Strategies for Avoiding Text Fixture Smells during Software Evolution

Michaela Greiler, Andy Zaidman and Arie van Deursen
Delft University of Technology, The Netherlands
{m.s.greiler,a.e.zaidman,arie.vandeursen}@tudelft.nl

Margaret-Anne Storey
University of Victoria, Canada
mstorey@uvic.ca

***Abstract*—An important challenge in creating automated tests is how to design test fixtures, i.e., the setup code that initializes the system under test before actual automated testing can start. Test designers have to choose between different approaches for the setup, trading off maintenance overhead with slow test execution. Over time, test code quality can erode and test smells can develop, such as the occurrence of overly general fixtures, obscure in-line code and dead fields. In this paper, we investigate how fixture-related test smells evolve over time by analyzing several thousand revisions of five open source systems. Our findings indicate that setup management strategies strongly influence the types of test fixture smells that emerge in code, and that several types of fixture smells often emerge at the same time. Based on this information, we recommend important guidelines for setup strategies, and suggest how tool support can be improved to help in both avoiding the emergence of such smells as well as how to refactor code when test smells do appear.**

***Index Terms*—test fixture smells; test evolution; maintenance;**

I. Introduction

Modern software development often includes the use of extensive automated test suites. While automated tests are helpful from a continuous integration and regression testing perspective, they lead to a substantial amount of test code [1]. This test code is often executed frequently, and needs to be maintained and understood. The long term success of automated testing is highly influenced by the maintainability of the test code [2]. To support easier maintainability of a system, test methods should be structured clearly, well named, small in size, and code duplication across test methods should be avoided [2], [3].

One important part of a test is the code that initializes the system under test (SUT), sets up all dependencies and puts the SUT in the right state to fulfill all preconditions needed to exercise the test. In line with Meszaros, we refer to this part of a test as the *test fixture* [2]. Developers have several options for structuring their test fixture code. The most straightforward option is to place the setup code directly in the test method, which we refer to as an *in-line setup*. An alternative approach is to place the setup code in helper methods that can be called by several test methods, the so-called *delegate setup* [2].

Today's testing frameworks, such as the widely used xUnit family, have dedicated mechanisms for managing setup code invocations [4], [5]. Therefore, helper-methods comprising the setup code can be marked (e.g., using annotations or naming conventions) as specific setup methods, and the test framework takes care of invoking them at a specific point in time. We refer to this as an *implicit setup* as the invocation happens implicitly.[1]

Developers must decide how to set up test fixtures and adjust their *fixture strategies* during the evolution of test code. Otherwise, they end up with poor solutions to recurring implementation and design problems in their test code, also known as *test smells* [6]. To support developers during the analysis and adjustment of test fixtures, we previously developed a tool called *TestHound*[2] to automatically detect test fixture smells and guide test code refactoring [7].

An evaluation of TestHound showed that developers are concerned about test fixture smells, and that TestHound helps them to discover and address those smells. But, we also learned that resolving these smells after they have been in the code for a long time can be problematic. Developers would benefit greatly from having the tool available in the continuous integration environment so that they are made aware of changes in the densities of test smells immediately.

In this paper, we investigate the evolution of test fixture smells and which software changes lead to increased test smell densities to determine the best time to alert developers about smell changes. We look at when and how test fixture smells are introduced, and what role the setup strategies play in smell evolution. Our contributions in this paper are:

1) a technique for analyzing multiple revisions of a software system for *fixture-related* test smells, and examining trends in smell evolution;
2) an implementation of this technique in a tool called *TestEvoHound*, which mines Git and SVN repositories for test fixture smells;
3) insights in test fixture smell evolution in real-world situations based on an investigation of five well-known open source systems;
4) strategies for avoiding test fixture smells.

Our investigation shows that fixture management strategies strongly influence fixture smell evolution. Also, we find that fixture smells remain stable over long periods of time, until certain code changes cause drastic changes in smell densities. Making developers aware of these changes can help prevent the introduction of test smells with only small adjustments to the test code in a continuous and incremental fashion. Further,

[1] For example, in the JUnit framework, methods can be named *setUp()* or marked with annotations such as *@Before* or *@BeforeClass*.

[2] http://swerl.tudelft.nl/bin/view/MichaelaGreiler/TestHound

we show that classes with a larger number of test methods have more test fixture smells than classes with fewer test methods, and thus recommend that classes with many test methods be avoided or refactored.

Section II briefly summarizes different test fixture smells. Section III details the experimental design used to investigate the evolution trends of test fixture smells. Section IV outlines the measurements implemented in the *TestEvoHound* tool. Section V details the results of our investigation, followed by a discussion of the findings in Section VI. In Section VII, we present related work and conclude in Section VIII.

II. Test Smells

The *code smells* metaphor was first introduced by Fowler, who describes a code smell as a poor solution to a recurring implementation and design problem [8]. Code smells are not a problem per se, but they may lead to issues such as understanding difficulties, inefficient tests and poor maintainability of a software system. Later, van Deursen et al. introduced the term *test smells* by applying the smell metaphor to test code [6]. Since then, their initial set of test smells has been extended [2], [9], [10]. In [7], we enhanced these test smells with additional *fixture-related* smells, derived metrics to aid in their detection, and implemented a technique to automatically detect test fixture smells in a tool called TestHound.

In this paper, we investigate the evolution of these test fixture smells, which are summarized below, in order to better understand how they can be avoided and how tool support would be most beneficial for developers. We refer the reader to [7] for more details on the *test fixture smells*.

General Fixture Smell. The general fixture smell occurs if test classes contain broad functionality in the *implicit* setup, and if several tests only access part of the fixture. Problems caused by a general fixture are two-fold. Firstly, the cause-effect relationship between fixture and the expected test outcome is less visible, and tests are harder to read and understand. This can lead to fragile tests: a change that should be unrelated affects tests because too much functionality is covered in the fixture. Secondly, test performance can deteriorate and long test execution times may eventually cause developers to stop running tests altogether. We identify a test method as a general fixture method when it uses less than 70% of the fields initialized in the setup.

Test Maverick Smell. A test method is a maverick when the class providing the test method contains an *implicit* setup, but the test method is completely independent from the implicit setup procedure. The implicit setup is executed before the test method, but it is not needed. In such cases, understanding the cause-effect relationship between setup and test method can be hampered because discovering that test methods are unrelated from the implicit setup can be time consuming. We identify a test method as a maverick when it does not use any of the setup fields initialized.

