems. Results show an increasing trend for all the code bad smells analyzed. The authors also observed as the diffusion of the three investigated smells is quite different: *Long Method* is by far more diffused than *Feature Envy* and *State Checking*. Also in our work we found specific test smells (e.g., the *Assertion Roulette*) to be much more diffuse than other test smells. Overall, the test smell diffusion we observe in this paper is higher than the code smell diffusion observed by Chatzigeorgiou and Manakos.

Also Peters and Zaidman (2012) study the evolution of code smells over time. The authors analyze the lifespans of instances of five code bad smell (i.e., *Blob*, *Feature Envy*, *Data Class*, *Message Chain*, *Long Parameter List*) in seven open source systems. Their results show that (i) several bad smells present in the analyzed systems were removed as a side effect of other maintenance activities or when a new functionality was implemented and (ii) even if developers are aware of the presence of code bad smells in their code, they are not always interested in activities aimed at removing them.

These findings are also confirmed by Arcoverde et al. (2011), who report the results of a survey aimed at understanding the longevity of code smells and the reasons why developers do not care to remove them from the code. The authors show that often code smells remain in source code for a long time and the main reason to postpone their removal through refactoring activities is to avoid API-breaking modifications (Arcoverde et al. 2011).

Abbes et al. (2011b) investigate the impact of two types of code bad smells - *Blob* and *Spaghetti Code* - on program comprehension. The results show that the presence of a code smell in the source code does not strongly decrease the developer's performance, while a combination of bad smells does result in a significant decrease in performance.

The interaction between code smells has been extensively studied by Yamashita and Moonen (2013). In particular, the authors show that the maintenance problems are not always deriving from the presence of a single code smell in a class, but they are strongly related to the co-occurrence of bad smells in the same file.

Deligiannis et al. (2003) conduct an experiment using students as subjects that showed that classes affected by the *God Class* smell are more difficult to maintain with respect to classes not affected by this smell.

Classes affected by code bad smells have also been studied in order to investigate their change and defect proneness. All the empirical studies conducted in the literature show that code smells make classes more change and defect prone with respect to classes not affected by such design problems (see, for example, Khomh et al. (2009)).

Li and Shatnawi (Li and Shatnawi 2007) found that the code smells *Shotgun Surgery*, *God Class*, and *God Methods* were associated positively with the number of defects in three releases of Eclipse (3.0, 2.1, 2.0).

While there is such an abundance of studies dealing with code smells, to the best of our knowledge only two papers focus their attention on test smells. The first is the work by Greiler et al. (2013). Here the authors investigate the evolution of fixture-related test smells with the final aim of providing recommendations for setup strategies. The second, is our previous pilot study (Bavota et al. 2012), on which this paper is built. While in the code smell literature (Abbes et al. 2011b; Yamashita and Moonen 2013) the performed empirical studies highlighted a negative impact of code smells on developers performances mainly when smells co-occur in the same code component, the results of our previous work (Bavota et al. 2012) and of this paper show a negative impact of (most) test smells even when occurring in isolation. From this point of view our results are more inline with what observed by Deligiannis et al. (2003) in their study on *God Classes*.

2.4 Empirical Studies on the Evolution of Test Code

Zaidman et al. (2011) study the co-evolution of software and test code by mining the history of three software systems (two open source and one industrial). Their main findings suggest that (i) test code and production code do not always co-evolve during the development history, and (ii) there is not an increase in testing activities right before major releases, even if there are peaks of intense testing in the projects history. Note that this result is somewhat contrasting with what observed by Hindle et al. (2007) who report an increased test writing activity immediately before releases of the MySQL project.

Pinto et al. (2012) investigated the evolution of test code to better understand how automated test-repair techniques can assist developers during tests maintenance. Their findings show that the need for repairing test cases as a consequence of production code evolution represents only one of the possible reasons for test case evolution, whereas most changes are related to refactoring, deletion, and addition of test cases. Also, the results in Pinto et al. (2012) suggest that the changes applied by developers during test maintenance are complex and hard to automate. While the work of Pinto et al. shown that test smells are not part of the analysis, the refactorings identified during the projects history might be a sign of attempts made by developers to improve the test quality.

3 The Distribution of Test Smells in Software Projects

This section reports the design and results of our empirical study to investigate the distribution of the test smells defined by Van Deursen et al. (2001) in open source and industrial systems. The replication package, containing the raw data of the study is available online (Bavota et al. 2013). Note that, since 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*. Indeed, the distinction between these three smells is subtle, and they can often be removed by undertaking similar refactoring actions. Thus, nine of the eleven smells are considered in our study.

3.1 Study Design

In this section we define the research questions of the study, explaining the process followed to assess them.

3.1.1 Definition and Context

The *goal* of our study is to understand for each of the nine analyzed smells (i) to what extent it is spread in real software systems and (ii) if its diffusion is linked to system characteristics (e.g., open source/industrial projects, system size, etc.). To address these goals, we formulated the following two research questions:

- **RQ**$_1$: What is the diffusion of test smells in software systems?
- **RQ**$_2$: Is the diffusion of test smells dependent on systems characteristics?

To answer these research questions we analyzed the diffusion of test smells in the test suites of 25 open source and two industrial systems (AgilePlanner and eXVantage). All object systems are written in Java and exploit the JUnit framework in their test suite. The total number of JUnit classes present in the object systems is 987.

Table 2 lists the 27 systems providing size characteristics, namely Kilo LOC (KLOC) of the production code, number of classes, number of JUnit tests under study, and KLOC for the JUnit tests, as well as historical information, namely the age of the project in days, and the average age of the JUnit tests contained in it. The age of a JUnit test is computed as the period of time going from the date in which the JUnit test has been added to the system repository until the date of the last commit while the age of a system is computed as the period of time going from the first commit until the last commit. In addition, Table 2 also shows the size of the development team working on the system, assuming this count is reflected in the number of committers mined from the repository log. Unfortunately, we were not able to extract historical and team size information for three software projects: eXVantage, Groboutils, and Ubermq.

3.1.2 Data Collection

Having 987 JUnit classes to analyze makes manual detection of instances of the nine test smells prohibitively expensive. In fact, checking the presence of nine smells in 987 JUnit classes would mean performing 8,883 code analysis tasks. For this reason, we developed a simple tool to detect the nine analyzed test smells. The tool takes as input a folder containing the system to analyze and outputs a list of candidate JUnit classes (production code classes for *For Testers Only*) potentially exhibiting a test smell. To ensure high recall, our detection

Table 2 Object systems used in our study

System	KLOC	# Classes	#JUnit Classes	JUnit KLOC	System Age (Days)	JUnit Age (Days)	Team Size
AgilePlanner 2.5	24	299	32	4	1,312	945	13
Apache Ant 1.8.1	108	851	75	8	5,113	4,039	46
ArgoUML 0.30.1	124	1,430	75	7	5,823	3,370	51
Barcode 2.1.0	14	167	35	3	222	101	1
Cobertura 1.9.4.1	66	121	21	1	3,224	2,724	19
Colossus 0.13.0	58	304	9	5	5,851	4,308	19
DependencyFinder 1.2.1.b3	29	498	120	19	4,337	2,937	3
eXVantage 20090507173755	28	348	17	4	–	–	–
FindBugs 2.0.0	92	1,023	27	2	3,944	2,259	25
Groboutils 5	104	393	174	19	–	–	–
Hsqldb 2.2.8	131	443	12	3	4,104	2,137	11
Jabref 2.7.2	62	544	50	5	2,948	2,844	34
Java2html 5.0	9	73	16	1	544	530	1
JMulTi 4.24	44	192	3	1	962	900	1
JwebUnit 3.0	8	36	30	3	2,394	1,466	2
Jvalidations 0.7	2	22	5	1	612	593	2
Morph 1.1.1	19	262	20	2	1,638	1,221	3
Optal 1.4	28	356	11	1	2,493	1,972	6
pmd 5.0.0	94	778	86	5	4,213	2,982	54
QuickFixj 1.5.2	19	204	49	6	3,234	2,785	10
jforum2 190d28b	65	368	13	1	998	852	5
Regain 1.7.11	22	223	4	1	52	31	2
Take 1.5.1	30	377	22	1	906	678	6
TripleA 1.3.2.2	97	640	54	9	4,376	2,602	22
Ubermq 2.7	31	246	6	1	–	–	–
xBaseJ 20090902	8	36	14	2	2,166	1,225	2
XuiPro 3.2	157	790	7	1	626	295	6
All Systems	1,473	11,024	987	116	–	–	–

tool uses very simple rules that overestimate the presence of test smells in the code. This is done at the expense of precision, but it was necessary because our goal was to avoid missing any test-smell instances. Table 3 presents the rules applied by our tool to detect each of the nine analyzed test smells. For example, we retrieve as affected by *General Fixture* all JUnit classes having at least one test method not using the entire test fixture defined in the `setUp()` method. All the defined rules clearly overestimate the presence of test smells in the code. Note that, to extract the information needed to verify the rules reported in Table 3, our tool performs static code analysis by relying on the Eclipse JDT parser. Our tool is publicly available in our online appendix (Bavota et al. 2013).

Once generated, the list of candidate affected classes, was manually validated. The final list included 1,477 candidate test smells instances grouped in 883 JUnit tests. Note that while the number of test smells instances to evaluate was still quite high after the tool selection, it is much easier to check inside a JUnit class the presence (or absence) of a

Table 3 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 invoking a *toString* method	JUnit classes having at least one assert statement
Test Code Duplication	JUnit classes identified as containing clones by the CCFinder clone detection tool (http://www.ccfinder.net)

specific smell indicated by our tool than it is to look for all instances of all nine smells. The validation was performed by three Ph.D. students[2] 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. In particular, the discussion involved 197 test smell instances (13 %) on which the Ph.D. students did not fully agree and lasted for almost 13 hours split in three working days. Of the 197 analyzed instances, 64 were *Indirect Testing*, 48 *General Fixture*, 39 *Eager Test*, 18 *Lazy Test*, 15 *Test Code Duplication*, and 3 *Mystery Guest*.

Note that we did not use existing detection tools because their detection rules are generally too restrictive since they aim at reaching a good compromise between recall and precision, and thus may miss test smell instances. As an example, to detect the *General Fixture* test smell, the *TestQ* detection tool (Breugelmans and Van Rompaey 2008) uses heuristics based on metrics, while we simply retrieve as candidates those JUnit classes having at least one method not using the entire test fixture defined in the *setUp()* method. Moreover, detecting 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 detection of these three test smells, we exploit test-to-code traceability information previously derived by the same three Ph.D. students. In particular, these links have been manually extracted relying on the support of the tools provided by the Eclipse IDE. The traceability information exploited by our tool is also available in our online appendix (Bavota et al. 2013).

[2]None of Ph.D. students co-authored the paper.

3.1.3 Data Analysis

Concerning $\mathbf{RQ}_1$, besides analyzing in each system the percentage of classes affected by each of the studied test smells, we also evaluate 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 measure the percentage of times that its presence in a JUnit class co-occurs with each other test smell T_j ($i \neq j$). Specifically, for each pair of test smells T_i, T_j, we measure the percentage of co-occurrences of T_i and T_j as

$$co\text{-}occurrences_{T_{i,j}} = \frac{|T_i \wedge T_j|}{|T_i|}$$

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 $co\text{-}occurrences_{T_{i,j}}$ differs from $co\text{-}occurrences_{T_{j,i}}$ since the formula's denominator changes from $|T_i|$ to $|T_j|$.

For $\mathbf{RQ}_2$, we analyze possible correlations between the systems' characteristics and the test smells' presence. We consider the following characteristics:

1. Production code LOC;
2. Number of Classes;
3. Number of JUnit Classes;
4. JUnit test LOC;
5. System age (in days);
6. Average age of JUnit Classes (in days);
7. Development team size.

We compute, for each object system, the Pearson product-Moment Correlation Coefficient (PMCC) (Cohen 1988) 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. (1988) provided a set of guidelines for the interpretation of the correlation coefficient, ρ. 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.

Note that correlations involving historical system characteristics (i.e., system age and average age of JUnit classes) as well as those involving team size information are computed by considering only the subset of 24 system for which the data was available.

3.2 Analysis of the Results

In this section we provide answers to the research questions formulated in Section 3.1.1.