Dead Fields Smell. The dead field smell occurs when a class or its super classes have fields that are never used by any of the test methods. Often, dead fields are inherited from a super class. This can indicate a suboptimal inheritance structure, or that the super class conflicts with the single responsibility principle [11]. Also, dead fields within the test class itself can indicate incomplete or deprecated development activities. We identify dead fields as all fields that are initialized by the *implicit setup*, but never used by any of the test methods.

Lack of Cohesion of Test Methods Smell. Cohesion of a class indicates how strongly related and focused the various responsibilities of a class are [12]. Low cohesive classes are smelly because they negatively affect code reuse, maintainability and comprehension [8], [13]. The smell Lack of Cohesion of Test Methods (LCOTM) occurs if test methods are grouped together in one test class without being cohesive. To measure LCOTM, we adjusted the Henderson-Seller Lack of Cohesion of Method metric [14]. Differing from the original metric, we focus on the cohesion between test methods in a class.

Obscure In-Line Setup Smell. Meszaros introduced the smell obscure test to refer to a test that is difficult to understand [2], and thus, unsuitable for documentation purposes. From this smell we derived the obscure in-line setup smell. An *in-line* setup should consist of only the steps and variables essential to understanding the test; necessary but irrelevant steps should be encapsulated into helper methods. An obscure in-line setup covers too much setup functionality within the test method, and this can prevent one from seeing the test's relevant verification steps. We measure the obscurity of an in-line setup based on the number of local variables directly defined in a test method.

Vague Header Setup Smell. A vague header setup smell occurs when fields are solely initialized in the header of a class. We consider this a smell as the behavior of the code is not explicitly defined and depends on the field modifier (static or member) as well as on the implementation of the test framework. Vague header setups might hamper code comprehension and maintainability, as fields can be placed anywhere in the class. Further, in many test frameworks exception messages are more expressive for fields initialized in the setup. We report a vague header smell when at least one field is solely initialized in the header of a class.

III. Experimental Setting

To understand test fixture smell evolution, we used case study research and investigated five research questions within five subject systems, as detailed below.

A. Research Questions

Our research questions focused on the evolution of test fixtures and test fixture smells.

RQ1: Do test fixture strategies change over time?
RQ2: Do test fixture smell densities increase over time?
RQ3: How are test fixture smells spread throughout a project?
RQ4: Which changes cause alterations in fixture smell trends?
RQ5: Are test fixture smells resolved?

We investigated different test fixture strategies and the changes made to test fixtures over time to get a general understanding of the characteristics of the systems under investigation (RQ1).

.

TABLE I
FIXTURE MANAGEMENT STRATEGIES

Project	KLOCs	# Test classes	# Test cases	No setup	No. revisions	Period in life cycle of project	Date
Voldemort JUnit	130	132	520	71	2900	start → end	04/2011-10/2012
PMD	174	118	739	102	1900	mid → end	09/2007-11/2012
Checkstyle	66	156	549	131	2251	start → end	06/2001-10/2010
Jsoup	20	23	372	23	973	start → end	12/2009-09/2012
Java Azure SDK	39	30	358	14	300	start → end	10/2011-10/2012

To understand whether test fixture smell densities increase during the life of a project, we looked at test fixture smell trends (RQ2). We also looked at fixture smell dispersion to understand how smells spread throughout a software system (RQ3). We investigated what causes test fixture smells to change statistically and by manual investigation of severe changes (fluctuations) in smell trends (RQ4). To see whether smells are resolved by the developers (RQ5), we followed smelly classes throughout their evolution.

B. Case Studies

To investigate the evolution of test fixture smells, we selected five well-known, Java based, open source projects that have test suites and automatic build files. We display important characteristics of the latest analyzed revision of these projects (i.e., size, analysis duration, and number of classes comprising setup methods) in Table I.

Checkstyle is a tool to help programmers write Java code that adheres to a coding standard. We analyzed 2252 Checkstyle revisions. The latest revision contained 549 test methods.

PMD is a tool to analyze Java source code for potential problems such as dead or duplicated code. We analyzed 1900 PMD revisions. The latest revision contained 739 test methods.

Jsoup is a Java HTML parser that provides an API to extract and manipulate data using DOM, CSS and JQuery-like methods. We analyzed all 973 revisions of Jsoup, and the latest revision contained 372 test methods.

Azure Java SDK provides the Azure (i.e., Microsoft Cloud Platform) libraries for Java. We analyzed all 300 revisions of Azure. The latest revision contained 358 test methods.

Voldemort is a distributed key-value storage system. We analyzed all 2900 revisions of Voldemort. The latest revision contained 520 test methods.

IV. ANALYSIS OF FIXTURE SMELL EVOLUTION

In this section, we introduce *TestEvoHound*, a tool we developed to automatically analyze the evolution of test fixtures and test fixture smells over multiple revisions of a software system. We also detail the measurements taken in order to answer our research questions.

A. TestEvoHound

We developed *TestEvoHound* to analyze fixture smell evolution. *TestEvoHound* works with Git and SVN repositories, and is available for download.[3] When analyzing code, *TestEvoHound* executes four tasks.

During the *Revision Checkout* task, *TestEvoHound* checks out each revision of the project under analysis, and for each revision it starts the *Build Process* task. Here, the tool searches for ANT or MAVEN build files, initiates the build process and compiles the source code (including tests). Then, the *Test Fixture Smell Analysis* task invokes the TestHound tool to analyze the current revision for smells and stores the outcome. Finally, when all revisions have been analyzed, the *Trend Analysis* occurs. Here, the *TestEvoHound* tool calculates the trends and measurements among all revisions, as described in the next subsection. This information is stored in comma-separated value format to allow easy visualization by tools like Excel or R.[4]

B. Measurements to Answer the Research Questions

RQ1. To see whether test fixture management changes over time, we looked at the presence and absence of the *implicit* setup mechanisms provided by the test frameworks. This means that, for each project and over all revisions, we analyzed how many of all test classes contained an *implicit* setup and whether this changed over time. We also analyzed how many fields were declared in a test class, as fields are also a way to create a state accessible by all test methods of a test class.

RQ2. To answer whether test fixture smell density increases over time, we measured the occurrence of the six different fixture smells for each revision, and charted whether the ratio between smelly entities to all entities changed. We looked at this ratio because all of the systems under study increased in size over time. Thus, the number of smelly entities has to be considered in relation to the overall number of entities. This ratio demonstrates whether the quantity of smells is rising or falling. In the remainder of this paper, we refer to a series of ratios as the trend of smell evolution.

For the general fixture, the test maverick, and the obscure in-line setup smells, we compared the number of test methods affected by these smells with the overall number of test methods in the code base. We calculated the trend in dead field evolution by comparing the number of dead fields with the overall number of fields. For the LCOTM and vague header smells, we compared the number of test classes affected with the overall number of test classes.

RQ3. To better understand whether test fixture smells are widely spread among all test classes, or whether certain test classes are more prone to fixture smells, we investigated the dispersion of fixture smells. To do this, we looked at histograms of the test fixture smells.

RQ4. To understand which software changes cause test fixture smell densities to change, we statistically tested whether the number of test methods or fields within a class correlates with the smell density of a class. Then, we manually investigated the code base and the commit logs for periods where severe fluctuations in the trends of test fixture smells (increase or decrease) occurred.

[3] http://swerl.tudelft.nl/bin/view/MichaelaGreiler/TestEvoHound

[4] http://www.r-project.org/

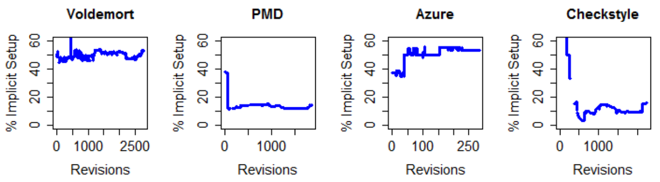

Fig. 1. Implicit Setup Trends

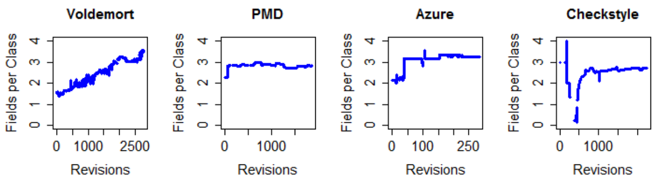

Fig. 2. Field Trends

For the statistical tests, we used Spearman correlations provided by the statistical program *R* (package *hmisc*) to investigate the following two hypotheses:

H1 The more test methods placed within a test class, the higher the smell density of a test class is.

H2 The more fields placed within a test class, the higher the smell density of a test class is.

RQ5. By tracking smelly classes over time and investigating decreases in test fixture smells, we see whether and how test fixture smells are resolved.

V. Investigation of Test Fixture Smell Evolution

In this section, we detail our findings from the investigation of the five research questions. Each subsection covers one of the research questions, with the exception of RQ4, to which we dedicate two subsections (Sections V-D and V-E).

A. *Evolution of Test Fixtures*

Our analysis shows that the test fixture strategies used across the projects differ greatly. In Jsoup, the developers completely refrained from using the *implicit* setup mechanism available in JUnit. In Checkstyle and PMD, approximately 10% and 13% (respectively) of the test classes comprise an implicit setup. In Voldemort and Azure, approximately 50% of the classes have an *implict* setup. Just as the setup management styles differ across all five projects, the fixture smells also differ, as illustrated in Fig. 3.

Setup Trends. To answer whether the setup strategies change over time (RQ1), we looked at the trends for the presence of implicit setups. As can be seen in Fig. 1, the usage pattern for the implicit setup changed. For example, in Azure, the density of setups per class increased over time, and in PMD, it decreased. On the other hand, apart from some severe fluctuations in density, the ratio stays quite stable for all four projects. We excluded Jsoup because we did not encounter a single implicit setup.

Setup Fluctuations. To understand the severe setup fluctuations, we manually investigated the changes between revisions that caused the fluctuations. In Checkstyle, the drop in *implicit* setup usage at the beginning is because the code base started with three test cases, each containing an *implicit* setup (i.e., 100%). Over time, tests without an *implicit* setup were added, causing the percentage to drop to 10%.

In PMD, the percentage of test classes that have an *implicit* setup drops at the beginning. This is due to a simple structural change where 38 test classes were removed by changing how methods were invoked. Before the change, many of these test classes contained only one test method that invoked another method. After the change, this functionality is covered by the parent test class within the setup, making 38 test classes obsolete. The commit log states: *"code refactoring: testAll() moved to parent, rules are now added in setUp() using addRule()"*, confirming our observation.

In Azure, the increase of implicit setups is due to the addition of six test classes, all of which have an implicit setup. The commit log states: *"table tests" have been added.*

Fields. As the fields of a class are a way to create a state accessible by all methods of a test class, we also analyzed how many fields are declared within each test class over time. In Checkstyle and PMD, even though only a small percentage of the classes contain a setup, they still have a similar number of class fields as compared with Azure and Voldemort (2.5 to 3 fields per test class). The only exception is Jsoup, which has only two class fields within all test classes. The trend of fields per test class, as illustrated in Fig. 2, reveals that in Voldemort, the number of fields per test class continually increases, which is an indication that the test classes become more complex. In the other three systems, there are distinct events where the number of fields fluctuates and then stays quite stable over time. For PMD, the 38 deleted test classes were quite simple and did not comprise fields, so their deletion led to an increased ratio of fields per test class. In Azure, the added test classes ("table tests") also led to an increase in fields per class, which is indicative of an increase in complexity.

B. *Discovery of Test Fixture Smell Trends*

Lehman's law of increasing complexity states that when a system evolves, its complexity increases unless work is done to maintain or reduce it [15]. Therefore, we expect that due to the increasing complexity of a given system – the test fixture grows, and more test methods are placed within the same test class – the potential for test fixture smells (such as the general fixture, LCOTM, test mavericks and dead fields) increases.