3.2.1 What is the Diffusion of Test Smells in Software Systems?

Table 4 shows the distribution of the test smells in the analyzed object systems. Note that the *For Testers Only* diffusion is not shown in the table since the instances of this smell appear in the production code and not in the test suite. In fact, *For Testers Only* represents a method (or an entire class) in the production code that is used only by test methods. We

Table 4 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	91 %	3 %	16 %	19 %	34 %	16 %	81 %	9 %	3 %
Apache Ant	75	87 %	27 %	29 %	16 %	51 %	0 %	56 %	13 %	4 %
ArgoUML	75	97 %	16 %	0 %	23 %	24 %	4 %	75 %	19 %	8 %
Barcode	35	83 %	11 %	3 %	0 %	31 %	3 %	71 %	6 %	6 %
Cobertura	21	90 %	48 %	33 %	38 %	57 %	0 %	76 %	0 %	0 %
Colossus	9	89 %	22 %	0 %	22 %	56 %	11 %	78 %	44 %	0 %
Dependency Finder	120	82 %	33 %	4 %	34 %	25 %	0 %	38 %	8 %	7 %
eXVantage	17	100 %	35 %	6 %	12 %	41 %	6 %	100 %	71 %	6 %
FindBugs	27	89 %	15 %	4 %	22 %	11 %	7 %	70 %	7 %	15 %
Groboutils	174	98 %	96 %	8 %	5 %	33 %	0 %	13 %	0 %	5 %
Hsqldb	12	92 %	67 %	17 %	17 %	8 %	0 %	58 %	0 %	0 %
Jabref	50	56 %	14 %	12 %	10 %	16 %	4 %	46 %	28 %	2 %
Java2html	16	88 %	0 %	6 %	6 %	63 %	0 %	81 %	0 %	0 %
Jmulti	3	100 %	33 %	33 %	0 %	33 %	0 %	100 %	33 %	0 %
JwebUnit	30	60 %	20 %	7 %	10 %	7 %	0 %	50 %	0 %	0 %
Jvalidations	5	100 %	0 %	0 %	0 %	40 %	0 %	80 %	0 %	0 %
Morph	20	40 %	5 %	0 %	10 %	5 %	0 %	25 %	0 %	0 %
Optal	11	91 %	45 %	0 %	0 %	73 %	18 %	82 %	64 %	0 %
pmd	86	90 %	15 %	7 %	9 %	38 %	0 %	77 %	0 %	6 %
QuickFixj	49	86 %	27 %	6 %	12 %	45 %	0 %	69 %	16 %	16 %
Rafaelsteil	13	69 %	15 %	0 %	38 %	54 %	0 %	46 %	0 %	0 %
Regain	4	50 %	0 %	0 %	0 %	25 %	0 %	25 %	0 %	0 %
Take	22	95 %	18 %	5 %	23 %	14 %	0 %	91 %	0 %	0 %
Triplea	54	91 %	33 %	0 %	30 %	50 %	11 %	87 %	31 %	6 %
Ubermq	6	83 %	33 %	0 %	33 %	67 %	0 %	50 %	0 %	0 %
Xbasej	14	93 %	0 %	7 %	7 %	64 %	0 %	93 %	21 %	0 %
XuiPro	7	56 %	29 %	0 %	0 %	43 %	0 %	29 %	0 %	0 %
All Systems	987	86 %	35 %	8 %	16 %	34 %	2 %	55 %	11 %	5 %

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 4 highlights their presence in the analyzed systems. *Assertion Roulette* is present in all 27 systems and affects 55 % of the 987 analyzed JUnit classes (i.e., 543 tests). The high diffusion of *Assertion Roulette* was also noted in our previous work presenting a technique to establish traceability links between test and production code (Qusef et al. 2011). Thus, understanding if it represents an actual problem for software maintenance is very important. Also the *Eager Test* smell affects at least one JUnit class in each of the analyzed systems. In total 336 JUnit classes are affected by this smell (34 %). This means that 34 % of the JUnit classes contain at least one test method exercising more than one method of the tested object, violating one of the JUnit best practices.

Also the *Test Code Duplication* smell is quite diffused. It affects 23 systems and a total of 345 classes. The possible side effects of having clones in test code are exactly the same as those well known for the maintainability of the production code. Other diffused test smells are *General Fixture* (23 systems - 16 %) and *Indirect Testing* (14 systems - 11 %).

On the other hand, the three test smells *Mystery Guest* (8%), *Sensitive Equality* (5%), and *Lazy Test* (2 %), have a low diffusion in the 27 systems. This means that developers dont tend to (i) use external resources (e.g., files, databases) when writing JUnit tests (*Mystery Guest*), (ii) invoke *toString* methods in assert statements, and (iii) write more than one test method to exercise a method of the tested object (*Lazy Test*).

A very unexpected result in our study is that among the 987 analyzed JUnit classes, only 134 (14 %) are not affected by any test smell. The remaining 853 (86 %) are affected by at least one test smell. Among these, 370 (37 %) are affected by only one test smell, 284

Table 5 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
Cobertura	2	3	4	7	4	1	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
Groboutils	3	85	70	10	6	0	0
Hsqldb	1	5	3	3	0	0	0
Jabref	22	10	6	7	3	1	1
Java2html	2	3	11	0	0	0	0
JMulTi	0	2	0	0	0	1	0
JwebUnit	12	9	8	1	0	0	0
Jvalidations	0	4	1	0	0	0	0
Morph	12	7	1	0	0	0	0
Optal	1	0	2	5	3	0	0
pmd	9	36	30	9	2	0	0
QuickFixj	8	13	14	8	5	2	0
Rafaelsteil	4	0	7	2	0	0	0
Regain	2	2	0	0	0	0	0
Take	1	13	4	4	0	0	0
TripleA	6	11	10	15	7	6	0
Ubermq	1	3	0	0	2	0	0
xBaseJ	1	4	6	1	2	0	0
XuiPro	1	5	1	0	0	0	0
All Systems	134	370	284	116	60	17	6

(29 %) by two, 116 (12 %) by three, 60 (6 %) by four, 17 (2 %) by five, and six (1 %) by six. Table 5 reports the detailed data for each system.

An example of a test suite affected by six test smells is the JUnit class *SynchronousPersisterTest* from 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.

Table 6 shows the co-occurrences of test smells in the analyzed JUnit classes. Most of the test smells frequently co-occur with *Assertion Roulette*. Note that this result is quite expected given the very high diffusion of this smell (55 % of JUnit classes affected).

More interesting are the co-occurrences observed between *Lazy Test* and *Eager Test*. It is worth remembering that we have a *Lazy Test* when several test methods check a method of the tested class using the same fixture, while an *Eager Test* arises when a test method checks several methods of the tested object. What we observe from Table 6 is that when a *Lazy Test* 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 5 % of the time. To understand the reasons behind these results, 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 it. 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 5 % 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 5 % 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). This result also highlights how test smells could be caused by problems in the tested code and thus, might be exploited as an indicator of some of its quality attributes like, for instance, testability.

Answer to RQ_1. The diffusion of the test smells in the 27 analyzed software systems is generally quite high. In particular, 86 % of the 987 analyzed JUnit tests exhibit at least one test smell, while 49 % contain at least two test smells. Their prevalence highlights the need for empirical evaluation targeted at analyzing the test smells influence on the maintainability of the test suites. In our second empirical study (Section 4) we provide such evidence.

Table 6 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.		11 %	17 %	41 %	3 %	41 %	9 %	8 %
Mystery Guest	48 %		25 %	49 %	4 %	**56 %**	16 %	6 %
General Fixture	37 %	13 %		33 %	3 %	**68 %**	20 %	6 %
Eager Test	42 %	12 %	16 %		5 %	**63 %**	18 %	7 %
Lazy Test	39 %	13 %	22 %	**78 %**		**83 %**	**61 %**	22 %
Assertion Roulette	26 %	8 %	20 %	39 %	3 %		16 %	25 %
Indirect Testing	31 %	12 %	29 %	**57 %**	13 %	**82 %**		7 %
Sensitive Equality	**54 %**	10 %	18 %	48 %	10 %	**54 %**	14 %	

3.2.2 *Is the Diffusion of Test Smells Dependent on Systems Characteristics?*

Before doing so, we turn to **RQ**$_2$ where we first compare the diffusion of test smells in the two industrial systems and in the 25 open source systems involved in our study. The goal is to understand if strong differences in the distribution of smells between the two types of systems arise. Note that, due to the difficulty in finding industrial repositories, we have analyzed only two industrial systems. This clearly limits the external validity of our observations regarding industrial systems.

We excluded the *For Testers Only* smell from the analysis since we already know that it is present only in two systems, the industrial system AgilePlanner and the open source system Apache Ant. Thus, it is poorly diffused in both kinds of systems.

Figure 1 reports the distribution of test-smell instances in open source (black bars) and industrial (gray bars) systems. The depicted trends look very similar for the two categories of systems. In fact, test smells diffused in open source systems are also diffused in the industrial ones and *vice versa*. However, from the analysis of Fig. 1 some interesting observations can be made. Generally, 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

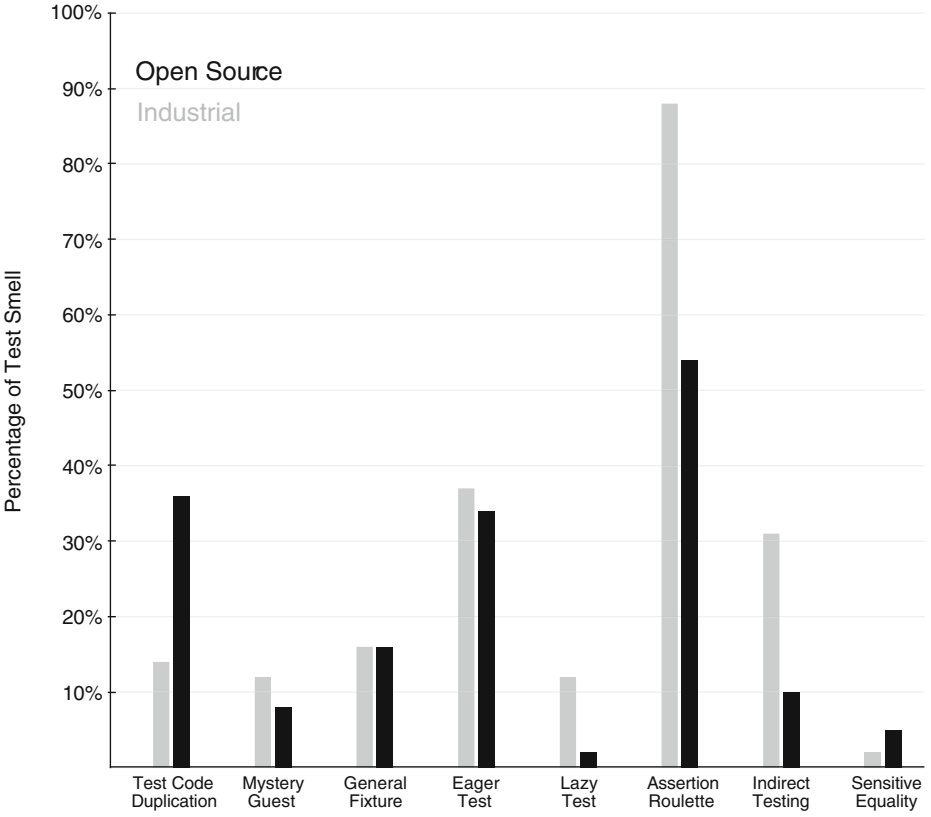

Fig. 1 Test smells distribution on industrial and open source systems

often found in an industrial context, which would make industrial programmers more prone to bad programming practices. To better investigate this result, we contacted developers of the two industrial systems object of our study to understand what are the quality assurance practices adopted in their teams. We got answers just from the eXVantage developers:

> *eXVantage has been produced using the fast Software Product Line Engineering (SPLE) process. We followed the process quite closely. All artifacts produced as part of the process underwent peer reviews using the active reviews process. We also used white box testing, black-box testing for the modules based on their interface specifications, and integration testing.*

Thus, our intuition is that the adopted quality assurance practices were mainly aimed at verifying the correctness of the implementation, rather than on the appropriate application of good Object-Oriented design principles.

Still in the context of Fig. 1, it is interesting to note that the test smell for which we observed a much lower diffusion in industrial systems is *Test Code Duplication*. This is likely due to a higher reuse of code in the open source community as compared to the industrial environments.