Over the investigated periods of all five projects, the number of test classes and test methods increased. On the other hand,

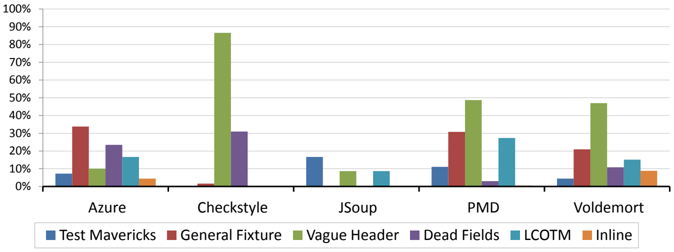

Fig. 3. Test Smell Density Among Projects

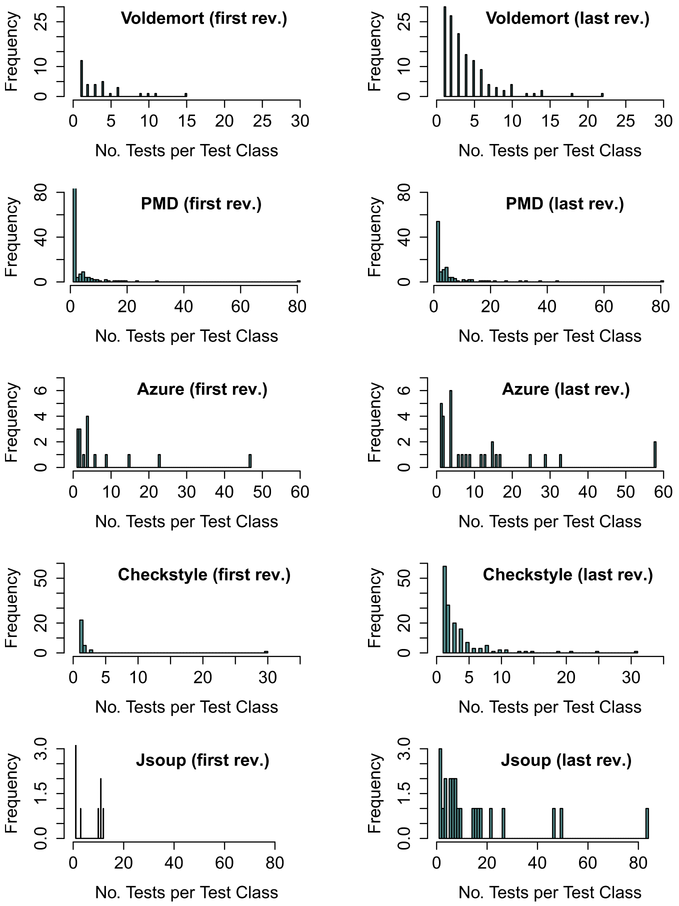

Fig. 4. Histograms of Tests per Class for first and last Analyzable Revisions

the ratio of test methods per test class stayed quite stable for PMD and Voldermort. In Azure and Checkstyle, the number of tests per class increased slightly over time, and in JSoup we observed the strongest increase. In Checkstyle, a peak occurred at the beginning because the test efforts started within a single class that comprised up to 24 test methods, and new test classes were introduced slowly. Even though the mean value for tests per class seems small and stable across all systems, the inspection of the dispersion of tests per class (see Fig. 4) shows, some test classes contain a significantly larger number of test methods (30, 40 or even 80) than other test classes. Over time, this imbalance grows.

Interestingly, the experiments revealed that even though test code becomes more complex (more tests and more fields per test class), a general growth of test fixture smells over time does not occur. As depicted in Fig. 5, the test fixture smell trends often stay stable over time, and fluctuate greatly only at a distinct point in time. Continual increases or decreases are exceptional, as in the case of Voldemort. In the following sections, we take a closer look at how smells dispersed in a system, and what causes the fluctuations in the smell trends.

C. Dispersion of Test Fixture Smells

Over the course of several revisions, we investigated how frequently smelly entities occur per test class. We used histograms to visualize whether all classes contain the same number of fixture smells or whether some classes are more prone to test smells.

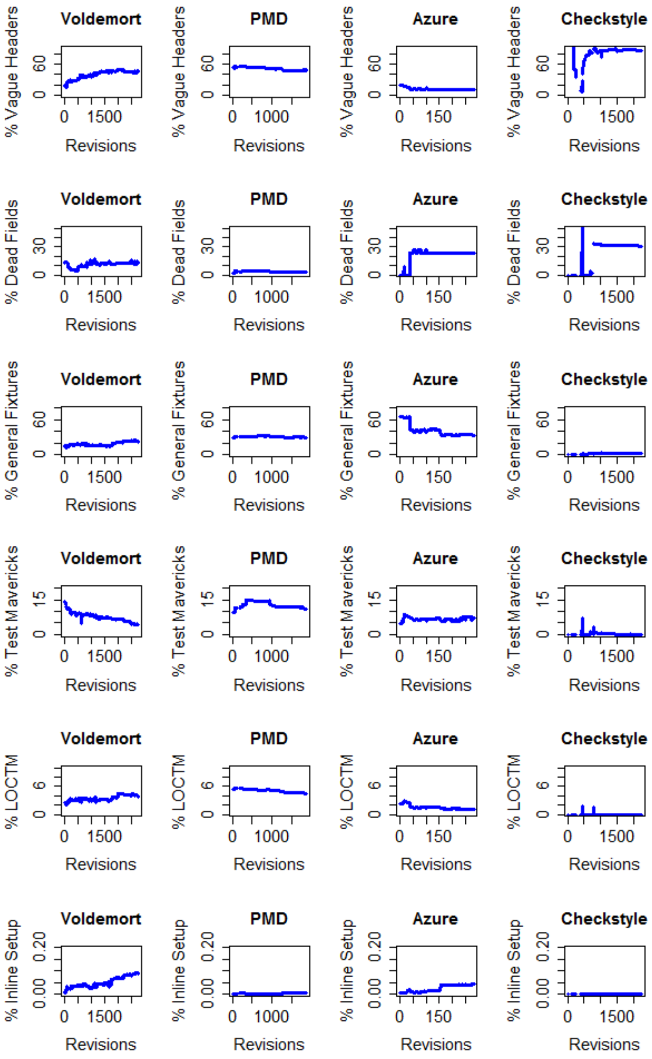

Fig. 5. All Smell Fixture Trends

For general fixtures, test mavericks and obscure in-line setups, smells cluster in a few classes. To illustrate the outcome, this paper shows the dispersion of the general fixture smell for the first, middle and last analyzable revisions of each project in Fig. 6. For all projects, some classes contribute a disproportionately large number of general fixture methods, and over time, these classes accumulate more general fixtures. In the last revision of PMD, three test classes contributed 45% of all general fixture methods (i.e., 2.5% of the classes), and 5.1% of the test classes contributed 60.1% of the general fixtures. In the first analyzed revision, three classes contributed 40% of the smells. A similar effect can also be observed for Azure, Checkstyle and Voldemort for the general fixture and test maverick smells. Jsoup does not contain general fixture methods, but it has a strong tendency for test mavericks to cluster. We can also observe that over time, the relative number of test smells per class increases (i.e., the ratio of smelly entities to entities per class).