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 504 out of 939 (54 %) in the open source systems. The second most frequent is *Eager Test*. On the other hand, test smells like *Sensitive Equality* and *Lazy Test* have few instances in both industrial and open source systems.

Concerning the correlation between the distribution of test smells and the systems characteristics, Table 7 reports the results of the PMCC. Strong correlations (> 0.5) are reported in **bold** in Table 7. *General Fixture* test smell, which 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. This is also confirmed by the fact that this smell is more diffused in systems developed by larger teams and thus, likely more complex systems.

Table 7 Correlations between systems characteristics and test smell presence (PMCC). In **bold** the strong correlations

Test Smell	LOC	#Classes	#JUnit Classes	JUnit LOC	System Age	JUnit Age	Team Size
Mystery Guest	0.14	0.09	**0.62**	0.46	0.03	0.15	0.10
General Fixture	0.47	**0.55**	**0.82**	**0.70**	**0.53**	0.47	**0.51**
Eager Test	0.24	0.31	**0.78**	**0.66**	0.11	0.13	-0.02
Lazy Test	0.14	0.35	0.24	0.33	0.31	0.26	0.40
Assertion Roulette	0.22	0.44	**0.93**	**0.78**	0.06	0.04	-0.06
Indirect Testing	0.11	0.36	0.41	**0.62**	0.46	0.47	0.24
Sensitive Equality	0.32	0.48	**0.85**	**0.80**	**0.50**	**0.50**	**0.50**
Test Code Duplication	**0.55**	**0.57**	**0.77**	**0.73**	0.43	0.36	0.30

The *General Fixture* distribution also exhibits a strong positive correlation with the system age (0.53). Moreover, it is positively correlated with the average JUnit age (0.47). The higher presence of this smell in older code components could be due to the addition, during the evolution, of new test methods into the existing JUnit tests. The addition of such new methods could have led to the insertion in the `setUp()` method

As for the other test smells, they generally correlate with the size of the systems test suite (expressed in terms of number of JUnit Classes and LOC) while only the *Test Code Duplication* smell additionally correlates with the production code size. This is probably due to the fact that in larger systems it is more likely to find similar classes in production code that could require very similar test code to be tested. This could induce developers to introduce *Test Code Duplication* smells.

The *Sensitive Equality* test smell also exhibits a positive strong correlation with both the system age as well as the JUnit classes age (0.50 in both cases). Note also that this smell strongly correlates with the system/tests size. However, the higher use of the `toString()` method in assert statements in older and larger software systems is difficult to explain. We can only argue that older and larger systems are likely more complex and thus richer of structured classes (e.g., `JavaBeans`) encapsulating more objects into a single one. For such classes, it is quite the norm to have a `toString()` method implemented that returns a string containing all the different information stored in the object. The higher diffusion of the *Sensitive Equality* test smell in more complex systems is also evidenced by the fact it is more diffused in systems developed by larger teams (0.50 of correlation – see Table 7). Indeed, it is reasonable to think that the higher the team size the more complex the software system.

Answer to RQ_2 The diffusion trend of test smells in industrial and open source systems is very similar, even though it is slightly higher in industrial projects. Also, as expected, we observed that the larger the software system, the higher the likelihood that its JUnit classes are affected by test smells. Moreover, some smells, and in particular the *General Fixture* and the *Sensitive Equality*, also positively correlate with the age of the software systems and of the test classes.

3.3 Threats to Validity

This section discusses the validity threats that could affect the validity of our results. instances present in the analyzed software system. Indeed, if a class does not have a test method using more than one method of the tested class, it clearly cannot have a method testing more than one method of the tested class. Thus, while we do not have data showing the high recall of our tool (since this would require the manual construction of an oracle for all test smells on at least one system), we are quite confident of attaining high recall levels.

Also the manual validation of the candidate test-smell instances performed by the three Ph.D. students is a possible threat. 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.

Threats to *internal validity* concern any confounding factor that could influence our results. Such threats typically do not affect exploratory studies like the one in this section. The only case worthy of discussion is the analysis of co-occurrences of test smells in the same JUnit class. Although we found some strong co-occurrences between certain kinds of test smells, we cannot claim there is a cause-effect based on these results. Thus, we manually inspected some of these cases to support the statistical findings.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. The analyses performed in this study mainly have an observational nature, although we used, where appropriate, statistical procedures to support our claims.

Threats to *external validity* concern the generalization of our findings. While the number of analyzed open source systems (25) could be sufficient to infer generalizations of the results, more industrial systems are needed beyond the two analyzed in this paper to corroborate our results. Moreover, the characteristics of the object systems might have influenced our findings. For instance, most of the involved systems have a quite low number of JUnit classes as compared to production-code classes and this factor could have influenced the presence of test smells in them. However, it is worth noting that also in the paper by Zaidman et al. studying the co-evolution of test and production code (Zaidman et al. 2011) the authors do observed a quite low number of JUnit classes as compared to the production classes.

4 On the Impact 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.

4.1 Study Design

The *goal* of the study is to analyze the effect of test smells on software maintenance. Thus, the following research question is investigated

- **RQ$_3$**: What is the impact of test smells on program comprehension during maintenance activities?

4.1.1 Context Selection

To answer **RQ$_3$** we performed a controlled experiment and three replications involving a total of 61 participants.

Participants The experiments involved four categories of participants having different experience:

- *Fresher students*: 1st year B.Sc. students from the University of Molise (Italy) that in their academic career have attended the Object-Oriented programming course (taught

in Java). These students did not have knowledge of systematic testing techniques nor of the JUnit framework when the experiment was performed.

- *Bachelors students*: 2^{nd} year B.Sc. students from the University of Salerno (Italy) who have taken Object-Oriented programming and Software Engineering courses. As part of their Software Engineering course these students developed a software system for which they wrote JUnit test suites.
- *Masters students*: 2^{nd} year M.Sc. students from the University of Salerno who have attended advanced courses of Programming and Software Engineering in the past. Besides their experience as Bachelors students, as part of their Advanced Software Engineering course, these students have also performed maintenance activities (i.e., from impact analysis until regression testing) on previously developed software systems.
- *Industrial developers*: professional developers working in industry. Note that the involved developers were not all from the same software house.

Twenty Masters students were involved in the original experiment (Exp_1 in the following), while 13 Bachelors students were involved in the first replication (Exp_2 in the following) and 16 Fresher students were involved in second replication (Exp_3 in the following). Finally, the third replication (Exp_4 in the following) was performed with 12 Industrial developers. Participants represented the only substantial difference among the experiment and the replications.

With regard to the ethics of the experiment, the experiments performed with students were part of a series of extra-laboratory exercises conducted within the Software Engineering and Advanced Software Engineering courses and students were not evaluated on their performance. Moreover, these laboratory exercises were not part of the courses and students were free to participate or not. As for the industrial participants, we invited a total of 22 developers to take part in our study and twelve of them accepted. Their participation was fully voluntary, without any kind of reward.

Experimental Data All the experiments were performed using two systems, AgilePlanner and eXVantage. We chose these systems since (i) both have at least one instance of each test smell (see Table 4) and (ii) they are both industrial systems. The latter reason aims at considering in our study two systems developed in the "same context", thus avoiding a confounding factor.

For each of the eight test smells we randomly selected a JUnit class presenting that smell from from each object system. Table 8 reports the selected JUnit classes with the methods affected by the test smells. In addition, Table 9 reports the severity of the selected test smell instances on both systems and the indicator used to evaluate such severity. For instance, *Eager Test* is considered more severe when there is a larger number of methods that it exercises in the tested object. As shown in Table 8, on five out of the eight considered smells the severity is the same on both systems. On the remaining three smells (i.e., *Eager Test*, *General Fixture*, and *Test Code Duplication*) the instances present in eXVantage are more severe with respect to those present in AgilePlanner.

To obtain test smell free versions of the selected JUnit classes, we manually refactored them following the guidelines provided by Van Deursen et al. (2001). Note that refactoring operations were applied just to remove the test smell, without performing any unnecessary changes aimed at improving other aspects of the code.

Table 8 JUnit classes and test methods involved in our study

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

In particular, for each of the eight involved smells the following refactoring actions were performed[3]:

- *Mystery Guest*: the external resources needed by the test method were moved from the external file hosting them into a string declared inside the test method. This refactoring is known as *Inline Resource* (Van Deursen et al. 2001).
- *General Fixture*: we leaved in the JUnit's `setUp` method only the part of fixture used by all test methods in the affected JUnit class. Then, the part of fixture removed (through an *Extract Method* refactoring (Fowler 1999)) from the `setUp` method was placed (through an *Inline Method* refactoring (Fowler 1999)) in the method(s) using it.
- *Eager Test*: the method affected by this smell was split (through an *Extract Method* refactoring (Fowler 1999)) into several methods, each one testing a specific method of the tested object.
- *Lazy Test*: all methods in the JUnit class testing the same method of the tested object with the same fixture were grouped together in a single method through *Inline Method* refactoring (Fowler 1999).
- *Assertion Roulette*: we added the missed "failure explanation parameter to the assert statements of the method affected by this smell. This was done through an *Add Assertion Explanation* refactoring (Van Deursen et al. 2001).
- *Sensitive Equality*: we replaced the `toString` method used as equality checks within assert statements with real equality checks by using the *Introduce Equality Method*

[3]Note that the same refactoring was applied on test smell instances from both systems.

Table 9 Severity of the smells considered in our study

Test Smell	System	Indicator	Severity
Mystery	AgilePlanner	Number of exploited	1
Guest	eXVantage	external files	1
General	AgilePlanner	Number of unused objects	2
Fixture	eXVantage	in the test fixture	7
Eager	AgilePlanner	Number of methods tested	3
Test	eXVantage	in the tested object	6
Lazy	AgilePlanner	Number of test methods testing	2
Test	eXVantage	the same method of the tested object	2
Assertion	AgilePlanner	Number of asserts	2
Roulette	eXVantage	without the textual explanation	2
Indirect	AgilePlanner	Number of objects used to	1
Testing	eXVantage	indirectly testing the tested object	1
Sensitive	AgilePlanner	Number of asserts using	1
Equality	eXVantage	a toString() method	1
Test Code	AgilePlanner	Number of copies of the	4
Duplication	eXVantage	duplicated code	6

refactoring (Van Deursen et al. 2001). For instance, consider the following snippet of code:

```
String hashCode = "XXXXXXX";
BacklogModel bl = new BacklogModel(hashCode);
assertEquals("Hash code"; bl.toString(); hashCode);
```

has been replaced with the following:

```
String hashCode = "XXXXXXX";
BacklogModel bl = new BacklogModel(hashCode);
assertEquals("Hash code"; bl.getHashCode(); hashCode);
```

- *Test Code Duplication*: duplicated code present in the test methods of the affected class (i.e., the same code was duplicated across different test methods) was extracted through *Extract Method* refactoring (Fowler 1999). All methods previously containing the duplicated code invoked the newly extracted method after the refactoring.

After performing the refactorings above we checked if the smell was removed by re-running our conservative detection tool described in Section 3.1.2. As a result of the performed refactorings, all smells were removed from the affected classes.

4.1.2 Study Procedure

Each experiment was organized in two laboratory sessions lasting 60 minutes each and were separated by a 15 minutes break[4]. Each participant worked on JUnit classes of a system with test smells in one laboratory session and on JUnit classes of the other system without test

[4]Participants were monitored during the break to ensure that they did not exchange information.

Table 10 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)

smells in the other laboratory session. The organization of each group of participants in each lab session (*Lab1* and *Lab2*) followed the design shown in Table 10. The rows represent the four experimental groups and the columns show the presence or absence of test smells in the analyzed JUnit classes. For example, participants in group A worked on JUnit classes from AgilePlanner without test smells in *Lab1* and on JUnit classes from eXVantage with test smells in *Lab2*.

Note that, before the experiments, we showed to participants some examples of JUnit test suites. This was necessary for Fresher students who had not worked yet with this framework. To avoid bias (i) the training was performed on a source code not related to the systems selected for the experiments and (ii) its duration was exactly the same for the experiment and the replications. We also provided the participants with a presentation with detailed instructions related to the tasks to be performed. Right before starting the experiment, we asked participants to fill in a pre-questionnaire aimed at gathering information on their background. We asked the following questions:

- Have you ever worked in industry? If yes, how long? Possible choice were *never*, *1 year*, *2-3 years*, *4-5 years*, *> 5 years*.
- How long have you been programming in Java? With a possible choice between *never*, *1 year*, *2-3 years*, *4-5 years*, *> 5 years*.
- How long have you been writing tests with the JUnit framework? With a possible choice between *never*, *1 year*, *2-3 years*, *4-5 years*, *> 5 years*.

At the end of both laboratory sessions, we also asked participants to fill-in a post-questionnaire (shown in Table 11) composed of questions expecting closed answers

Table 11 Post-experiment survey

Id	Question
Q1	I had enough time to perform the required task
Q2	The domain of the subject systems was perfectly clear to me
Q3	The objectives of the lab were perfectly clear to me
Q4	The tasks I had to perform were perfectly clear to me
Q5	I experienced no major difficulties in performing the tasks

according to the Likert (Oppenheim 1992) scale – from 1 (strongly agree) to 5 (strongly disagree) – to assess if the systems' domain and task were clear, if subjects had enough time, and other related questions.

The experimental results indicate that participants had the ability to correctly understand maintenance activities. This was evaluated by asking participants to answer a questionnaire (similar to the one used by Ricca et al. (2010)) consisting of 16 questions (eight for each system). Each of the eight questions for a system covers a different test smell and the JUnit class containing it. Note that we used the same question for both versions (with and without test smells) of a JUnit class, so the only difference is the presence of the test smell in the analyzed code. The questionnaire was in the form of a web-application able to (i) automatically balance the participants among the four experimental groups, (ii) show the questions to the participants in a random order to reduce the impact of learning effects and participant fatigue, and (iii) measure the time spent by each participant in answering each question.

Figure 2 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 (Bavota et al. 2013).

4.1.3 Variable Selection and Data Analysis

We performed a single factor within-participants 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 participant to correctly understand the maintenance activities, and **time**, which measures the time spent by the participant in answering each question. Concerning the **correctness**, for each of the required tasks i performed by a participant p it was possible to compute (i) the number of answers correct "by oracle" for i, (ii) the number of correct answers provided by p, and (iii) the number of wrong answers provided by p. For instance, in the task for *Test Code Duplication* shown in Fig. 2 there were four correct answers admitted (i.e., the four lines of code impacted by the change performed to the constructor of `PersisterToXML`: 46, 182, 241, 277). A participant answering *46, 182, 217* would have collected two correct and one wrong answer. Starting from these data, we used a combination of the two well known Information Retrieval metrics, recall and precision (Baeza-Yates and Ribeiro-Neto

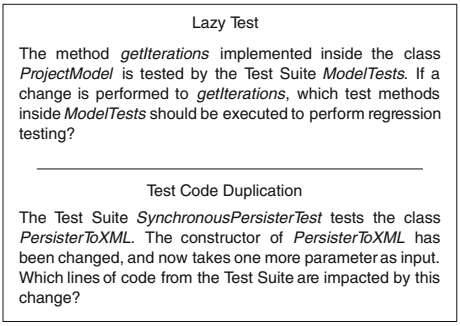

Fig. 2 AgilePlanner: sample questions

1999), to assess **correctness**. Let Q be the set of questions, these two metrics are defined as follows:

$$recall_{p,i} = \frac{\sum_{i \in Q} |answer_{p,i} \cap correct_i|}{\sum_{i \in Q} |correct_i|} \%$$

$$precision_{p,i} = \frac{\sum_{i \in Q} |answer_{p,i} \cap correct_i|}{\sum_{i \in Q} |answer_{p,i}|} \%$$

where $answer_{s,i}$ is the set of answers given by participant p 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 participant does not provide an answer to a given question (Antoniol et al. 2002). Finally, recall and precision measure two different (but related) concepts, and thus we use their harmonic mean (i.e., F-measure (Baeza-Yates and Ribeiro-Neto 1999)) to obtain a balance between them when measuring **correctness**:

$$F{-}measure_{p,i} = 2 * \frac{precision_{p,i} * recall_{p,i}}{precision_{p,i} + recall_{p,i}} \%$$

Going back to the previous described example, the participant providing two correct and one wrong answers would have achieved $recall = 0.5$, $precision = 0.67$, and $F{-}measure = 0.57$.

As for the **time**, we measured the time (in seconds) spent by the participants 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 (Conover 1998) was used to analyze the differences exhibited by participants working with and without test smells for both **correctness** and **time**. Moreover, we used a paired test because each participant 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). Differences were considered statistically significant at the $\alpha = 0.05$ level. We also estimated the magnitude of the difference between the number of changes for the two considered groups of releases (upgraded and not upgraded by clients) using the Cliff's Delta (or d), a non-parametric effect size measure (Grissom and Kim 2005) for ordinal data. We followed the guidelines of Cliff (Grissom and Kim 2005) to interpret the effect size values: small for $0.148 \leq d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Finally, to better assess the effect of the test smells on the participants"xt of our study, we identified the following confounding factors:

- **System**: since our experiments 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 confounding factor.
- **Lab**: as explained before, the experiments were organized in two laboratory sessions. Although the experimental design limits learning and fatigue effects, it is still important to analyze whether participants perform differently across subsequent lab sessions.

- **Experience**: in the context of our experiments we had four different populations of participants, i.e., Fresher, Bachelor, and Masters students, and Industrial developers. Since these participants have different level of experience in software development, design, and testing, we are interested in analyzing the effect of the participant's experience on the results.

To analyze the effects of the confounding factors on participant performance and their interaction with the main factor we used interaction plots (Devore and Farnum 1999). Interaction plots are simple line graphs where the means on the dependent variable for each level of one factor are plotted over all the levels of the second factor. The resulting lines are parallel when there is no interaction and nonparallel when interaction is present. For each set of controlled experiments, interaction plots are used to analyze the influence of participants' experience on the dependent variable. To perform such an analysis we considered all the participants, similar to other's work (Wohlin et al. 2000; Ricca et al. 2007). This was possible because for each set of experiments the design, material, and procedure were exactly the same. Indeed, the only difference among participants involved in the experiments was their experience, which is considered as an experimental confounding factor. Also, to statistically analyze the effects of confounding factors and their interaction with the main factor we performed the non-parametric Kruskal-Wallis one-way analysis of variance by ranks (Kruskal and Wallis 1952).

4.1.4 Replication Package

Most of the data and material used in this study is publicly available in our online appendix (Bavota et al. 2013). In particular, we provide (i) the complete questionnaire used in the experiments, (ii) the raw data of all four experiments, and (iii) all boxplots reporting participants' results. Unfortunately, we cannot make available the code of the industrial systems.

4.2 Analysis of the Results

This subsection discusses the result from the main experiment as well as the three replications. In particular, we first analyze the results of the pre-questionnaire conducted to assess participants' background. Then, we discuss the effect of the main factor (i.e., the presence/absence of test smells in the JUnit classes object of the required tasks) on the participants' performances in terms of **correctness** and **time**. After that, the impact of the confounding factors on the achieved results is presented. The analysis of the post-questionnaire answers close this subsection.

4.2.1 Pre-Questionnaire Results

Figure 3 shows the boxplots for participant answers to the pre-questionnaire. Concerning the industrial experience (left part of Fig. 3), Fresher, Bachelor, and Master students generally have not worked in industry (median=never), with some exceptions for the Master students. On the other side, industrial developers have industrial experience ranging from one to more than five years (median=from 2 to 3 years).

From the middle part of Fig. 3 it can be seen that Fresher students mostly had one year of programming experience in Java, Bachelor between two and three, Master between four and five, while industrial developers more than five.

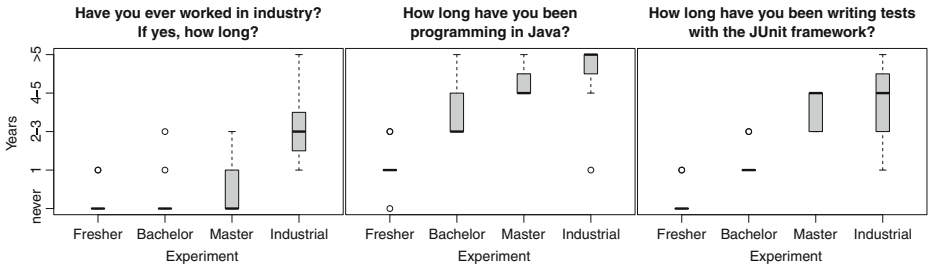

Fig. 3 Pre-Questionnaire answers provided by participants in the four performed experiments

Finally, the years of experience with the JUnit framework (right part of Fig. 3) were generally zero for Fresher students, one for Bachelor, and between four and five for Master students and Industrial developers.

From the analysis of Fig. 3, it is also worth noting that the participants in each experiment had a very homogeneous background, at least for the factors considered in our pre-questionnaire.

4.2.2 Effect of the Main Factor

Figures 4, 5, 6, 7, 8, 9, 10, and 11 report the boxplots for the dependent variables, **correctness** and **time**, achieved by participants when working on JUnit tests with and without test smells. In particular, each figure refers to a specific test smell (e.g., Fig. 4 shows the results for the *Mystery Guest* smell) and reports the results of all four experiments (i.e., "Fresher", "Bachelor", "Master", and "Industrial" in the figures) as well as when considering all participants as a unique dataset[5] (i.e., "All" in the figures). Results in terms of **correctness** are reported in the upper part of each figure, while **time** data are shown in the bottom part. The red crosses present present on each boxplot represent the 95 % confidence interval for the represented distribution of data.

In addition, Table 12 reports Wilcoxon test p-values, the Cliff's d effect size, and the descriptive statistics for the differences in performance on code with and without test smells. Also in this table results are shown for all four experiments in isolation as well as considering all participants as a single dataset.

To get a quick idea of the differences in terms of **correctness** with and without test smells, the participant F-Measure without test smells is 29 % higher than with test smells for Fresher students (70 % *vs* 41 %), 32 % for Bachelors students (83 % *vs* 51 %), 37 % for Masters students (89 % *vs* 52 %), and 22 % higher for Industrial developers (93 % *vs* 71 %). In the following, results are discussed by test smell.

Mystery Guest Results for tasks related to the *Mystery Guest* test smell are reported in Fig. 4. As it is clear by the distributions reported in the upper part of Fig. 4, the presence of this test smell in the JUnit classes strongly lowered the participants **correctness**. On average, this decreases is 59 % for Fresher students, 40 % for Bachelor students, 61 % for Master students, and 25 % for Industrial developers (49 % on the entire dataset). Remember

[5]Note that this is possible since the only difference among the four experiments are the involved participants.