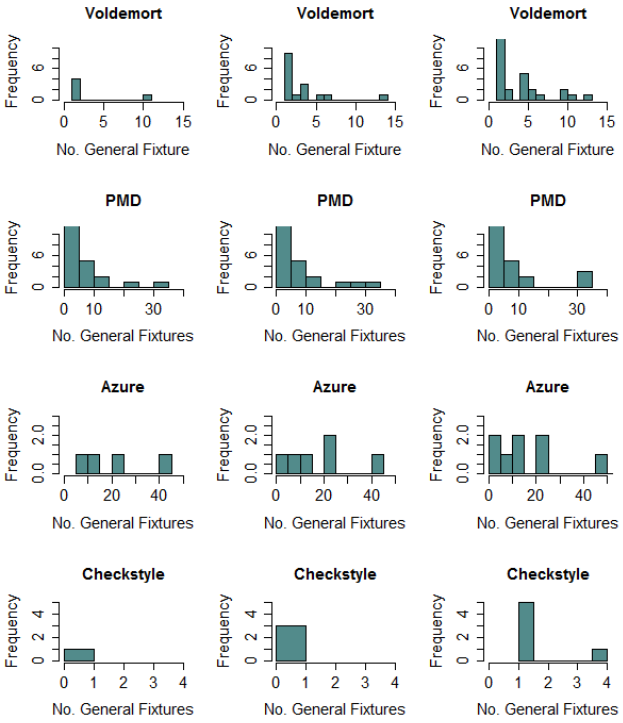

Fig. 6. Dispersion of General Fixtures for first, middle and last Analyzable Revisions

On the other hand, dead fields did not cluster. As vague headers or a LCOTM are either present or absent in a class, their distribution is not of interest.

Summary. Since some classes are more prone to test smells, such as general fixture, test maverick and obscure in-line setup, refactoring activities can be directed to these classes. This shows that one can reduce the majority of test fixture smells by cleaning a few smelly classes.

D. Development of Test Fixture Smells

The smell trends do not show a general continual increase of smell densities over time as expected, but considering the smell distribution, there is a tendency for some classes to accumulate more smells (absolute and relative) over time than others. We also see that some classes comprise an unproportionally large number of test methods. The more test methods contained within a single class, the more diverse the requirements for the test fixtures can become. Because of this, we expect that test classes with a higher number of test methods have a higher smell density than their smaller counterparts. Also, we expect that test classes with more fields have a higher smell density than test classes with fewer fields.

To investigate these hypotheses, we used statistics and correlated the number of test methods, or fields per class, with the percentage of test methods affected by a smell. We excluded Jsoup from this experiment as the project has few smells and few fields. The results are given in Table II. We used the thresholds defined by Hopkins [16], where a correlation is considered moderate when the value is higher than 0.3 (or lower than -0.3), and strong when the value is higher than 0.5 (or lower than -0.5). For a correlation to be significant, the p-value of the test needs to be below 0.01. This means the chance that the correlation is due to random chance is less than 1 percent.

Data Points. We did not use each data point (i.e., all classes for each revision) because the two variables correlated (i.e., test methods or fields, and a particular smell factor) might be stable over time. This can cause certain combinations (observations) to seem more likely than others. Therefore, we reduced the dataset to include only unique combinations. For example, in case test class "TestA" has three test methods and one general fixture method over a period of 10 revisions, we include this incident only once in our data set, whereby we look for uniqueness considering more than 21 different characteristics of a class.[5] Further, as a class without a field can not have a general fixture, LCOTM, test maverick, dead field or vague header smell, we excluded classes without a field. For LCOTM, we also only considered data points with a valid LCOTM value (i.e., LCOTM is undefined when a class has a single method). Therefore, the results for LCOTM detail the number of data points separately.

Test Methods. For three of the four projects, the results indicate that for test smells such as general fixture, LCOTM, test maverick and obscure in-line setup, *a higher number of test methods per class* correlates with a higher density of these smells. For Checkstyle, the correlation cannot be shown as very few test smells exist. We also investigated whether more dead fields exist in classes with more test methods. This correlation does not hold for Checkstyle and PMD. For Azure and Voldemort, the results show a weak negative correlation, indicating that with a higher number of test methods per class, the dead fields density decreases.

Fields. The second characteristic we expect to find related to the smell density of a class is the number of fields per class.

We observe a correlation between *smell density and the number of fields* for the general fixture and LCOTM smells within the PMD and Azure systems, as listed in Table II. A weak correlation also exists for a higher obscure in-line setup density and fields in PMD and Voldemort. Azure is the only other system which shows a correlation between a higher test maverick density and fields, and a weak negative correlation for dead fields (i.e., more fields are correlated to a lower dead field density).

We investigated why an increased number of fields also increases fixture smell density for the PMD and Azure systems, and saw that this has to do with the style of fixture management. For the majority of new test methods, developers introduce a new dedicated field, and often there is one field for each test method. This means that if new test methods are added to a class, it is very likely that all of them are general fixture methods, and the lack of cohesion between the test methods increases further. With some test classes in PMD, either almost all test methods are general fixtures, or they do not have anything to do with the fields of the class, making

[5] Detailed information on the attributes covered for each class is outlined at http://swerl.tudelft.nl/bin/view/MichaelaGreiler/TestEvoHound