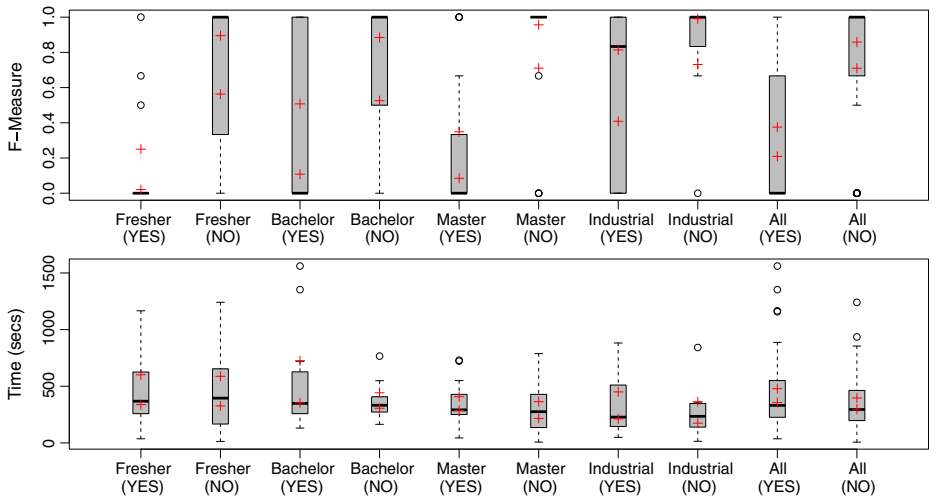

Fig. 4 *Mystery Guest*: Boxplots of participants performances (in terms of F-measure and time spent) with (YES) and without (NO) smell

that a test suite affected by this smell *"uses external resources, such as a file containing test data"* (Van Deursen et al. 2001). Our questionnaire asked participants 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 (Van Deursen et al. 2001) had been applied, putting the test data inside a String defined in the test suite. The effect of this simple refactoring was very strong. Looking at the results of the Wilcoxon test reported in Table 12 it is interesting to note that the only experiment in which the differences of **correctness** achieved by participants with

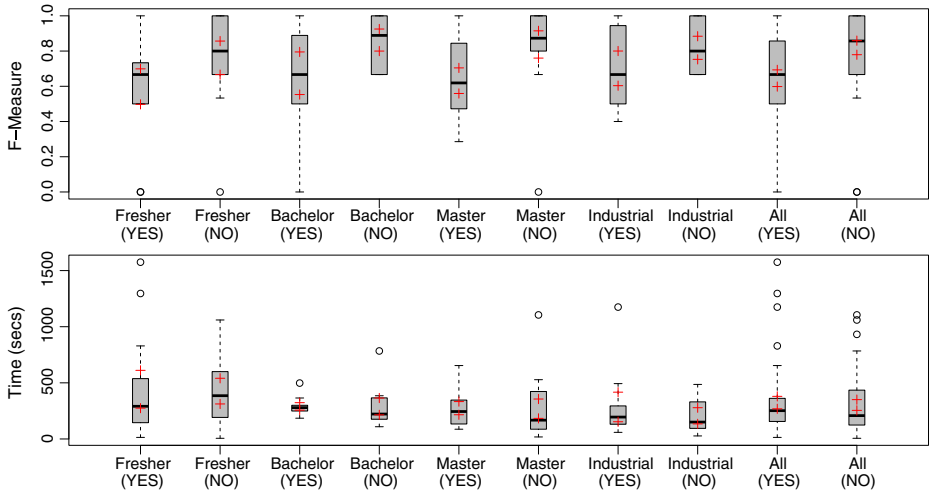

Fig. 5 *General Fixture*: Boxplots of participants performances (in terms of F-measure and time spent) with (YES) and without (NO) smell

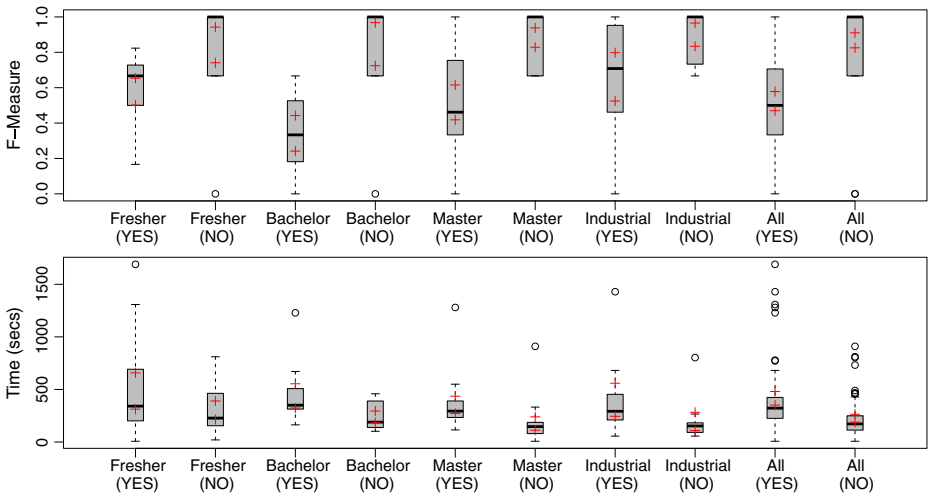

Fig. 6 *Eager Test*: Boxplots of participants performances (in terms of F-measure and time spent) with (YES) and without (NO) smell

and without the *Mystery Guest* is not statistically significant is the one involving industrial developers (p-value 0.06). As explained before, while industrial developers have suffered an average lost in **correctness** of 25 %, on average, they are less affected by the smell presence. This result is likely due to their higher experience in dealing with complex test classes making heavy use of test data. Concerning the other experiments, instead, we not only observe statistically significant difference, but also a medium or large effect size (the effect size is also large on the entire dataset).

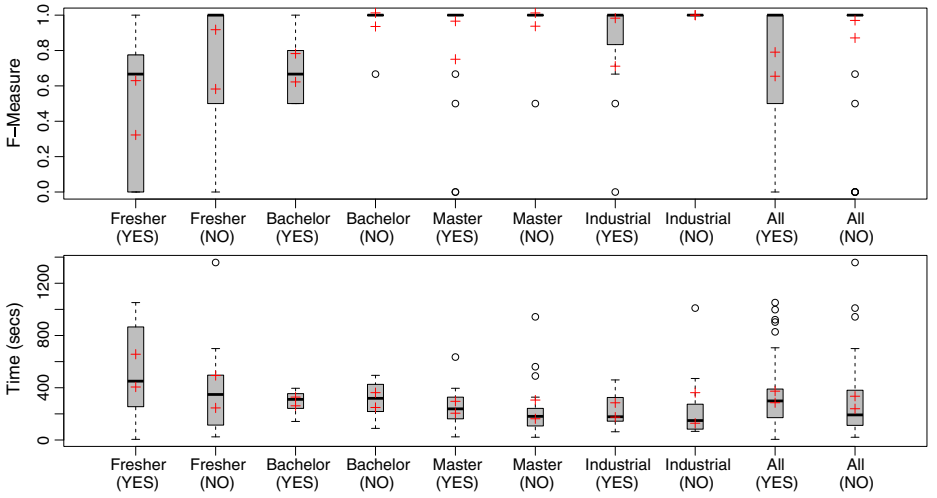

Fig. 7 *Lazy Test*: Boxplots of participants performances (in terms of F-measure and time spent) with (YES) and without (NO) smell

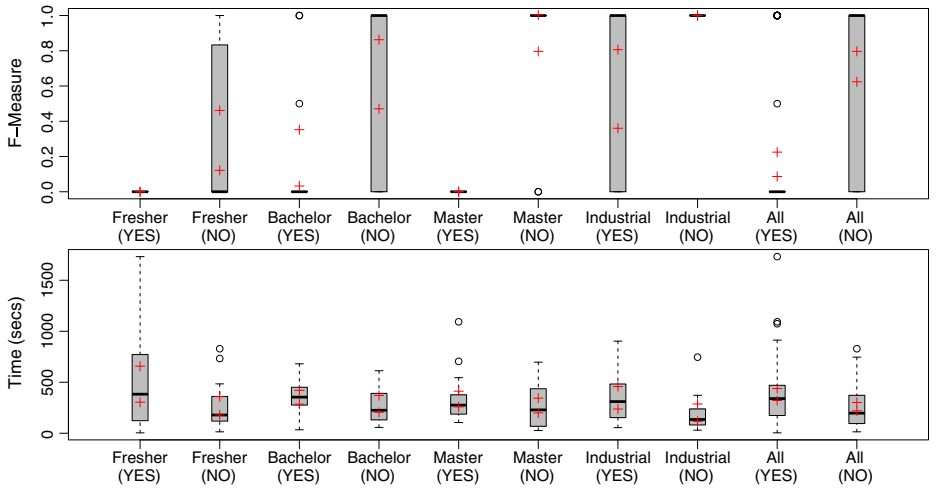

Fig. 8 *Assertion Roulette*: Boxplots of participants performances (in terms of F-measure and time spent) with (YES) and without (NO) smell

In terms of the **time** spent performing the tasks, we did not observe any particular effect of the test smell presence (see bottom part of Fig. 4). While participants were able to save some seconds when the JUnit class was not affected by the *Mystery Guest* smell (70, on average over the entire dataset), this difference was never statistically significant (see Table 12).

General Fixture Also the presence of the *General Fixture* test smell had a negative impact on participants performances (see Fig. 5) even if, as compared to the *Mystery Guest* smell, it was more limited. In particular, in terms of **correctness**, 16 % was paid, on average, by

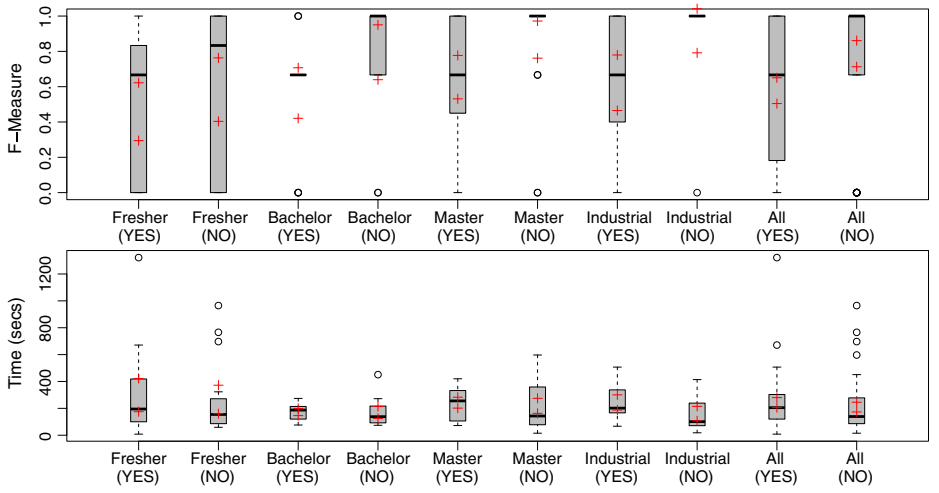

Fig. 9 *Indirect Testing*: Boxplots of participants performances (in terms of F-measure and time spent) with (YES) and without (NO) smell

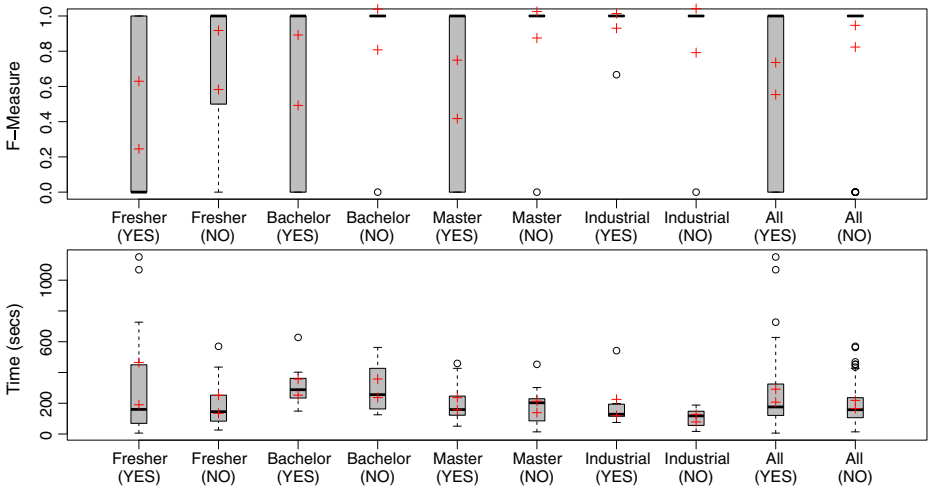

Fig. 10 *Sensitive Equality*: Boxplots of participants performances (in terms of F-measure and time spent) with (YES) and without (NO) smell

Fresher students, 19 % by Bachelor, 20 % by Master, and 12 % by Industrial (17 % on the entire dataset). Also in this case these differences are all statistically significant (with a medium or large effect size) but the one for industrial developers (see Table 12). Thus, also in this case industrial developers are less impacted by the presence of the test smell in the JUnit class.

For this smell it is also quite interesting to observe how the different severity of the instances considered on AgilePlanner and eXVantage impacted participant performance. Indeed, as shown in Table 9 the severity of this smell is substantially higher on the eXVantage instance with respect to the AgilePlanner one. Overall, when considering the entire

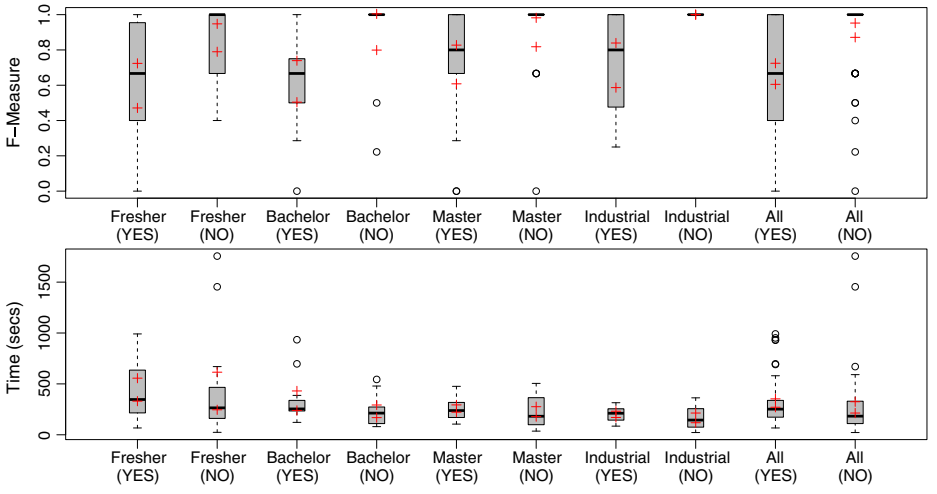

Fig. 11 *Test Code Duplication*: Boxplots of participants performances (in terms of F-measure and time spent) with (YES) and without (NO) smell

Table 12 Wilcoxon test for **correctness** and **time** by test smell

Test Smell	**correctness**					**time**				
	NoTS$FM - TS$FM			p-value	eff. size	NTS$Time - TS$Time			p-value	eff. size
	Mean	Med	StDv			Mean	Med	StDv		
16 FRESHER STUDENTS										
Myst. Guest	0.59	0.83	0.46	<**0.01**	-0.65	-13	-80	313	0.39	0.04
Gen. Fixture	0.16	0.21	0.36	**0.03**	-0.44	-17	25	325	0.48	-0.06
Eager Test	0.26	0.31	0.29	<**0.01**	-0.64	-182	-206	412	**0.04**	0.23
Lazy Test	0.27	0.23	0.34	<**0.01**	-0.45	-162	-115	343	**0.02**	0.32
Assert. Roul.	0.29	0.00	0.45	**0.02**	-0.31	-212	-83	492	0.07	0.24
Ind. Testing	0.13	0.00	0.56	0.20	-0.19	-33	-23	233	0.34	0.08
Sens. Eq.	0.31	0.00	0.70	0.06	-0.31	-134	-106	337	0.09	0.07
Code Duplic.	0.27	0.27	0.41	**0.01**	-0.50	-13	-19	476	0.33	0.22
13 BACHELOR STUDENTS										
Test Smell	**correctness**					**time**				
	NoTS$FM - TS$FM			p-value	eff. size	NTS$Time - TS$Time			p-value	eff. size
	Mean	Med	StDv			Mean	Med	StDv		
Myst. Guest	0.40	0.50	0.62	**0.04**	-0.41	-166	-6	499	0.20	0.11
Gen. Fixture	0.19	0.17	0.24	**0.02**	-0.41	0	-27	168	0.34	0.11
Eager Test	0.50	0.48	0.38	<**0.01**	-0.83	-200	-155	338	**0.04**	0.53
Lazy Test	0.27	0.33	0.21	<**0.01**	-0.71	10	-15	156	0.64	-0.05
Assert. Roul.	0.47	1.00	0.65	**0.02**	-0.50	-65	-65	266	0.22	0.27
Ind. Testing	0.23	0.33	0.46	0.06	-0.48	-3	-6	135	0.39	0.14
Sens. Eq.	0.23	0.00	0.44	0.07	-0.23	-7	-24	183	0.47	0.04
Code Duplic.	0.28	0.33	0.43	**0.02**	-0.62	-105	-98	197	**0.02**	0.32
20 MASTER STUDENTS										
Test Smell	**correctness**					**time**				
	NoTS$FM - TS$FM			p-value	eff. size	NTS$Time - TS$Time			p-value	eff. size
	Mean	Med	StDv			Mean	Med	StDv		
Myst. Guest	0.62	1.00	0.60	<**0.01**	-0.67	-57	-22	336	0.27	0.19
Gen. Fixture	0.21	0.21	0.26	<**0.01**	-0.55	-5	-22	327	0.43	0.13
Eager Test	0.37	0.34	0.37	<**0.01**	-0.76	-178	-207	326	<**0.01**	0.70
Lazy Test	0.12	0.00	0.35	0.11	-0.15	-16	-18	264	0.23	0.23
Assert. Roul.	0.90	1.00	0.31	<**0.01**	-0.90	-63	-82	383	0.28	0.18
Ind. Testing	0.21	0.33	0.54	0.05	-0.39	-24	-12	204	0.32	0.10
Sens. Eq.	0.37	0.00	0.48	<**0.01**	-0.40	-20	-69	161	0.31	0.05
Code Duplic.	0.18	0.14	0.41	**0.01**	-0.42	-31	-81	175	0.29	0.15

Table 12 (continued)

12 INDUSTRIAL DEVELOPERS

Test Smell	correctness					time				
	NoTS$FM - TS$FM			p-value	eff. size	NTS$Time - TS$Time			p-value	eff. size
	Mean	Med	StDv			Mean	Med	StDv		
Myst. Guest	0.25	0.00	0.53	0.06	-0.29	-62	35	435	0.48	0.06
Gen. Fixture	0.12	0.15	0.30	0.14	-0.33	-79	-25	379	0.36	0.24
Eager Test	0.24	0.29	0.41	**0.03**	-0.54	-206	-164	458	**0.03**	0.59
Lazy Test	0.15	0.00	0.31	0.09	-0.25	15	-28	303	0.28	0.18
Assert. Roul.	0.42	0.00	0.51	**0.02**	-0.42	-146	-148	363	0.07	0.42
Ind. Testing	0.29	0.33	0.53	0.05	-0.54	-83	-66	199	0.08	0.36
Sens. Eq.	-0.05	0.00	0.19	0.98	0.00	-68	-37	129	0.07	0.35
Code Duplic.	0.28	0.20	0.29	**0.01**	-0.67	-35	-59	149	0.23	0.24

ENTIRE DATASET (61 PARTICIPANTS)

Test Smell	correctness					time				
	NoTS$FM - TS$FM			p-value	eff. size	NTS$Time - TS$Time			p-value	eff. size
	Mean	Med	StDv			Mean	Med	StDv		
Myst. Guest	0.49	0.67	0.56	<**0.01**	-0.54	-70	-6	384	0.14	0.11
Gen. Fixture	0.17	0.17	0.29	<**0.01**	-0.46	-21	-11	305	0.36	0.06
Eager Test	0.34	0.33	0.37	<**0.01**	-0.71	-189	-203	371	<**0.01**	0.49
Lazy Test	0.20	0.00	0.32	<**0.01**	-0.37	-42	-51	280	**0.03**	0.16
Assert. Roul.	0.55	1.00	0.53	<**0.01**	-0.56	-119	-84	387	**0.02**	0.27
Ind. Testing	0.21	0.33	0.52	<**0.01**	-0.38	-33	-28	196	0.12	0.18
Sens. Eq.	0.24	0.00	0.52	<**0.01**	-0.26	-57	-58	221	**0.04**	0.11
Code Duplic.	0.25	0.20	0.39	**0.01**	-0.52	-43	-68	281	**0.02**	0.21

NoTS = NoTestSmell - TS = TestSmell

dataset, participants achieved an average F-Measure of 69 % when working on the AgilePlanner *General Fixture* smell instance, against the 60 % with eXVantage. On the other side, when working on the refactored source code, participants performed better on eXVantage (89 %, on average) than on AgilePlanner (76 %, on average). Thus, the higher severity of the eXVantage instance had a stronger negative impact on participant performance with respect to the AgilePlanner instance.

Concerning the **time** spent by participants, also in this case we never observed statistically significant differences for tasks performed on JUnit classes *affected* and *not affected* by test smell, as it is also clear by looking at the distributions reported in the bottom part of Fig. 5.

Eager Test Figure 6 shows data gathered from the tasks related to the *Eager Test* smell. The presence of this smell resulted in several difficulties for participants. Fresher students decreased their performance by 26 %, Bachelors students by 51 %, Masters students by 36 %, and Industrial developers by 24 %. This smell arises when *"a test method checks several methods of the tested object* (Van Deursen et al. 2001). We asked participants to identify

all methods in the production code exercised by a given test method. When analyzing their responses, we noticed that their recall (i.e., their ability of identifying all correct exercised methods) is strongly negatively impacted by the presence of *Eager Test* while the smallest effects were observed on precision. In particular, Fresher students decrease their recall by 34 % (from 84 % to 50 %) and their precision by just 10 % (from 88 % to 78 %), Bachelors students decrease their recall by 51 % (from 81 % to 30 %) and their precision by 38 % (from 91 % to 53 %), Masters students decrease their recall by 37 % (from 85 % to 48 %) and their precision by 29 % (from 97 % to 68 %), and Industrial developers achieved a -36 % in terms of recall (from 84 % to 38 %) and a -25 % in terms of precision (from 94 % to 69 %). Thus, participants found it quite difficult identifying all methods exercised by a test method affected by *Eager Test*. This is also confirmed by the fact that the differences in terms of **correctness** achieved by participants with and without this test smell are statistically significant and with a large effect size for all experiments as well as when considering the entire dataset.

The *Eager Test* smell is also the only one for which we observed a significant difference in terms of time spent by participants performing the required task in all four experiments with an effect size going from small for Fresher students (0.23) to large for Bachelor students (0.53), Master students (0.70), and Industrial developers (0.59)–see Table 12. This result is somewhat expected since this smell was removed using the Extract Method Refactoring (Fowler 1999), which separates the test code into several test methods each of which only tests one method. Thus, the time needed to answer the question in absence of the test smell was considerably lower.

Also the instances of *Eager Test* selected from AgilePlanner and eXVantage had a different severity (see Table 9), with eXVantage presenting the more severe instance. However, participants generally achieved better performances when working with eXVantage for this smell achieving, on average, an F-Measure of 63 % versus the 43 % achieved with AgilePlanner. Thus, we further investigated the code of the JUnit classes of the two object systems. We found that the method affected by this smell on AgilePlanner (i.e., `testUpdatedIterationModelAndStoryCardModel()`) is considerably more complicated than the one affected by it from eXVantage (i.e., `test1()`). In particular, the AgilePlanner method has 51 LOC and 9 objects declared in it, against the 27 LOC and 3 objects declared in the eXVantage method. Thus, the lower **correctness** achieved by participants on AgilePlanner is likely due to the more complex method object of the task related to this smell. Indeed, also when performing the task on the refactored version of the same method, participants achieved much better levels of F-Measure on eXVantage (95 % on average) than on AgilePlanner (78 % on average). Note that the different complexity of the two instances of *Eager Test* present in AgilePlanner and eXVantage do not represents a bias for our results. On the contrary, it highlights the negative effect of test smells presence on participants performances independently from complexity of the code component affected by the smell.

Lazy Test Results achieved for the *Lazy Test* smell show an interesting trend related to participants experience. While all participants achieved a better **correctness** in absence of this smell (on average, +27 % for Fresher, +50 % for Bachelor, +12 % for Master, and +15 % for Industrial), this difference is statistically significant only for Fresher and Bachelor students. Thus, the **correctness** of the more experienced participants involved in our experiments were not affected in a statistically significant manner. Remember that we have a *Lazy Test* when *"several test methods check a method of the tested class using the same fixture"* (Van Deursen et al. 2001). Given the method t_m of the tested object exercised by several

methods of the affected JUnit class, we asked participants to identify, as consequence of an hypothetical change to t_m, the test methods to execute in order to perform regression testing. The more expert participants, likely more used to such complex situation, were able to achieve good performance also in presence a *Lazy Test* smell. On the other side, participants with a low experience are likely more used to work on simple systems, mostly exhibiting a one-to-one relationship between test and production methods.

In terms of **time**, only the Fresher students exhibit a statistically significant difference saving, on average, 161 seconds when performing the required task without the *Lazy Test* smell. In the other experiments, we did not observe any significant difference in terms of time spent, even though participants generally save time when the test smell is not present. This is also confirmed by the fact that, when considering the whole dataset, the difference in terms of time spent is statistically significant for the *Lazy Test* smell, even if with a small effect size (0.16).