TABLE II
STATISTICAL CORRELATIONS BETWEEN 1) NO. OF TESTS PER CLASS OR 2) NO. OF FIELDS PER CLASS AND SMELL DENSITY

H	Project	N	GF		TM		OI		DF		LCOTM		
			Cor	p	Cor	p	Cor	p	Cor	p	Cor	p	N
Tests per Class	Voldemort	705	**0.63**	0	**0.34**	0	0.06	0.09	**-0.41**	0	**0.60**	0	687
	PMD	268	**0.51**	0	**0.56**	0	**0.37**	0	0.05	0.40	**0.67**	0	263
	Checkstyle	437	0.11	0.03	-0.09	0.06	Na	Na	-0.14	<0.001	0.11	0.02	430
	Azure	118	**0.51**	0	**0.40**	0	**0.48**	0	**-0.46**	0	**0.48**	0	118
Fields per Class	Voldemort	705	0.23	0	-0.15	0	**0.32**	0	0.24	0	0.14	<0.001	687
	PMD	268	**0.72**	0	0.23	<0.001	**0.38**	0	0.03	0.67	**0.55**	0	263
	Checkstyle	437	-0.06	0.24	-0.04	0.43	Na	Na	0.22	0	-0.02	0.74	430
	Azure	118	**0.58**	0	**0.64**	0	0.25	0.005	**-0.40**	0	**0.57**	0	118

them test mavericks. For example, in the "JDKVersionTest", all 33 test methods are general fixture methods, whereby each method uses very few of the 37 fields declared. The LCOTM value in this class is as high as 0.99 (with 1 representing a completely noncohesive class).

E. Fluctuations in Test Fixture Smells

As is apparent from Fig. 5, fixture smell trends tend to change drastically at certain points in time. We manually investigated code base changes that occurred during the periods of severe smell trend changes to understand "Which changes cause alterations in fixture smell trends" (RQ4).

Vague Header Smell. A drastic change in the vague header smell is seen in Checkstyle alone. Checkstyle started out with a single test class that contains a vague header (i.e., 100%). In the next revisions, several tests without vague headers were added, until revision 433, where the smell continually increases due to changes in the inheritance structure and refactoring activities.

Dead Field Smell. For the dead field smell, we analyzed the drastic changes in Azure and Checkstyle. The main test fixture smell in *Checkstyle* is the *dead field* smell, which drastically increases around revision 760-769 (as illustrated by Fig. 7). The investigation of the changes between these revisions revealed that all additional dead fields are inherited from a single base class that is extended by almost all test classes (BaseCheckTestCase). In this test class, a helper method was introduced, and one field (which was no longer needed) was forgotten and remained in the class as legacy. In the commit log, the developer notes: *"Added a helper method to create a configuration for a check..."*. This dead field remained for the duration of our investigation. With *Azure*, a strong increase at revision 39 is also due to dead fields inherited from one super class, which is extended by six test classes.

General Fixture Smell. The main fluctuation in general fixture smell trends can be seen in *Azure*. For Azure, this is also the predominant smell. The trend, displayed in Fig. 8, shows two distinct decreases: at revision 40 (from over 60% to less than 42%) and at revision 150. In revision 40, a drastic decrease occurred because new test classes providing 87 tests (testing the Table client) were added. The commit log shows *"Table Client commit [...]"*. With all these added tests, only two new general fixture methods were introduced. Also at revision 150, 3 new test classes and 76 tests were added, most of them without general fixture methods. In both cases, none of the already existing general fixtures have been resolved.

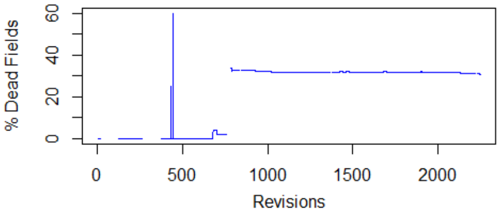

Fig. 7. Dead Field Density in Checkstyle

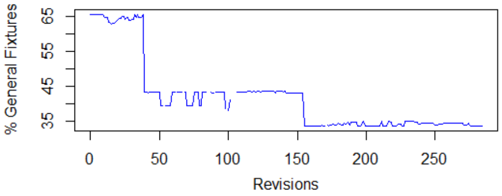

Fig. 8. General Fixture Trend Azure

Test Maverick Smell. There are stronger fluctuations in the test maverick density in PMD and Jsoup, as well as a drastic continual decrease of the test maverick density in Voldemort.

In *PMD*, 12-14% of the methods are test mavericks, and the smell density increases and then decreases again, as illustrated in Fig. 9. The first increases are due to the aforementioned removal of several test classes and test methods, and the introduction of vague headers in classes. The manual investigation of the strong decrease around revision 900 reveals that the Java version was changed during this period. The commit log contains the following comment: *"Remove support for Java 1.4 runtime. [...] changes are made to code which made checks for running on a 1.4 JVM (found via searches)."* This change causes a drop in the number of test mavericks as the *implicit setup* of two test classes, used only to set the Java version, was removed.

In Jsoup, test mavericks are caused by two classes that contain a vague header setup.[6] The fluctuations (illustrated in Fig. 10) occurred when the two classes were introduced to the code base (revisions 20 and 80). Over time, more test methods were added to these classes, which make no use of the fields of the classes. Depending on whether more non-smelly tests are added, or more tests are added to these two classes, the smell

[6] `ElementTest` and `UrlConnectTest`

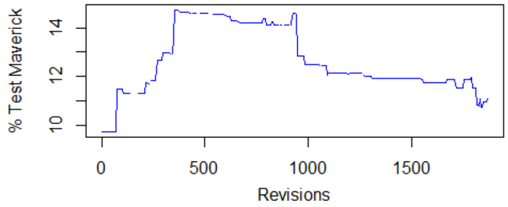

Fig. 9. Test Maverick Trend in PMD

density increases or decreases. In the latest revision, these two classes cause over 60 test maverick methods.

The test maverick density in *Voldemort* steadily decreases. This is because fewer test mavericks are introduced over time, whereby the number of test methods increases (as illustrated in Fig. 11). The test maverick trend increases visible in the graph are due to the introduction of a field (via a vague header) which is only used by a few of the methods, and to a new test class which has a setup used by only a few test methods. The decreases are because new fields are added to classes with *implicit setups*, and previous test mavericks now use the fields.

LCOTM Smell. The LCOTM smell trends do not increase over time. In PMD, the relative number of test classes suffering from the LCOTM smell even decreases. On the other hand, when inspecting the smelly classes, we observe that their LCOTM value does not decrease, but it either stays stable or increases. In Azure, the more severe fluctuations in LCOTM trends are because of the drastic increase of non-smelly classes.