Assertion Roulette The *Assertion Roulette* is one of the test smells that more strongly impacts **correctness**. This is particularly important given the fact that it is also the most frequent test smell in the 27 projects analyzed in Section 3, occurring in 55 % of the JUnit tests. The distributions reported in Fig. 8 clearly show that when this test smell is present participants are generally not able to perform the required maintenance activity. In fact the F-Measure is different from zero just in the experiment performed with Bachelors students (19 %) and with Industrial developers (58 %) where, however, it is still strongly negatively impacted by the presence of the smell (Bachelor students paid, on average, 48 % correctness, while Industrial developers 42 %). In the questions related to this test smell we asked participants 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"* (Van Deursen et al. 2001) and thus if one of the assertions fails it is difficult to identify it since no explanation is present in the reported error trace. This is the cause of the very low F-Measure achieved by participants in presence of this test smell. On the other side, when this test smell is not present, Industrial developers achieve, on average, 100 % **correctness**, Masters students 90 %, Bachelors students 67 % while Fresher students still exhibit difficulties in performing the required task (achieving only a 29 % F-measure). This result is likely due to their lower level of experience.

The bottom part of Fig. 8 also shows that all participants saved time when performing the task on the refactored JUnit classes (i.e., the ones without the *Assertion Roulette* smell). However, this difference is statistically significant only when considering the entire dataset (see Table 12) where it has a small effect size.

Indirect Testing For this smell we did not observe any statistically significant difference in the **correctness** achieved by participants. This holds for all four experiments as well as when considering the entire dataset as a whole (see Table 12). Despite this, by looking at Fig. 9 it is clear that some negative effect on **correctness** results from the test smell's presence. Remember that this smell arises when *"a test interacts with the object under test indirectly via another object"* (Van Deursen et al. 2001). In the task related to this smell we asked participants to identify the class tested by the JUnit class affected by *Indirect Testing*. Our results indicates that, while the identification of the tested class is more challenging when the testing is *indirectly* performed, participants achieved good results in presence of the test smell. Also in terms of **time** no significant effect of the smell's presence was observed.

Sensitive Equality Figure 10 shows that the presence of this smell has a (low) negative impact on participants performance in terms of both **correctness** and **time**. However, Industrial developers seem to be not affected by the presence of this smell. The Wilcoxon test highlights a significant negative impact on **correctness**, but only with Master students (see Table 12).

Test Code Duplication As can be observed from Fig. 11, this smell has a strong negative impact on participants **correctness**. In particular, in presence of this smell Fresher students decreased their F-Measure by 27 % (from 87 % to 60 %), on average, Bachelor by 23 % (from 90 % to 62 %), Master by 18 % (from 90 % to 72 %), and Industrial by 29 % (from 100 % to 71 %). Thus, surprisingly, this is the only bad smell for which industrial developers seem to be more impacted than the students. While it is not easy to explain this result, it is possible that students are used to duplicated code spread among projects they worked on during their academic career. All the differences observed in terms of **correctness** are statistically significant, even if with a small effect size (see Table 12). On the **time** side, by observing Fig. 11 it is clear that participants saved time on the refactored code. However, the observed differences are not statistically significant except with the Master students and when considering the whole dataset.

For this smell, it is also interesting to analyze the impact of the different severity of the instances present in the two object systems. As seen in Table 9, the instance from eXVantage is more severe than the one from on AgilePlanner. Participants achieved, on average, 73 % when working on the *Test Code Duplication* instance from AgilePlanner, against 59 % for eXVantage. Instead, when the smell was not present, the performance were almost the same for the two systems (92 % for AgilePlanner *vs* 91 % for eXVantage). This means that the more severe instance of this smell present on eXVantage actually had a stronger negative impact on participant performance than the one present in AgilePlanner.

Results Summary Results achieved in the four performed experiments provided us with two important lessons learned:

1. test smells generally have a negative impact on the **correctness** of the task performed by participants. When considering the entire dataset, this impact is statistically significant for all test smells. On the other side, no strong influence of **time** was observed even though, generally, participants appeared to save time when performing the tasks on the refactored code. These findings highlight the potential harmfulness of test smells.
2. The intensity of the impact of the test smells is different for participants having different levels of experience. For instance, six test smells have a significant, negative impact on the experiments performed with students, while for industrial developers the number of significantly impacting test smells goes down to three. While it is important to highlight that industrial developers achieved better results on the refactored code (and this result holds for all test smells), this finding suggests that tasks being performed on poor-written test cases should be assigned to the most experienced developers.

4.2.3 Influence of Confounding Factors

As explained in Section 4.1.3, to better asses the effect of the main factor (i.e., **TestSmells** – the presence or not of test smells in JUnit classes), we also analyzed the influence of three confounding factors: (**System**, **Lab**, **Experience**) on the participant performance.

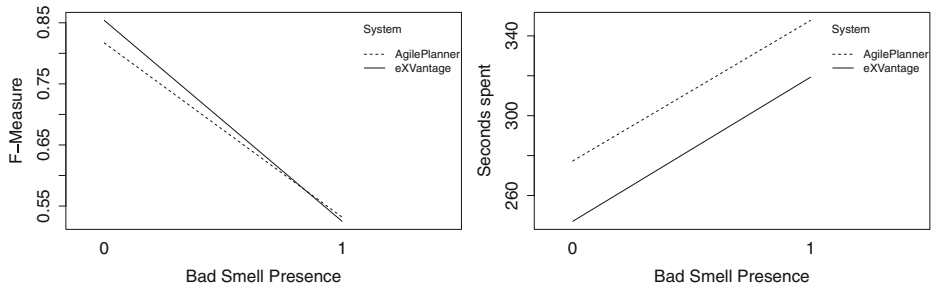

Fig. 12 Interaction plots for the confounding factor System

Figures 12, 13, and 14 reports the interaction plots for **System**, **Lab**, and **Experience**, respectively. Also, Table 13 shows the results of the KruskalWallis one-way analysis of variance.

Concerning the **System** we do not observe any interaction on the F-Measure achieved by participants (the left of Fig. 12), as confirmed using the KruskalWallis test, which reports a p-value of 0.54. On both systems participants achieved a higher F-Measure when working on JUnit classes without test smells. As for the average number of seconds spent by participants on each task (see right part of Fig. 12), participants working on AgilePlanner generally spent around 30 seconds longer per maintenance task as compared to participants working with eXVantage. This difference is almost stable when working on both JUnit classes with and without test smells. The interaction of the **System** on the **time** spent performing the tasks is statistically significant (p-value$<$0.01).

The absence of interactions with **System** on the F-Measure and the higher time spent by participants on AgilePlanner might seem surprising given the fact that, as shown in Table 9, the smell severity is generally higher in eXVantage. However, as previously explained, this is likely due to the higher complexity of the AgilePlanner code when compared with that of eXVantage. Since it is difficult to verify such a conjecture with metrics (for example, a larger class does not always imply a more complex class) we asked at the end of the experimentation the participants opinion in our fourth experiment[6] (i.e., the one performed with industrial developers). Ten out of the twelve developers confirmed that the AgilePlanner code is in general more intricate and difficult to comprehend with respect to the eXVantage one. The remaining two developers did not notice any significant difference between the two systems.

For the **Lab** confounding factor, we do not observe any influence on participant F-Measure (see left part of Fig. 13). In fact, the lines representing the two laboratory sessions virtually overlap. The absence of interaction between the participant F-Measure and **Lab** is confirmed by the results of the Kruskal-Wallis test (p-value=0.18–see Table 13). On the other hand, the right side of Fig. 13 shows a small interaction of **Lab** on the time (in seconds) spent by participants in performing the required tasks. In this case, for the tasks performed during *Lab 1* the difference between the tasks performed on JUnit classes with and without test smells is stronger than in *Lab 2*. This could be due to a slight

[6]Note that we were not able to do the same in our previous experiments since the need to verify our conjecture came out after a first analysis performed on the data achieved in the first three experiments.

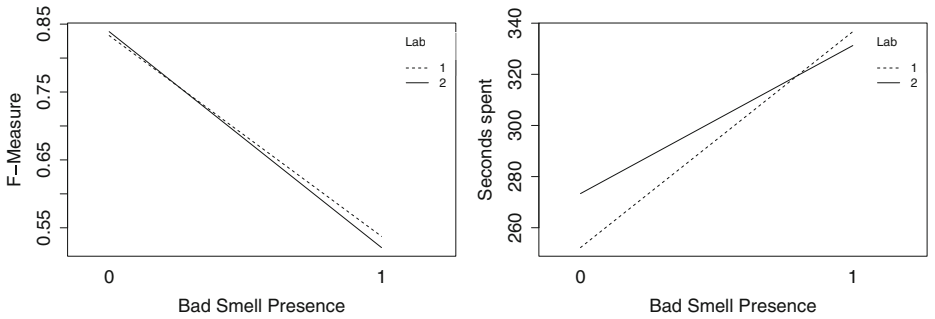

Fig. 13 Interaction plots for the confounding factor Lab

learning effect experienced by the participants. However, it is worth noting that (i) this interaction is not statistically significant (p-value=0.32) and (ii) in both labs participants spent more time working on JUnit classes affected by test smells.

Finally, Fig. 14 reports interaction plots for the **Experience** confounding factor. The observation derived from these plots is clear: *participants having higher experience achieve a higher F-Measure (on both JUnit classes affected and not by test smells) spending less time*. In fact, we can observe higher F-Measures for Industrial developers, followed by Masters students, Bachelors students, and then Fresher students. Industrial developers achieved an overall F-Measure of 82 %, Masters students 71 %, Bachelors students 67 %, and Fresher students 55 %. As for the time, Industrial developers spent on average 235 seconds per task, Masters students 258, Bachelors students 308, and Fresher students 387. However, it is interesting to note how all participants are equally impacted by the presence of test smells (the experience's lines are almost parallel) independently from their experience, thus confirming again the harmfulness of test smells when performing maintenance activities on JUnit test suites. Table 13 shows that the impact of the participants experience on both **correctness** and **time** is statistically significant (<0.01 in both cases).

4.2.4 Post-Questionnaire Results

Figure 15 shows the boxplots of the answers to our post-questionnaire. Participants generally *agree* that the time allowed to complete the required tasks was sufficient, with

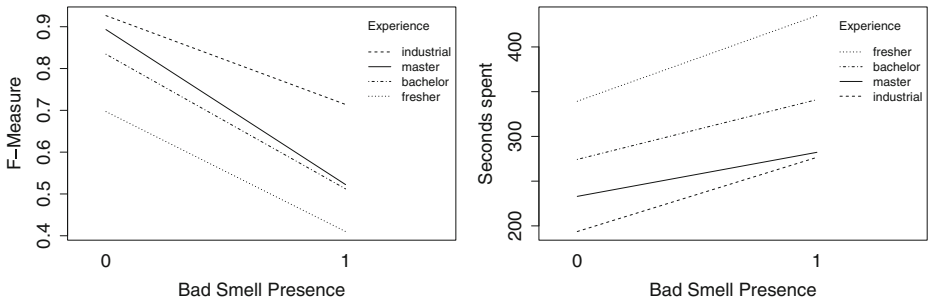

Fig. 14 Interaction plots for the confounding factor Experience

Table 13 Kruskal-Wallis one-way analysis of variance

Test	p-value
F-Measure∼System	0.54
Seconds spent∼System	<**0.01**
F-Measure∼Lab	0.18
Seconds spent∼Lab	0.32
F-Measure∼Experience	<**0.01**
Seconds spent∼Experience	<**0.01**

the exception of the experiment performed with Bachelor students, where the median of answers is *neutral* (left-upper part of Fig. 15).

Concerning the clearness of the experiment, the domain of the subject systems was clear to participants (median=*agree* for all experiments) as well as the objectives of the lab (right-upper part of Fig. 15) and the tasks to be performed (left-bottom part of Fig. 15).

When asking about the difficulties experienced in performing the required tasks (right-bottom part of Fig. 15), as expected Fresher students experienced the most difficulties, followed by Bachelor students, Master students, and finally Industrial developers.

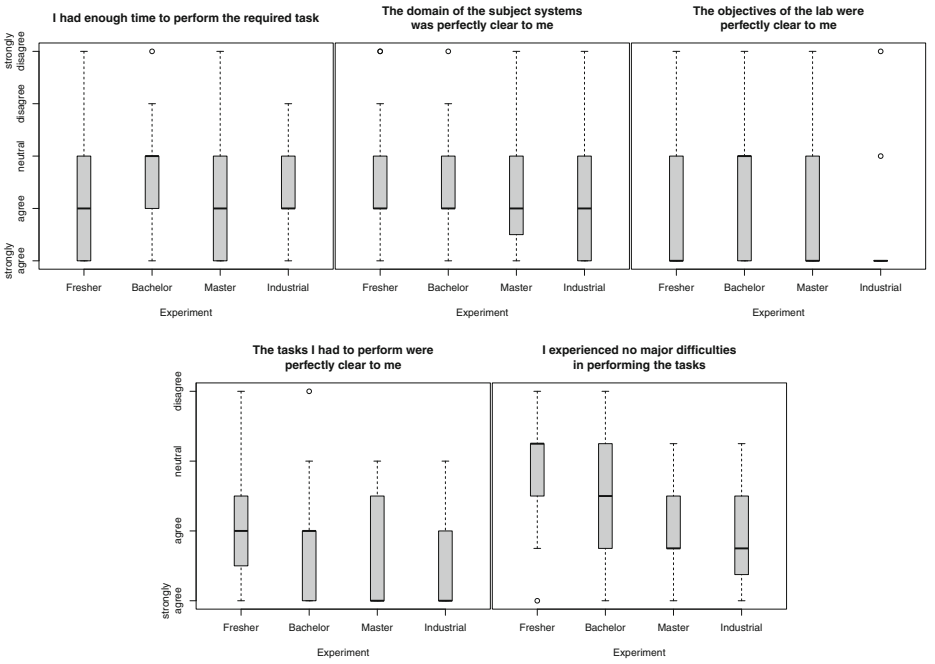

Fig. 15 Post-Questionnaire answers provided by participants in the four performed experiments

4.3 Threats to Validity

In this section we discuss the threats that could affect the validity of our results, focusing the attention on *construct*, *internal*, *external*, and *conclusion* validity threats.

Concerning the *Construct* validity threats that may be present in this experiment, the interactions between different treatments were mitigated by a proper design that allowed us to separate the analysis of the different factors and of their interactions. To avoid social threats due to evaluation apprehension, students were not evaluated on their performance. During the experiment, we monitored the participants to verify whether they were motivated and paid attention to their assigned task. We observed that all participants performed the required task with dedication and there was no abandonment. Moreover, subjects were not aware of the goal of our experiment nor of the dependent variables. Another *construct* validity threat concerns the way the test smells have been refactored. To limit this threat, we paid attention to refactoring operations aimed solely at removing the test smells, without performing any unnecessary changes aimed at improving other aspects of the code. However, we cannot exclude the possibility of other influences on the results resulting from the undertaken refactorings.

Internal validity threats can be due to the learning (or tiring) effect experienced by participants between labs. We tried to mitigate these issues through the experiment design: participants worked, over the two labs, on different tasks. Nevertheless, there is still the risk that, during labs, participants might have learned how to improve their performance. We tried to limit this effect by means of a preliminary training phase. In addition, one possible issue related to the chosen experimental design concerns the possible information exchange among the participants between the laboratories. To mitigate such a threat the experimenters monitored participants during the experiment execution to avoid collaboration and communication between them. Finally, participants worked on different systems during the two labs. In particular, we performed the experiment on two industrial systems (AgilePlanner and eXVantage) because both have at least one instance of each test smell and belong to the same category of systems. This choice helps to reduce the possibility for the development style acting as a confounding factor.

As for the *conclusion validity*, 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 (i.e., the paired Wilcoxon test) was used. We verified these conditions using the non-parametric Wilk-Shapiro test (Conover 1998).

Finally, concerning the generalization of our findings (*external validity*), there are two main threats. The first concerns the subjects involved in our study. While we did our best to involve participants having different levels of experience, replicating our study with a different population might lead to different results.

A second threat to the external validity is represented by the questions chosen to test the effects of the test smells on software maintenance. In designing our survey we tried to balance two contrasting goals. First, we needed to ensure that the required tasks involved in some way the refactored test code. For instance, a task performed on refactored *Assertion Roulette* code is useless to assess the impact of this test smell if it does not involve in any way the assert statements of the refactored test method. On the other hand, we also needed to limit as much as possible any bias in the questions. These two goals are somewhat at odds because, by definition, we expect the refactored code to be simpler than that containing the test smell (making the task potentially easier with the refactored code). For this reason, a set of different questions (tasks) might lead to different results.

5 Conclusion and Future Work

Test smells represent a potential danger to the maintainability of production code and test suites. This paper presented the first empirical evidence highlighting that test smells (i) occur quite frequently in software systems and (ii) negatively impact programmer comprehension during maintenance activities.

Replications in different contexts, with different participants and objects, is the only way to corroborate our findings. It would be interesting to consider alternative experimental settings in several respects, but maybe the most important one is the profile of the participants involved and the object systems. Replicating this study with students/professionals having different backgrounds would be useful in understanding how well our findings generalize. To facilitate replication, the materials used and the raw data of the performed studies are available online (Bavota et al. 2013).

Finally, our results highlight the importance for the community to develop methods and tools able to (i) detect candidate test smell instances and (ii) automatically refactor them.

References

Abbes M, Khomh F, Guéhéneuc YG, Antoniol G (2011) 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. IEEE Comput CS Press, Oldenburg, pp 181–190

Abbes M, Khomh F, Guéhéneuc YG, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: 15th european conference on software maintenance and reengineering, CSMR 2011, 1-4 March 2011. IEEE Computer Society, Oldenburg, pp 181–190

Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng 28(10):970–983

Arcoverde R, Garcia A, Figueiredo E (2011) Understanding the longevity of code smells: preliminary results of an explanatory survey. In: Proceedings of the international workshop on refactoring tools. ACM, pp 33–36

Baeza-Yates R, Ribeiro-Neto B (1999) Modern information retrieval. Addison-Wesley

Baker P, Evans D, Grabowski J, Neukirchen H, Zeiss B (2006) Trex-the refactoring and metrics tool for ttcn-3 test specifications. In: TAIC PART, pp 90-94

Bavota G, Qusef A, Oliveto R, DeLucia A, Binkley D (2013) Are test smells really harmful? an empirical study. Tech. rep. http://www.dmi.unisa.it/people/bavota/www/reports/TestSmells/

Bavota G, Qusef A, Oliveto R, Lucia AD, Binkley D (2012) An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: ICSM, pp 56–65

Breugelmans M, Van Rompaey B (2008) 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)

Chatzigeorgiou A, Manakos A (2010) Investigating the evolution of bad smells in object-oriented code. In: In: QUATIC, IEEE Computer Society

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

Conover WJ (1998) Practical nonparametric statistics, 3rd edn, Wiley

Deligiannis IS, Shepperd MJ, Roumeliotis M, Stamelos I (2003) An empirical investigation of an object-oriented design heuristic for maintainability. J Syst Softw 65(2):127–139

van Deursen A, Moonen L (2002) 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, pp 71–76

Devore JL, Farnum N (1999) Applied statistics for engineers and scientists, Duxbury

van Emden E, Moonen L (2002) Java quality assurance by detecting code smells. In: Proceedings of the 9th working conference on reverse engineering (WCRE'02). IEEE CS Press. citeseer.ist.psu.edu/vanemden02java.html

Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley
Greiler M, van Deursen A, Storey MAD (2013) Automated detection of test fixture strategies and smells. In: IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, pp 322–331
Greiler M, Zaidman A, van Deursen A, Storey MAD (2013) Strategies for avoiding text fixture smells during software evolution. In: In: MSR
Grissom RJ, Kim JJ (2005) Effect sizes for research: a broad practical approach, 2nd edn. Lawrence Earlbaum Associates
Hindle A., Godfrey M., Holt R. (2007) Release pattern discovery via partitioning: methodology and case study. In: 4th international workshop on mining software repositories, 2007. ICSE Workshops MSR '07
Khomh F, Di Penta M, Gueheneuc YG (2009) An exploratory study of the impact of code smells on software change-proneness. In: Proceedings of the 2009 16th working conference on reverse engineering, WCRE '09. IEEE Comput Soc
Khomh F, Penta MD, Guéhéneuc YG, Antoniol G (2012) An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empir Softw Eng 17(3):243–275
Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2009) A bayesian approach for the detection of code and design smells. In: Proceedings of the 9th International Conference on Quality Software. IEEE CS Press, Hong Kong, pp 305–314
Kruskal W. H., Wallis W. A. (1952) Use of ranks in one-criterion variance analysis. J Am Stat A 47(260):583–621
Lanza M, Marinescu R (2006) Object-oriented metrics in practice: using software metrics to characterize,evaluate, and improve the design of object-oriented systems. Springer
Li W, Shatnawi R (2007) An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J Syst Softw 80(7):1120–1128
Marinescu R (2004) Detection strategies: Metrics-based rules for detecting design flaws. In: 20th international conference on software maintenance (ICSM 2004), 11-17 September 2004 Chicago, IL, USA, 350–359
Meszaros G (2007) XUnit test patterns: refactoring test code. Addison-Wesley
Moha N, Gueheneuc YG, Duchien L, Le Meur AF (2010) Decor: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36
Munro MJ (2005) Product metrics for automatic identification of "bad smell" design problems in java source-code. In: Proceedings of the 11^{th} International Software Metrics Symposium. IEEE Computer Society Press
Neukirchen H, Bisanz M (2007) Utilising code smells to detect quality problems in ttcn-3 test suites. In: Proceedings of the 19th IFIP TC6/WG6.1 international conference, and 7th international conference on Testing of Software and Communicating Systems, TestCom'07/FATES'07. Springer, Berlin, Heidelberg, pp 228–243
Olbrich SM, Cruzes D, Sjberg DIK (2010) Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In: ICSM, pp. 1–10. IEEE Computer Society
Oppenheim AN (1992) Questionnaire design, interviewing and attitude measurement. Pinter Publishers
Palomba F, Bavota G, Penta MD, Oliveto R, Lucia AD, Poshyvanyk D (2013) Detecting bad smells in source code using change history information. 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, pp. 268–278
Peters R, Zaidman A (2012) Evaluating the lifespan of code smells using software repository mining. In: European conference on software maintenance and reengineering, pp. 411–416. IEEE
Pinto LS, Sinha S, Orso A (2012) Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, FSE '12, pp. 33:1–33:11. ACM
Qusef A, Bavota G, Oliveto R, Lucia AD, Binkley D (2011) Scotch: test-to-code traceability using slicing and conceptual coupling In: Proceedings of the 27th IEEE international conference on software maintenance. Williamsburg, VA, USA, pp 63–72
Ratiu D., Ducasse S., Gîrba T, Marinescu R (2004) Using history information to improve design flaws detection In: Proceeding of the 8th european conference on software maintenance and reengineering (CSMR 2004), 24-26 March 2004. IEEE Computer Society, Finland, pp 223–232
Reichhart S., Gîrba T, Ducasse S. (2007) Rule-based assessment of test quality. J Object Technol 6(9):231–251
Ricca F, Di Penta M, Torchiano M, Tonella P, Ceccato M (2007) The role of experience and ability in comprehension tasks supported by UML stereotypes. Proceedings of 29th ICSE. IEEE Computer Society, Minneapolis, pp 375–384

Ricca F., Penta M. D., Torchiano M., Tonella P., Ceccato M. (2010) How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments. IEEE Trans Softw Eng 36:96–118

Simon F., Steinbr F., Lewerentz C. (2001) Metrics based refactoring. Proceedings of 5th European Conference on Software Maintenance and Reengineering. IEEE CS Press, Lisbon, pp 30–38

Tsantalis N., Chatzigeorgiou A. (2009) Identification of move method refactoring opportunities. IEEE Trans Softw Eng 35(3):347–367

Van Deursen A., Moonen L., Bergh A., Kok G. (2001) Refactoring test code. Tech. rep., Amsterdam

Van Rompaey B., Du Bois B., Demeyer S., Rieger M. (2007) On the detection of test smells: A metrics-based approach for general fixture and eager test. IEEE Trans Softw Eng 33(12):800–817

Wohlin C., Runeson P., Host M., Ohlsson M. C., Regnell B., Wesslen A. (2000) Experimentation in Software Engineering - An Introduction. Kluwer

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

Zaidman A., Rompaey B. V., van Deursen A., Demeyer S. (2011) Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empir Softw Eng 16(3):325–364