Obscure In-line Setup Smell. In Azure, the number of obscure in-line methods increases (around revision 150). The majority of smelly methods come from two new test classes (“CloudQueueTests” and “CloudBlobContainerTests”). The other obscure in-line setups were added to an already existing class (“TableClientTests”). In Voldemort, there is a continuous increase in the obscure in-line setup smell, where a few classes, such as “RoutedStoreTest” and “AbstractRebalanceTest”, accumulate more smells over time. Interestingly, in Jsoup, even though the test setup was placed within test methods (or helper methods), not a single test method suffers from the obscure in-line setup smell.

F. Test Fixture Smell Resolution

Previous research on test smells suggests that test smells do not get resolved unless the test class or methods are deleted [17]. We investigated whether this phenomenon can also be observed in our subject systems for *test fixture smells*.

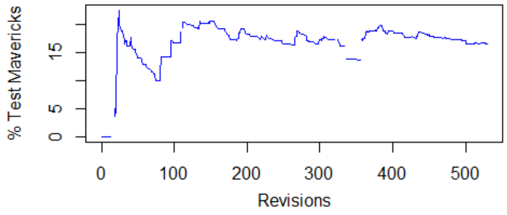

Fig. 10. Test Maverick Smell Trend in Jsoup

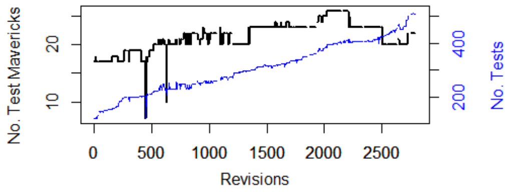

Fig. 11. Test Maverick Smell in Voldemort

In general, we did not observe a major reduction of test smells. Several times, test smells decreased because new non-smelly test methods were added, thus reducing the percentage of smelly methods, as was seen with Azure. Other times, changes to the code base caused the smells to be reduced. For example, changing the Java version resolved many of PMD’s test mavericks. In the majority of cases, we saw that fixture smells are either not resolved, or are only resolved by deletion. Here, we will summarize the exceptional cases of smell resolution without deletion.

In Checkstyle, during a short period (revisions 442-444; also visible in Fig. 7), an overly general inheritance structure causes several dead inherited fields and test mavericks. The log states: *“Refactored the tests to only use the Checker interface”* and *“Refactoring the way the Checker is configured. Not happy with the current approach - it was hack”*. This change addresses the problem of the dead fields and the test mavericks, and shows that the developers made an effort to resolve the smells by changing the inheritance structure and using the functionality of the super class. Another time, test mavericks appear for a short time while the helper method in “BaseCheckTestCase” is implemented.

In Voldemort, a large number of test mavericks are resolved over time (visible in the continual decrease in the test mavericks trend) because new fields, which are then used by test methods, were added to existing classes (also visible in Fig. 2). For example, this occurs in test “ConsistentRoutingStrategyTest”.

In PMD, we see that some of the obscure in-line setups are resolved. For example, in the test class “RuleSetFactoryTest”, an “*Extract Method*” refactoring [8] was performed, which resolved two in-line smells. In Voldemort, in-line setups are also resolved from time to time. On the other hand, in Azure, none of the obscure in-line methods are resolved.

General fixture, LCOTM and vague header smells are seldomly resolved in all systems. Exceptions are cases where the final functionality of a test is not yet implemented. For example, in Voldemort, developers added a test method that comprised only a few statements and a “todo” comment. After some time, the method was fully implemented and the general fixture resolved.

VI. Discussion

A. Findings

Our study revealed many interesting findings about fixture management strategies and their influence on test smells, as well as how fixture strategies and smells evolve during the lifetime of several open source projects. In summary, our investigation of test fixture smells showed that:

- the style of fixture management varies greatly between projects;
- the projects' test suites also suffer from different patterns of test fixture smells;
- the number of test methods per test class correlates with the density of test fixture smells;
- more fields in a class do not necessarily correlate with a higher test fixture smell density;
- drastic increases in test smells are often caused by structural changes (such as refactorings with forgotten legacy functionality);
- once introduced, test smells tend to stick around and do not get resolved;
- most likely, fixture smells disappear because the test class or test method is deleted.

B. Implications for Automated Test Fixture Smell Detection

As we saw from a previous study [7], developers are concerned with the quality of their test code and see test fixture smells as potential problems. On the other hand, they are under pressure to develop production code, not to improve test code. The developers indicated that in terms of improving test code, they look for "low hanging fruit", i.e., easy changes that do not involve major refactoring. The developers also indicated that they want to be notified when smells are introduced. We believe that the findings of this study can be used to design smell alerting mechanisms for developers evolving test code.

We saw that the test fixture management strategy greatly varies between projects, and that projects stay with the style they started with. For project leads, this means that they have to make conscious decisions at the beginning, and know which test fixture smells are more likely to occur as a consequence of the chosen fixture management strategy.

Also, even though some classes become smellier over time, we saw that test smell trends do not continually increase. This suggests that developers eventually learn the right strategies for writing test cases for a given system, and that new classes tend to be less smelly than old ones, which is in line with the information developers gave in the interviews [7].

Further, we also saw that simple structural changes in test classes can lead to both a drastic increase as well as a decrease in test fixture smells. First, because small changes can have a major impact on smell development, we believe getting developers' attention when they are about to introduce test smells is an important first step to smell avoidance. Second, as test smells tend to accumulate in a few classes, refactoring these classes can greatly reduce test fixture smells.

Further, as the number of test methods per class impacts the smelliness of test code, developers should reconsider the current practice of grouping test methods within a class. Alerting developers about the lack of cohesion of test methods might be a first step in reducing class size.

C. Strategies and Recommendations

Based on the observations and findings of this study, we defined the following strategies and recommendations for smell avoidance:

- Keep test classes small (and therefore coherent), by reducing the number of test methods within a class.
- Keep inheritance structures flat, and also limit the scope of super classes (e.g., do not implement one super test class that provides functionality for all base test classes).
- Use composition instead of inheritance to provide test classes with helper functionality.
- Create data classes that provide test inputs to avoid overloading test classes with field declarations that are only used for a single test method. This improves performance and understandability of test classes.
- Be aware that declaring fields in the header may impact tests in terms of understandability and test performance.
- Reconsider the "one test class per class" organization in case test methods within a single test class require diverse states and configurations for the system under test.
- Carefully consider the consequences of the chosen fixture strategies and the impact those decisions will have on the projects, as this is not likely to change.

D. Threats to Validity

In terms of generalizability, our current implementation only works for Java-based systems that use JUnit or TestNG test frameworks and have automatic build files (i.e., Maven or ANT) available. On the other hand, it should be easy to adjust our tool's build process so that developers using other means to build systems can analyze their history. Further, we believe that this technique is not only easily transferable to other xUnit testing frameworks, but also to other languages. Also, our evaluation was limited to five software systems. However, we chose systems that are well known, publicly accessible and actively developed.

With respect to internal validity, the analysis may be incomplete or contain bugs. To conquer this threat, we implemented multiple test cases. Also, the extensive manual inspection of the systems under study confirmed our confidence in the correctness and precision of the analysis. For future work, we plan to assess the accuracy of the results in additional case studies.

We also had to make some changes to the systems under study. For *Checkstyle*, a wide range of revisions had a non compilable test class checked in[7]. To be able to compile and analyze the rest of the tests, we deleted this class in case it failed to compile. For several revisions in *PMD*, the Maven build file did not link to the correct dependencies. So, we updated the dependency information in order to be able to analyze a wide range of revisions.

VII. Related Work

As discussed in Section II, test smells have been studied previously, with some research efforts focusing on the auto-

[7] `LocalizedMessageTest.java`.

matic detection of test smells. Among them, Van Rompaey *et al.* tried to detect *General fixture* and *Eager test* test smells by means of metrics [18]. Subsequently, they described a tool which used well-known software metrics to predict a broader variety of potential problems and test smells [19]. Our study of test smells differs in several ways. First, we focus on test fixture management and analyze test code for specific fixture problems relevant in practice. We also provide concrete refactoring suggestions for developers. In contrast to our work, Borg *et al.* described automated refactoring for acceptance tests based on the FIT framework [20]. To the best of our knowledge, fixture-related test smells and refactoring have not been studied so far.

Co-evolution of software and test code has been investigated by Zaidman *et al.* in [1]. Pinto *et al.* investigated the evolution of test code in order to better understand how test repair tooling can assist developers during test maintenance [21].

Gälli *et al.* presented a taxonomy of (Smalltalk) unit tests in which they distinguish tests based on, for example, the number of test methods per method under test, and whether or not exceptions are taken into account [22].

In general, *code and design smells* have been researched in previous work. For example, Moha *et al.* outline a method called DECOR and its implementation to detect several code and design smells and evaluate their technique in several case studies [23]. Lanza and Marinescu use metrics to identify classes that might have design flaws [24], [25]. The metrics and smells presented in this paper address properties of code exclusively present in test code, i.e., the creation and tear down of the test fixtures.

VIII. Conclusion

In this paper, we investigated the evolution of test fixture smells so that we could discover the most beneficial time to alert developers about smell changes, and to learn which software changes lead to test smell increases. Therefore, we investigated when and how test fixture smells are introduced, and which roles the setup strategies play. Our findings indicate that test fixture smells do not continually increase over time, even though system complexity increases. There is a correlation between the number of tests within a class and smell density. An important insight is the clustering effect of test smells; the few classes that contribute the majority of test smells are the classes developers should be made aware of.

Our contributions in this paper are:

1) an implementation of a tool which supports the mining of repositories;
2) a technique (and implementation) to analyze several revisions of a software system for fixture-related test smells, and to understand the trends in smell evolution;
3) an investigation of test fixture smell evolution in five well-known open source systems;
4) strategies and guidance on how to avoid test fixture smells.

In future work, we plan to integrate the TestEvoHound tool in the continuous integration environment in order to give immediate feedback to developers based on the findings of this paper.

References

[1] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

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

[3] S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*, 1st ed. Addison-Wesley, 2009.

[4] K. Beck, *Test Driven Development: By Example*. Addison-Wesley, 2002.

[5] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley, 2003.

[6] A. van Deursen, L. Moonen, A. v. d. Bergh, and G. Kok, "Refactoring test code," in *Proc. of the Int'l Conf. on Extreme Programming and Flexible Processes (XP)*. University of Cagliari, 2001, pp. 92–95.

[7] M. Greiler, A. van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *Proc. of the Int'l Conf. on Software Testing, Verification and Validation (ICST)*. IEEE CS, 2013.

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

[9] B. Van Rompaey, B. Du Bois, and S. Demeyer, "Characterizing the relative significance of a test smell," in *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*. IEEE CS, 2006, pp. 391–400.

[10] H. Neukirchen and M. Bisanz, "Utilising code smells to detect quality problems in ttcn-3 test suites," in *Proc. of the Int'l Conf. on Testing of Communicating Systems and the Int'l Workshop on Formal Approaches to Testing of Software (TestCom/FATES)*. Springer, 2007, pp. 228–243.

[11] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Prentice Hall, 2008.

[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Softw. Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[13] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, 1993.

[14] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, 1996.

[15] M. M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," *Journal of Systems and Software*, vol. 1, pp. 213–221, 1984.

[16] W. G. Hopkins, *A new view of statistics.* Internet Society for Sport Science, 2000.

[17] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 2012, pp. 411–416.

[18] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Trans. on Softw. Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[19] M. Breugelmans and B. Van Rompaey, "TestQ: Exploring structural and maintenance characteristics of unit test suites," in *1st Int'l Workshop on Academic Software Development Tools and Techniques*, 2008.

[20] R. Borg and M. Kropp, "Automated acceptance test refactoring," in *Proceedings of the 4th Workshop on Refactoring Tools (WRT)*. ACM, 2011, pp. 15–21.

[21] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proc. Int'l Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012, pp. 33:1–33:11.

[22] M. Galli, M. Lanza, and O. Nierstrasz, "Towards a Taxonomy of SUnit Tests ?" in *International Smalltalk Conference*, 2004.

[23] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. on Softw. Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

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

[25] R. Marinescu, "Detecting design flaws via metrics in object-oriented systems," in *Proc. of the Int'l Conf. on Technology of Object-Oriented Languages and Systems (TOOLS)*. IEEE CS, 2001, pp. 173–182.